Cisco ParStream

Cisco ParStream Manual

<November 14, 2019>

Document Information:

Title:	Cisco ParStream Manual
Version:	6.2.1
Date Published:	<November 14, 2019>
Date Printed:	November 14, 2019

Table of Contents

Preface

Welcome to the *Cisco ParStream Manual*. This manual describes the Cisco ParStream database.

Note:
Ensure you are familiar with the "***Important Constraints Using Cisco ParStream***" (see chapter 4, page 26) before using Cisco ParStream.

About Cisco ParStream

Cisco ParStream is a *massively parallel* (MPP) *shared-nothing* data management system designed to run complex analytical queries over extremely large amounts of data on a cluster of commodity servers. Cisco ParStream is specifically designed to exploit advantages of modern processor architectures.

License

Cisco ParStream is licensed as a component of the Cisco Kinetic Edge & Fog Processing Module (EFM). Licensing is managed through the Kinetic EFM Smart Licensing Tool.

Key Features

Cisco ParStream can be seamlessly added to your existing environment and processes. Key features include:

- Fast setup (tools support structure and data migrations)
- Fast import, transformation and indexing (file and stream interface)
- All ordinary data types supported (integer, floating-point, date, time, timestamp, string, blob, etc.)
- Easy querying (SQL 2003 Select is supported)
- Advanced analytics (easily extended via user-defined functions /C++ API)
- Schema-based (multi-dimensional partitioning, single and multi-table support, columns and indices)
- Infrastructure independent (running on single servers, dedicated clusters and virtualized private and public clouds)
- Platform independent (many Linux distributions supported)
- Software-only product (no need for special hardware as it runs on standard CPUs, RAM, SSDs, and spinning disks)

Document Audience

The following installation guide is targeted at experienced database and Linux system administrators.

Prerequisites

This document assumes that you have:

- Expertise in SQL and Linux
- A supported Linux operating system
- A hardware platform that meets the minimum requirements

Typographical Conventions

This document uses the following typographical conventions to mark certain portions of text:

Conventions	Description
Italics	New terms, foreign phrases, and other important passages are emphasized in italics.
`Monospaced`	Everything that represents input or output of the computer, in particular commands, program code, and screen output, is shown in a monospaced font.
`Monosp.italics`	Within such passages, italics indicate placeholders; you must insert an actual value instead of the placeholder.
Bold	On occasion, parts of program code are emphasized in bold face if they have been added or changed since the preceding example.

Command Conventions

The following conventions are used in the synopsis of a command:

Conventions	Description
Brackets (`[` and `]`)	Indicates optional parts.
Braces (`{`, `}`)	Indicates that you must choose one alternative.
Vertical lines (`|`)	Indicates that you must choose one alternative.
Dots ('...')	The preceding element can be repeated.
Prompt (=>)	SQL commands are preceded by the prompt =>, where it enhances the clarity.
Prompt (`$`)	Shell commands are preceded by the dollar prompt.

Normally, prompts are not shown in code examples.

Administrator/User

The following defines how the terms *administrator* and *user* are used in this document:

- An *administrator* is generally a person who is in charge of installing and running the Cisco ParStream Server.
- A *user* can be anyone who is using, or wants to use, any part of the Cisco ParStream system.

Note:

The terms *administrator* and *user* should not be interpreted too narrowly. This document does not have fixed presumptions about system administration procedures.

Installation

Installation Overview and Checklist

Pre-Installation Checklist

Ensure you have the following before beginning the installation tasks:

- **Obtain superuser (*root*) permissions** or *sudo* access to all servers in your cluster to perform the installation.
- Download the Cisco ParStream installation package that is part of the Cisco Kinetic Edge & Fog Processing Module.

Installation Tasks

The following is a list of required Cisco ParStream installation tasks. Each task is described in later chapters of this guide:

- **Procure and configure the servers and networking equipment** in accordance with hardware requirements provided in this guide (see section 2.3, page 5).
- **Configure Linux OS** per Cisco ParStream installation pre-requisites on each node in the cluster (see section 2.4, page 7).
- **Install Cisco ParStream Server software** by running the installer on each node in the cluster (see section 2.5, page 10).
- Configure the **Administrative User account "*parstream*"** (see section 2.8, page 11).
- Follow the **Getting Started Tutorial** section in this guide (optional; see section 3, page 15).

Securing the Installation

Every install of a Cisco ParStream instance will have to meet specific requirements for performance and security. It is generally advisable, to configure the underlying platform Linux OS as tight as possible by minimizing the number of amount and privileges of processes running and services offered. Suggested is adherence to general hardening guidelines as provided by the NSA hardening guide collection at `https://www.nsa.gov/` or platform specific formulations (as noted below in section 2.2, page 4). To enable educated decisions, when the grade of security impacts performance, and where to strike a balance acceptable for the local install, the sections in this chapter (starting with section 2.3, page 5) offer helpful information and relations.

Supported Platforms and Packages

This section provides important information about Cisco ParStream supported platforms and lists associated software package information for installation, development and drivers.

Server Packages

Cisco ParStream Server is supported on the following 64-bit Operating Systems on the x86_64 architecture:

Operating System (64-bit, x86_64 architecture)	**Cisco ParStream Server Installation Packages**

Red Hat Enterprise Linux 7
CentOS 7
	`parstream-database-<`*version*`>.el7.x86_64.rpm`
	`parstream-authentication-<`*version*`>.el7.x86_64.rpm`

JDBC Package

ParStream JDBC Driver is provided on the following 32-bit and 64-bit Operating Systems on x86 and x64 architectures:

Java Platform	**Cisco ParStream JDBC Driver Package**
Java 8, all editions	`parstream-jdbc-<`*version*`>.el7.noarch.rpm`

JSII Package

ParStream Java SSI Driver is provided on the Operating Systems on x86 and x64 architectures:

Operating System (64-bit, x86_64 architecture)	**Cisco ParStream Server Installation Packages**
Red Hat Enterprise Linux 7 CentOS 7	`parstream-sii-<`*version*`>.el7.x86_64.rpm`

Security Guidelines

Installs of Cisco ParStream are expected to rely on a system adhering to platform specific security guidelines, where offered by vendor / distributor. The places where to find normative information are subject to change, thus only sample URLs are given here:

Operating System	**Security Guidelines Sample URL**
Red Hat Enterprise Linux 7	`https://access.redhat.com/documentation/en-US/`... `Red_Hat_Enterprise_Linux/7/pdf/Security_Guide/`... `Red_Hat_Enterprise_Linux-7-Security_Guide-en-US.pdf`
Red Hat Enterprise Linux 7 CentOS7	`https://wiki.centos.org/HowTos/OS_Protection/`

Hardware Requirements

This section provides the Cisco ParStream requirements, as well as important details and considerations.

Note:

To eliminate potential resource contention, **do not run any 3rd party applications** on any Cisco ParStream node.

x86_64 Processor Architecture

Cisco ParStream Server software runs on x86-64 architecture hardware. Cisco ParStream software will run on any compliant platform, including virtualized.

Note:

A processor's clock speed directly affects the Cisco ParStream database response time.

- A larger number of core processors enhance the cluster's ability to simultaneously execute multiple massively parallel processing (MPP) queries and data loads.
- A popular, proven, and cost effective platform for the Cisco ParStream cluster node is 2-socket industry-standard server with Intel® Xeon® 6- or 8-core processors, such as Xeon E5-2600 or Xeon 5600 Series.
- The minimum acceptable total number of core processors (not HT) in a server node (e.g., a virtual machine) is 4.

RAM

A sufficient amount of memory is required to support high-performance database operations, particularly in environments with high concurrency and/or mixed workload requirements.

Cisco ParStream requires a **minimum of 2GB per physical CPU core, however 4GB or more per CPU is recommended**. For example, the minimum amount of RAM for a server node with 2 hyper-threaded eight-core CPUs is 32GB (2 CPUs * 8 cores * 2GB), though 64GB is recommended.

This guidance provides a degree of flexibility enabling you to provision RAM in compliance with DIMM population rules to maintain the highest supported RAM speed. On modern Intel Xeon architectures this typically means:

- To utilize the highest supported DIMM speed, all channels should be loaded similarly, i.e., no channel should be left completely blank.
- The maximum number of DIMMs per channel is 2.

Storage

Since Cisco ParStream is designed as a *shared-nothing* MPP system, Cisco ParStream cluster nodes can utilize any storage type – internal or shared/attached (SAN, NAS, DAS) as long as the storage is presented to the host as a Cisco ParStream supported file system and provides a sufficient I/O bandwidth. Internal storage in a RAID configuration offers the best price/performance/availability characteristics at the lowest cost.

The following are guidelines for internal storage provisioning:

- To maximize I/O performance, spread the I/O across multiple individual drives. Cisco ParStream requires at least 4 individual drives dedicated to the Cisco ParStream Data Storage Location. For production environments, provisioning 8 or more drives to the Cisco ParStream Data Storage Location is recommended.

- All internal drives used for Cisco ParStream data storage should be connected to a single RAID controller and presented to the host as **one contiguous RAID device**. This means that a single RAID controller must "see" all available internal drives.

  Note:
  Some servers with multiple internal drive cages requiring separate RAID controllers may have design limitations and are not recommended as Cisco ParStream nodes.

- Select a RAID controller with 1GB or more cache, with write caching enabled.
- Ensure that the storage volume for Cisco ParStream Data Directory is **no more than 60% utilized** (i.e., has at least 40% free space).

Network

Cisco ParStream software forms a cluster of server nodes over an Ethernet network and uses TCP P2P communications.

The network provisioned for operating a Cisco ParStream cluster should be 1GB or greater. In addition

- For best performance, all cluster nodes should reside on the **same subnet** network.
- IP addresses for the cluster nodes must be assigned **statically** and have the same subnet mask.
- We recommend that the cluster network is provisioned with **Ethernet redundancy**. Otherwise, the network (specifically the switch) could be a single point of a cluster-wide failure.

Power Management and CPU Scaling

CPU scaling may adversely affect the database performance.

Note that due to the internal architecture of Cisco ParStream it is important to **disable** any BIOS option for efficient dynamic power management or frequency scaling of CPU's. The reason is that updates due to the dynamic management react too slow so that such a feature is even highly counter-productive for an efficient usage of Cisco ParStream. Thus:

> **Disable any dynamic management of CPU frequencies** (which is more and more enabled by default).

As a result, you should not see different CPU frequencies in `/proc/cpuinfo`.

Although CPU scaling can also be controlled via governors in the Linux kernel, CPU scaling control is usually hardware specific. For background information, see
`http://en.wikipedia.org/wiki/Dynamic_frequency_scaling`.

Configuring Linux OS for Cisco ParStream

This section details the steps that must be performed by the ***root user*** on ***each server in the cluster***.

Note:

All nodes in the cluster must be identically configured.

After making all the changes outlined in this section, restart the servers and verify that the recommended settings are implemented.

Swap Space

Cisco ParStream recommends allocating, at minimum, the following swap space:

System Ram Size (GB)	Min Swap Space (GB)
4 or less	2
4 to 16	4
16 to 64	8
64 to 256	16

The swap file or partition should not be co-located on the same physical volume as the Cisco ParStream data directory.

Data Storage Location

The Cisco ParStream Data Directory should be placed on a dedicated, contiguous storage volume. If internal storage is used, the physical data drives should form one hardware RAID device presented to the host as one contiguous volume.

ext4 is the recommended Linux file system for the Data Storage Location.

Due to performance and reliability considerations, Cisco ParStream **does not recommend using LVM** in the I/O path to the Data Storage Location. Further, Cisco ParStream does not support Data Storage Location on logical volumes that have been extended beyond their initially configured capacity.

IPTables (Linux Firewall)

You should allow access in the firewall for ports used by ParStream depending on your configuration/usage.

For client access use (only open the one you need):

```
$ firewall-cmd --zone=public --add-service=parstream-netcat
$ firewall-cmd --zone=public --add-service=parstream-postgresql
```

If you have configured a cluster, you need additional ports for intercluster communication (you have to open all of the following ports):

```
$ firewall-cmd --zone=public --add-service=parstream-cluster-messages
$ firewall-cmd --zone=public --add-service=parstream-partition-activation
$ firewall-cmd --zone=public --add-service=parstream-find-nodes
$ firewall-cmd --zone=public --add-service=parstream-registration-port
```

You should limit access to the intercluster communcation ports using appropriate firewall rules only allowing access to / from necessary machines (all cluster nodes) or if necessary protect against denial of service attacks using connection rate limits.

See section 13.3.1, page 135 for an overview of the ports Cisco ParStream uses.

SELinux

SELinux is not recommended on cluster nodes as it may complicate cluster operations.

If it is enabled:

- If it doesn't violate your security practices, in the file `/etc/sysconfig/selinux` change the setting for SELINUX to `disabled`:

  ```
  SELINUX=disabled
  ```

- Immediately change the current mode to permissive (until SELinux is permanently disabled upon the next system restart):

  ```
  $ setenforce 0
  ```

  To check the current settings, you can call:

  ```
  $ getenforce
  ```

Clock Synchronization

When using the date/time functions, the clocks on all servers in the cluster must be synchronized to avoid inconsistent query results.

Ensure the NTP package is installed and the system is configured to run the NTP daemon on startup:

```
$ chkconfig ntpd on
$ service ntpd restart
```

To check the current settings:

```
$ chkconfig --list ntpd
$ service ntpd status
```

Verify the level of server's clock synchronization:

```
$ ntpq -c rv | grep stratum
```

A high stratum level, e.g., 15 or greater, indicates that the clocks are not synchronized.

Note:

These instructions are intended for RHEL/CentOS which names the NTP daemon process *ntpd*.

Maximum Number of Open Files

If the error "too many file open" appears, the setting for the maximal number of open files must be increased. Cisco ParStream will print a warning if the value is less than 98,304.

To change the setting, add the following to the file `/etc/security/limits.conf`:

```
*                 hard    nofile          131072
*                 soft    nofile          131072
root              hard    nofile          131072
root              soft    nofile          131072
```

Logout and log back in for the changes to take immediate effect.

To check the current settings, call:

```
$ ulimit -n
```

max_map_count Kernel Parameter

If the error "cannot allocate memory" appears, the Linux kernel parameter *vm.max_map_count* must be increased.

To change the setting, add the following to the file `/etc/sysctl.conf`:

```
vm.max_map_count = 1966080
```

and reload the config file for the changes to take immediate effect:

```
$ sysctl -p
```

Check the current settings as follows:

```
$ cat /proc/sys/vm/max_map_count
```

Installing Cisco ParStream Server

This section details the steps that must be performed by the ***root user*** on ***each server in the cluster***.

Cisco ParStream Server software depends on supplemental Linux packages that may or may not already be installed on your servers. For this reason, your servers should be able to download and install additional Linux packages from official repositories while installing this software.

CentOS, RHEL

Install Cisco ParStream Server software:

```
$ yum install parstream-database-<version>.el7.x86_64.rpm
```

PAM Authentication

The installer will install a new PAM configuration file for the authentication to the Cisco ParStream Server. It will be installed as `/etc/pam.d/parstream`. In order to login with the user *parstream*, you have to follow the instructions in section 2.8.1, page 12.

Cisco ParStream authenticates users via an external application called `parstream-authentication`. This application is provided by an additional software package, that has to be installed separately. For each supported platform, a different package has to be installed. See section 9.2, page 78 for further information about user authentication.

CentOS, RHEL

To install Cisco ParStream authentication software:

```
$ yum install parstream-authentication-<version>.el7.x86_64.rpm
```

Cisco ParStream Installation Directory Tree

The Cisco ParStream installation directory tree is organized as follows:

Path	Description
`/opt/cisco/kinetic/parstream-database`	Cisco ParStream installation directory, pointed to by the $PARSTREAM_HOME environment variable
`/opt/cisco/kinetic/parstream-database/bin`	Executable binaries and scripts
`/opt/cisco/kinetic/parstream-database/lib`	Shared libraries
`/opt/cisco/kinetic/parstream-database/examples`	Examples
`/var/log/parstream`	Message logs

Administrative User '*parstream*'

This section provides *parstream* login and account information.

The Cisco ParStream Server installation procedure automatically creates a Linux user *parstream* if it doesn't exist. The user *parstream* is:

- The owner of the product installation in `/opt/cisco/kinetic/parstream-database`
- The Administrative user of the Cisco ParStream database. The configuration and administration tasks in the Cisco ParStream database environment, including starting and stopping the database server, should be performed by this user.

Note:

The Cisco ParStream database processes **should not** be started with ***root*** user privileges.

Enabling Interactive Login for *'parstream'*

The installation procedure creates the user *parstream* without a pass phrase. You can start an interactive shell session as *parstream* user with `su` command from *root* account. However, all other logins as user *parstream* are disabled until a pass phrase is set.

You can set a pass phrase for the user *parstream* by running the following command as *root*:

```
$ passwd parstream
```

Useful Environment Variables

Define `PARSTREAM_HOME` which is mostly used by the example execution scripts:

```
export PARSTREAM_HOME=/opt/cisco/kinetic/parstream-database
```

The `PATH` variable should be extended by the bin folder of the installation to be able to use commands like `pnc` directly from your shell:

```
export PATH=/opt/cisco/kinetic/parstream-database/bin:$PATH
```

When executing the `parstream-server` or `parstream-importer` manually, the `LD_LIBRARY_PATH` should be set as we use dynamic linking for our executables:

```
export
    LD_LIBRARY_PATH=/opt/cisco/kinetic/parstream-database/lib:$LD_LIBRARY_PATH
```

Systemd

Setting up systemd for Cisco ParStream

Before starting the Cisco ParStream server you need to configure the packaged `parstream` systemd daemon.

To configure the daemon put the following into

`/usr/lib/systemd/system/parstream-database@srv1.service.d/local.conf`:

```
[Unit]
AssertPathExists=/psdata/tutorial
[Service]
WorkingDirectory=/psdata/tutorial
```

You can now control the service using `systemctl` using the servicename `parstream-database@srv1`.

Use `systemctl status` to check its status.

```
* parstream-databasesrv1.service - Cisco ParStream database - srv1Loaded:
    loaded (/usr/lib/systemd/system/parstream-database.service; disabled;
    vendor preset: disabled)
Drop-In:
    /usr/lib/systemd/system/parstream-databasesrv1.service.d|-local.confActive:
    inactive (dead)
```

Use `systemctl start` to start the server. If it fails to start it will not show you any errors - you need to use `systemctl status` afterwards to check its status.

Use `systemctl stop` to stop the server.

Note:

If the initial start of the cluster fails, you have to clean-up any temporary files created for the first failed cluster setup before you start the cluster initialization again. Beware that this will remove all metadata and wipe all data from the server and is not recommened once the system has been running correctly. (there are other options, which go beyond this initial tutorial):

```
$ cd /psdata/tutorial
$ rm -rf journals
```

More information on systemd and how to use and configure it can be found under `https://www.freedesktop.org/software/systemd/man/systemd.html`

Configuring the Cisco ParStream daemon to start automatically

The packaged systemd allows the server to be configured to automatically start and stop with the system.

To register the service use the `systemctl enable` command (it will not implicitly start the server):

```
$ systemctl enable parstream-databasesrv1
```

Use `systemctl disable` to unregister the server (it will not implicitly stop the server).

Note:

If you stop the server using `systemctl stop` it will be restarted on the next reboot. If you use systemd to start your server, you should only use systemctl to stop your parstream server. If you use an ALTER SYSTEM CLUSTER shutdown command, the systemd service will immediately restart the server, which might leave it in an undefined state.

Getting Started Tutorial

This tutorial provides instructions for typical tasks used when operating a Cisco ParStream database. It covers initial setup, loading data, running queries, and basic performance tuning optimizations.

Cisco ParStream Database Software

Before proceeding, ensure that Cisco ParStream is installed on a node or a cluster of nodes according to this installation guide. That is:

- Cisco ParStream database software is installed in `/opt/cisco/kinetic/parstream-database` (see section 2.5, page 10),
- The installed software shall be owned by the user `parstream` (see section 2.8, page 11).
- Environment variables such as `$PARSTREAM_HOME` are set-up (see section 2.8.2, page 12).

Database Data Storage Location

Each Cisco ParStream server node must be configured with a storage volume for database data files. This storage volume should be provisioned as recommended in section 2.4.2, page 8.

In this tutorial, we presume the databases will be created on a dedicated storage volume mounted as `/psdata`. The user `parstream` must have read-write privileges in the directory `/psdata`.

Additional Packages Required to Run the Tutorial

Running this tutorial requires some Linux packages to be installed as follows:

CentOS7 / RHEL 7 / Oracle Linux 7

```
yum install python-argparse nc telnet PyGreSQL
```

General Directory Structure

Cisco ParStream uses the following directory layout for each instance. Data and journal directories are created by Cisco ParStream if necessary.

Path	Description
`conf`	Directory containing configuration files
`import`	Staging directory for the import process
`journals`	Directory containing server metadata
`partitions-[srvname]`	Default directory for data stored inside Cisco ParStream

Create a Minimal Cluster Configuration

All of the following steps in this tutorial are performed by the user `parstream`. First, we need to create the *conf* directory:

```
mkdir -p /psdata/tutorial/conf
```

Secondly, we create a configuration file called `parstream.ini` in this directory.

An example of a minimal configuration file for this tutorial is provided here:

```
clusterId = ParStreamTutorial
registrationPort = 9040

[server.srv1]
host = localhost
port = 9210
rank = 1
datadir = partitions-srv1

[server.srv2]
rank = 2
host = localhost
port = 9220
datadir = partitions-srv2

[server.srv3]
rank = 3
host = localhost
port = 9230
datadir = partitions-srv3

[import.imp1]
host = localhost
leaderElectionPort = 9099
rank = 9999
sourcedir = import
targetdir = partitions-imp1
```

This configuration file defines:

- a cluster '`ParStreamTutorial`' to manage all server and import nodes of this tutorial:
  - All servers and importer use the common registration port '`9040`'.
  - To start the database, we give 120 seconds time (this is especially necessary to start clusters with multiple servers; as we only have one server here, you can set this value to 20).
- a couple of settings for each **server**, called '`srv1`', '`srv2`', and '`srv3`'. listening for client connections (for the moment we need only the first server, but we provide all definitions so that we can keep this INI file when we run the database with multiple servers)
  - Each server has to define a host, i.e., a network address to bind to listen on.

  - Each server has to define an initial port (internally servers use up to 5 ports starting with this port).
  - Each server needs its own rank, which is used to define, which server is the initially preferred server for cluster management
  - Each server defines where to keep the database files (the so-called **partitions**).
- a local data **importer** 'imp1' that will automatically load CSV data files from the specified staging directory
  - Each importer has to define a 'leader election port' for the cluster management and a rank.
  - Each importer needs to define where to find the CSV files to import data from and a partition directory used to temporarily store data while it is being transferred to the server(s).

Copy this template to your *parstream.ini* file.

Provide a Table Definition

To make the server configuration meaningfully complete, we need to define at least one table. Database tables are defined by submitting CREATE TABLE commands (see section 24.2, page 278). For this tutorial, we provide a simple CREATE TABLE statement:

```
CREATE TABLE measurements (sensor_id UINT64 INDEX EQUAL,
                           sensor_name VARSTRING(255) COMPRESSION HASH64
    INDEX EQUAL,
                           value float)
PARTITION BY sensor_id
DISTRIBUTE OVER sensor_id
IMPORT_FILE_PATTERN 'measure.*';
```

Partitioning a table is a way of organizing data. Each unique value of a partitioning column will be used to organize the data physically. Hence, filtering on a partitioning column will greatly reduce the amount of data that needs to be inspected by Cisco ParStream.

The distribution over statement controls how the data is distributed among the cluster nodes. Each value of the column will be assigned round-robin to a different cluster node. Therefore, it would be ideal if the data is distributed evenly among all the different values of that column. For a more in-depth explanation, see section 6.3, page 53.

Start the Cluster

The recommended way to start Cisco ParStream is to use the provided systemd scripts described in section 2.9, page 12. Alternatively, you can start the Cisco ParStream server process from the tutorial's base directory with the following command:

```
/opt/cisco/kinetic/parstream-database/bin/parstream-server <srvname>
```

To start a cluster for the first time, promptly start the Cisco ParStream servers on all cluster nodes within 60 seconds (half of the default time set for option `clusterInitTimeout`).
So start the first server, '*srv1*':

```
cd /psdata/tutorial
/opt/cisco/kinetic/parstream-database/bin/parstream-server srv1
```

server '*srv2*':

```
cd /psdata/tutorial
/opt/cisco/kinetic/parstream-database/bin/parstream-server srv2
```

and server '*srv3*':

```
cd /psdata/tutorial
/opt/cisco/kinetic/parstream-database/bin/parstream-server srv3
```

Wait until all log files contain lines that the cluster is successfully started, which take about two minutes by default. During this period, the log files end with messages such as:

```
...
[2018-10-11T14:42:25]:srv1-144224:PROT-77065: Running cluster leader
    detection (limited until 2015-Mar-13 14:44:25)
```

When the servers are up and running, you should see output similar to this:

```
...
[2018-10-11T14:43:25.321578]:srv1-144224:PROT-77066: Cluster leader detected:
    leader = 'srv1'  I AM THE LEADER  (elapsed time = 00:01:00)
[2018-10-11T14:43:25.587458]:srv1-144224:INFO-77063: Registration of cluster
    node 'srv1' done
...
[2018-10-11T14:43:39.456872]:srv1-144224:PROT-77018: Activating node srv1
[2018-10-11T14:43:39.758525]:srv1-144224:PROT-77011: Starting to listen on
    port 9010 for client connections
```

and this:

```
...
[2018-10-11T14:42:34.654425]:srv2-144233:PROT-77065: Running cluster leader
    detection (limited until 2015-Mar-13 14:44:34)
[2018-10-11T14:43:25.214547]:srv2-144233:PROT-77066: Cluster leader detected:
    leader = 'srv1'  (elapsed time = 00:00:51)
[2018-10-11T14:43:25.578975]:srv2-144233:INFO-77063: Registration of cluster
    node 'srv2' done
[2018-10-11T14:43:35.133581]:srv2-144233:PROT-77093: Cluster follower
    detected: follower = 'srv2'  I AM A FOLLOWER
[2018-10-11T14:43:38.577258]:srv2-144233:PROT-77011: Starting to listen on
    port 9020 for client connections
```

As the messages indicate, the Cisco ParStream servers are up and ready for client connections.

View Processes and Open Ports

To view the processes, run the command:

```
ps x
```

```
   PID TTY      STAT   TIME COMMAND
...
 98440 pts/0    Sl     0:00
   /opt/cisco/kinetic/parstream-database/bin/parstream-server srv1
 98443 pts/0    Sl     0:00
   /opt/cisco/kinetic/parstream-database/bin/parstream-server srv2
 98447 pts/0    Sl     0:00
   /opt/cisco/kinetic/parstream-database/bin/parstream-server srv3
...
```

You can also verify that the processes are listening on the defined ports by running the command:

```
lsof -i
```

If the `lsof` command is not available on your platform, you might have to install the corresponding package, which usually has the same name (e.g., calling `yum install lsof` on CentOS).

```
COMMAND     PID      USER   FD   TYPE  DEVICE SIZE/OFF NODE NAME
parstream 98440 parstream   6u   IPv4 1229966      0t0  TCP *:9210 (LISTEN)
parstream 98440 parstream  10u   IPv4 1229967      0t0  TCP *:9211 (LISTEN)
parstream 98443 parstream   6u   IPv4 1229966      0t0  TCP *:9220 (LISTEN)
parstream 98443 parstream  10u   IPv4 1229967      0t0  TCP *:9221 (LISTEN)
parstream 98447 parstream   6u   IPv4 1229966      0t0  TCP *:9230 (LISTEN)
parstream 98447 parstream  10u   IPv4 1229967      0t0  TCP *:9231 (LISTEN)
```

- 9010 is the default listening port for pnc/netcat ASCII character connections of srv1
- 9011 is the default listening port for ODBC and JDBC client connections of srv1
- 9020 is the default listening port for pnc/netcat ASCII character connections of srv2
- 9021 is the default listening port for ODBC and JDBC client connections of srv2
- 9030 is the default listening port for pnc/netcat ASCII character connections of srv3
- 9031 is the default listening port for ODBC and JDBC client connections of srv3

Using Interactive SQL Utility

Even though no data has been loaded into the database yet, you can start the Cisco ParStream interactive SQL utility `pnc` and create some practice SQL queries to familiarize yourself with the environment.

You can connect to *any query node* of the cluster to issue queries. To call `pnc` you have to pass a user name. By default, a user `parstream` is created, so that `pnc` should be called as follows (passing user name and port of srv1):

```
pnc -U parstream -p 9010
```

The output will be:

```
password:
Connecting to localhost:9010 ...
Connection established.
Encoding:  ASCII
```

After typing in your pass phrase, the prompt signals it is ready to submit commands or queries:

```
Cisco ParStream=>
```

For example, you can get the list of tables in the database by querying a system table:

```
Cisco ParStream=> SELECT table_name FROM ps_info_table;
```

Output:

```
#table_name
```

We receive an empty list because we did not create any table, yet.

Similarly, you can query for details about the current configuration:

```
Cisco ParStream=> SELECT * FROM ps_info_configuration;
...
```

To exit the `pnc` utility, either press `Ctrl-D` or type `quit` at the prompt:

```
Cisco ParStream=> quit;
```

Output:

```
Lost connection.
```

Connect and View Cluster Information

Connect to any node of the cluster, for example node '*srv1*', with the `pnc` utility:

```
\$ pnc -U parstream -p 9010
password:
Connecting to localhost:9010 ...
Connection established.
Encoding:  ASCII
Cisco ParStream=>
```

and type at the prompt:

```
Cisco ParStream=> SELECT name, type, host, port, node_status, leader,
    follower FROM ps_info_cluster_node ORDER BY port;
```

which if everything is up and running should print:

```
#name;type;host;port;node_status;leader;follower
"srv1";"QUERY";"localhost";9210;"active";1;0
"srv2";"QUERY";"localhost";9220;"active";0;1
"srv3";"QUERY";"localhost";9230;"active";0;0
[ 0.001 s]
```

This confirms that all query nodes are active and online and that one node is the *leader* (i.e., currently managing the cluster) and that one of the other two nodes is a so-called *follower* (backup for the leader).

Defining a Table

Using `pnc`, we can create a first table. We can interactively submit a corresponding `CREATE TABLE` command, but we can also call `pnc` again, processing the contents of a corresponding SQL file with the table definition as input:

```
pnc -U parstream -p 9010 < table.sql
```

After connecting and asking for the pass phrase again, the resulting output is:

```
Cisco ParStream=> ....................................
Table 'measurements' successfully created.
```

If we repeat our query for a list of tables now:

```
Cisco ParStream=> SELECT table_name FROM ps_info_table;
```

we get the output:

```
#table_name
```

```
"measurements"
```

View the definitions of the columns in our sample table with the following query:

```
Cisco ParStream=> SELECT column_name, column_type, sql_type, column_size
                     FROM ps_info_column WHERE table_name='measurements'
    ORDER BY column_name;
```

Output:

```
#column_name;column_type;sql_type;column_size
"sensor_id";"numeric";"UINT64";<NULL>
"sensor_name";"numeric";"VARSTRING";255
"value";"numeric";"FLOAT";<NULL>
```

You can also confirm that our sample table has no data:

```
Cisco ParStream=> SELECT COUNT(*) AS cnt FROM measurements;
```

Output:

```
#cnt
0
```

Start the Importer to Load Data

We have configured the import data directory in `/psdata/tutorial/conf/parstream.ini`:

```
...
[import.imp1]
...
sourcedir = /psdata/tutorial/import
...
```

Additionally, there are also *importer*-related parameters configured in each table definition:

```
CREATE TABLE measurements
(
  ...
)
PARTITION BY sensor_id
DISTRIBUTE OVER sensor_id
IMPORT_FILE_PATTERN 'measure.*';
```

Once started, the importer monitors the import directory for new data files with names matching the specified `IMPORT_FILE_PATTERN`. In this tutorial, the *importer* '*imp1*' is configured to continuously load CSV files matching the `'measure.*'` pattern from any directory in or below `/psdata/tutorial/import` into the table `measurements`.

You can create an example csv file and store it in the `/psdata/tutorial/import` directory with a name starting with measure, e.g., `measurement01.csv`:

```
1;'temp-01';23.3
2;'humid-01';13.23
3;'vibr-01';25.23
1;'temp-01';25.3
2;'humid-01';11.23
3;'vibr-01';29.23
```

Verify that the directory contains the created CSV file and then start the *importer* as follows:

```
cd /psdata/tutorial
\$PARSTREAM_HOME/bin/parstream-import imp1
```

You should see output similar to this:

```
Import Process ID: 14989
Output is written to: /var/log/parstream/import-14987-20181013-1324.log
```

Monitor the importer's log file to view the output's progress:

```
tail -f /var/log/parstream/import-14987-20181013-1324.log
```

You should see a line stating the import of the csv files. If you had multiple csv files, the importer would import them in packs of three files by default.

```
...
[2018-10-11T13:24:41.234123]:imp1-132441-0:PROT-77061: **** import file:
    ".../import/measurements01.csv"
```

After some time, the output will be:

```
...
[2018-10-11T13:26:10.142643]:imp1-132441:PROT-77064: Distributing created partition
    '1Z_2018-10-13T12:24:41_imp1_0_PM'
[2018-10-11T13:26:10.647825]:imp1-132441:PROT-77064: Distributing created partition
    '2Z_2018-10-13T12:24:41_imp1_0_PM'
...
[2018-10-11T13:26:11.357618]:imp1-132441:PROT-77064: Distributing created partition
    '3Z_2018-10-13T12:24:41_imp1_0_PM'
[2018-10-11T13:26:11.876354]:imp1-132441:PROT-77041: Will sleep for 6 seconds...
```

The example data is loaded entirely when you see repeated log file messages similar to this:

```
...
[2018-10-11T13:28:02.255485]:imp1-132441:PROT-77046: Table measurements': no files
    with pattern 'measure.*' found to import
[2018-10-11T13:28:02.678525]:imp1-132441:PROT-77041: Will sleep for 6 seconds...
```

```
...
```

You can stop the importer with `Ctrl-C`. All successfully imported csv files will be moved to a hidden `.backup` directory below the import directory. Additionally, all malformed rows that could not be imported will be placed in a secondary hidden folder `.rejected` below the import directory.

Run Queries

To run SQL queries, start the Cisco ParStream interactive SQL utility `pnc` and at the prompt type:

```
Cisco ParStream=> SELECT COUNT(*) FROM measurements;
```

```
#auto_alias_1__
6
[ 0.009 s]
```

This confirms the data was loaded correctly.

Stop the Cisco ParStream Server and the Cisco ParStream Importer

Stop the Cisco ParStream Cluster

To stop the cluster, connect to any server via `pnc` and issue the following command:

```
Cisco ParStream=> ALTER SYSTEM CLUSTER SHUTDOWN;
```

Output:

```
ALTER OK
```

Or, just start `pnc` by typing this command only:

```
echo 'ALTER SYSTEM CLUSTER SHUTDOWN;' | pnc -U parstream
```

Stop a single Cisco ParStream Server

To stop a single server, connect to the server via `pnc` and issue the following command:

```
Cisco ParStream=> ALTER SYSTEM NODE SHUTDOWN;
```

Output:

```
ALTER OK
```

Or, just start `pnc` by typing this command only:

```
echo 'ALTER SYSTEM NODE SHUTDOWN;' | pnc -U parstream
```

Stop the Cisco ParStream Importer

To stop the importer, stop the corresponding process by pressing Ctrl-C or killing the process with `kill` *`pid`*.

Cleanup the Cluster and Restore the Tutorial Environment

To clean up the example and restore the tutorial environment:

Stop any running Cisco ParStream servers and the Cisco ParStream importer (see section 3.14, page 24).

Delete the database that was created in previous steps:

```
rm -rf /psdata/tutorial
```

Important Constraints Using Cisco ParStream

Cisco ParStream is optimized to provide incredible performance (especially speed) when analyzing a huge amount of data, concurrently imported. These optimizations are only possible with a specific design that leads to some constraints that might be surprising for ordinary database administrators and users.

Some of these constraints violate the usual expectations in key concepts of databases and SQL. So, everybody dealing with Cisco ParStream should know them to avoid buggy assumptions, design, or computation.

This chapter lists these overall constraints, where common expectations are not met. Note also the limitations and constraints for specific Cisco ParStream features, mentioned where the features are introduced and described.

Important General Constraints with Data Types

- **Special Values**
  All numeric types have special values to deal with `NULL`. These special values can't be used as ordinary values. Numeric values usually use the largest positive value as `NULL`.
  For example:
  – For a type `INT8` the value 127 can't be used as ordinary value.
  – For a type `UINT8` the value 255 can't be used as ordinary value.
  If the values may occur, use a type with a larger value range. See section 23.2, page 267 for details.
- **Empty Strings and `NULL`**
  Strings use the empty string internally as `NULL`. Thus, imported empty strings become `NULL` and are or have to be handled as `NULL` in queries.
  See section 23.5.1, page 273 for details.
- **`BLOBS`**
  A Cisco ParStream `BLOB` is technically a `CLOB`.

Important General Constraints with SQL Commands

- **`UNIQUE`** and **`PRIMARY KEY`**
  Columns marked with `UNIQUE` or `PRIMARY KEY` do not guarantee uniqueness of the data. The keywords are a hint that the Cisco ParStream engine can assume that data is unique, but this assumption has to be ensured by the user importing data. Thus, `UNIQUE` means "assume that the data is unique."
  The reason for this behavior is that checking, whether a value already exists in the database, would take too long and slow down imports significantly.
- **`DEFAULT`**

`DEFAULT` values for columns are only effective when columns are added, or when streaming import is used.
CSV imports and INSERT INTO imports currently require also to pass data for columns with default values.

Important General Constraints when Importing Data

- **No imports of rows containing only NULL/empty values**
  Cisco ParStream never imports data, where all values of all columns/fields are `NULL` or empty strings.
  See section 10.2, page 89 for details.
- **Imports and `UNIQUE` and `PRIMARY KEY`**
  As mentioned above, the same value can be imported even if a column is marked as `UNIQUE` or `PRIMARY KEY`.
  The reason for this behavior is that checking, whether a value already exists in the database, would take too long and slow down imports significantly.

Data Loss Prevention

The persistent storage of data is one of the most crucial tasks for a database system. Once the database signals that data has been imported correctly, the database has to make certain that data has been stored in such a way that even in the case of a complete power outage, the data will be available to the user in the future. Therefore, the database system stores the data in non-volatile memory, e.g., a hard drive. Unfortunately, this is a costly operation and over the course of time, many optimizations and caching layers were introduced to alleviate the long waiting time for disk access. Hence, we need to take a closer look at these caching layers to understand when data is really stored persistently and which steps we have to take to guarantee a smooth operation.

Every write operation by a process uses an API of the operating system. The operating system in turn signals back when it has stored the data. The operating system caches multiple writes in memory to reorder and write them more efficiently to disk, even though it has already signaled to the process that the data was written successfully. From the processes point of view, the data has been written to persistent storage and the operating system will fulfill that contract even if the process is killed, quits, or crashes. However, if the operating system loses power (or a hard reset is performed), the contract can no longer be fulfilled and data is lost. Therefore, we recommend equipping your servers with Uninterruptable Power Supply (UPS) units that bridge the short amount of time of the power loss and if that is not possible, initiate a proper shutdown of the system. The API of the operating system provides different options to prevent data loss in the case of an interrupted service by allowing a process to signal the operating system to circumvent the caching and directly write the data to disk. One of these options are the fsync operations, which only return once the data is persisted. This is a costly operation and can be enabled in Cisco ParStream using the “synchronizeFilesystemWrites” config option. However, beneath the operating system are even more caching layers that play a crucial role in persisting data correctly.

Following the caching of the operating system, the data is sent to the RAID or hard drive controller which in turn usually provides a caching to improve write rates. There are different kinds of hard drive caches that behave differently in the case of a power loss. We will only focus on one property that is most important for our analysis: Battery Backed Write Cache. If the RAID controller is equipped with a battery, all write operations that are cached will be written to the disk even if the system loses power. We recommend to only use battery backed RAID controllers. If your RAID controller is not equipped with a battery for the write cache, all the data still contained in the write cache will be lost in the case of a power outage. Therefore, you need to disable the write cache in such RAID controllers at the cost of performance.

The persistent storage should be set up as a RAID to reduce the risk of data loss in case of disk failure. Each RAID mode offers different performance and fault tolerance characteristics. Additionally, the system administrator should monitor the remaining disk size regularly to react early to the risk of running out of storage space.

Database Design

The name *ParStream* reflects the main concepts of the Cisco ParStream system:

- **Par** - from the word parallel, representing the nature of parallel executions of a single query
- **Stream** - for streamed query execution, similar to an assembly line or pipelining within a CPU

This chapter describes some of the design concepts and features of Cisco ParStream uses to produce high performance for huge amounts of data. Understanding these features is useful for you, a sophisticated user of Cisco ParStream. These include:

- Management of its data in **partitions**
- Management of schema changes with **metadata versions**
- The fundamental concept of **bitmap indices**

Data Partitioning

Cisco ParStream physically splits the data of a table into data partitions. Partitioning allows new data can be appended to a table and allows a single query to be executed by multiple CPU cores.

There are two types of partitions, logical and physical:

- **Logical partitions** are distinct parts of the same table. Combined, all logical partitions of a single table comprise the complete table data.
- **Physical partitions** are used to manage and manipulate logical partitions. In fact, a logical partition can consist of multiple physical partitions. Some advantages of multiple physical partitions are:
  - Each import creates separated physical partitions. If data of a logical partition is imported in multiple steps (with different imports and/or multiple times), it will produce multiple physical partitions.
  - If a logical partition is too large to be managed as one physical partition, you can split it by configuring the limit of physical partitions.
  - Temporarily, you might need multiple physical partitions to merge or replace data. In that case, the physical partitions reflecting the current state are *activated* while the other physical partitions are not activated.

The way Cisco ParStream creates logical partitions is data dependent instead of "basically randomly" splitting data according to its size. In fact, the different partitions of one table rarely have the same size.

Having partitions with different size sounds counterproductive to query latency; however, this approach is an important improvement in data processing speed. By analyzing queries, Cisco ParStream can exclude partitions in query processing that do not match the SELECT clause in an SQL statement. For example, if you have data about humans, and most queries ask for data about men, partitioning by gender would make a lot of sense, so that the women's data partition need not be analyzed. In addition, the compression ratio of a partition is usually better if the data within it is somehow related. For example: Men tend to be taller than women, so partitioning by gender produces two data groups with people whose heights are closer together and are thus more compressible than those with random partitioning.

You can specify as many partitioning columns as you want using the `PARTITION BY` clause:

```
CREATE TABLE MyTable

(
  dt DATE INDEX RANGE INDEX_GRANULARITY DAY,
  user VARSTRING COMPRESSION HASH64 INDEX EQUAL,
)
PARTITION BY dt, user
...
```

You can even use expressions as partition criterion:

```
CREATE TABLE MyTable
(
  num UINT32 INDEX EQUAL,
  name VARSTRING COMPRESSION HASH64 INDEX EQUAL,

PARTITION BY (num MOD 25), name
...
```

The number of partitions created by that partitioning column is the number of distinct values in that column.

Although there are no technical limits to the number of partitioning columns, it is reasonable to limit their number to 2-15, depending on the number of distinct values of the used columns: Using a column representing floating point values is less desirable than using a Boolean or integral column. If only using Boolean columns, with 20 partitioning columns you create about one million partitions (2ˆ20).

Currently, Cisco ParStream requires the database administrator (DBA) to choose suitable partitioning criteria. Inefficient partitioning criteria can significantly increase the query response time, or decrease the performance of the Cisco ParStream import.

Ideally,

- there should be at least as many partitions as there are CPU cores,
- the partitions should be of similar size, and
- there should be no partitions that contain less than a few thousand rows.

Because continuously appending data to a table is an intended usage scenario for Cisco ParStream, there is a mechanism that aggregates small partitions into larger partitions at regular intervals.

Information:

For tables with few data rows, logical data partitioning can be disabled.

Merging Partitions

When new data is appended to a table, this data is always put into new physical data partitions, even if logical data partitions with the same values in partitioning columns already exist. To avoid generating more and more physical data partitions, Cisco ParStream can merge physical data partitions that belong to the same logical partition at regular intervals.

Cisco ParStream has two default merges: hourly, that merges new partitions that were created over the course of the previous hour (24 hourly merges per day), and daily, that merges partitions that were created over the course of the previous day.

The merge intervals are just abstractions and are configurable. Thus, an “hourly” merge might happen after 5 minutes or 10 hours. You can also have “weekly” and “monthly” merges. So, you have 4 level of configurable merges.

Please note that for Java Streaming Import (see chapter 19, page 216) partitions are even merged every minute. This is because external sources might create partitions every second, which might result into poor performance if these partitions are only merged after one hour.

You can limit the maximum number of rows of a partition, `partitionMaxRows` (see section 13.2.1, page 126) to avoid getting partitions that are too large. For a merge, the criteria for `partitionMaxRows` is the number of rows in the source partitions. That ensures that the merge result is never split up into multiple partitions and avoids merges that may be partially performed or even skipped.

During a merge, the data can be transformed or purged using ETL merge statements. The number of rows can be reduced by combining multiple rows using a `GROUP BY` clause (see section 14.2, page 154 for an example). Using this, merges are never skipped and might result in one source partition being replaced by a (possibly smaller) transformed partition.

Partition merge commands leave behind unused partition directories. These are deleted shortly after the merge as soon as a rollback of the merge is no longer possible.

See chapter 14, page 151 for further details about merging partitions.

Partitioning by Functions

It is possible to specify a function instead of a real column for a partitioning value. This is especially useful for timestamp columns where you only want to use part of the date and time information. The example below uses only the month part of a timestamp for the partitioning.

```
CREATE TABLE MyTable
(
  dt DATE INDEX RANGE INDEX_GRANULARITY DAY,
  user_group UINT32 INDEX EQUAL,
)
PARTITION BY DATE_PART('month',dt), user_group
...
```

You can use all available SQL functions and expressions. Note that you have to use parentheses for an expression:

```
CREATE TABLE MyTable
(
  lon DOUBLE INDEX EQUAL INDEX_BIN_COUNT 360 INDEX_BIN_MIN 0 INDEX_BIN_MAX
    360,
  lat DOUBLE INDEX EQUAL INDEX_BIN_COUNT 180 INDEX_BIN_MIN -90 INDEX_BIN_MAX
    90,
  point_in_time UINT32 INDEX EQUAL,
)
PARTITION BY (lon MOD 90), (lat MOD 45), (point_in_time MOD 43800)
...
```

Note:

- A partitioning value that is calculated by a function or expression is not allowed as distribution column.
- A column appearing as function argument in a "partitions" expression must be unique within that expression. Otherwise the partitioning behavior is undefined.

Partitions Layout

Cisco ParStream data is stored in regular files in the filesystem. In fact, each physical partition is stored as a directory with a path and filename according to the current partitioning.

Path of Partition Directories

For example, if you import data

- into datadir `./partitions`
- into table `Hotels`
- logically partitioned by
  - a hashed string for the column `city`
  - a Boolean value (integer `0` or `1`) for column `seaview`
  - a hashed string for the column `bedtype`

each import creates a partition directory, as in this example:

```
./partitions/Hotels/12129079445040/0/12622392800185Z_2012-09-05T15:09:04_first_42_PM
```

In this instance, the path has the following elements:

- the name of the targetdir/datadir.
- the table name (`Hotels`)
- the “path” of the logical partition values:
  - `12129079445040` as hash value for the city
  - `0` as for the Boolean value for the “seaview”
  - `12622392800185` as hash value for the “bedtype”

- a `Z_`
- followed by the timestamp of the creation `2012-09-05T15:09:04` of the physical partition,
- followed by the node that created the physical partition (name of the importing or merging process)
- followed by the sequence number, an increasing number greater than 0 for an imported partition (or 0 if this partition was created by merge)
- and a final suffix for the partition type (here: `_PM`).

If you later import additional data for the same logical partition, you will get another directory for the corresponding physical partition having the same path with a different timestamp and a different sequence number.

If you get multiple physical partitions due to size limits (i.e., partitions that have the same import date and the same import process), the name of the partition directory is extended by "`_1`", "`_2`", and so on.

If no logical partitioning is defined, the physical partition files would be named something like:

```
./partitions/Hotels/Z_2012-09-05T15:09:04_first_43_PM
```

That is, all data of an import would be stored in one physical partition having the timestamp and importing process as (directory) name.

Partition File Naming Conventions

Imported partitions might have one of two partition name suffixes:

- `_PM`, stands for "minute" partitions, which are initially created by CSV imports and `INSERT INTO` (see chapter 10, page 88).
- `_PS` stands for "seconds" partitions, which is used if data is imported via Java Streaming Imports (see chapter 19, page 216).

Different partition name suffixes exist because of Cisco ParStream's merge concept (see section 5.1.1, page 31). Partition merges can merge:

- "seconds" partitions to "minute" partitions,
- "seconds" and "minute" partitions to "hour" partitions,
- "seconds", "minute", and "hour" partitions to "day" partitions,
- and analogously for "week" and "month" partitions combining data of a "week" or a "month".

See section 14.1, page 151 for details.

Note that the names "minute", "hour", etc. are just pure abstractions for initial imports and merged partitions. You can define when merges from one level to the next apply and therefore indirectly define your understanding of an "hour" or "day".

Merged partitions might also have a filename ending with other suffixes, such as `_PH` for "minute partitions" that have been merged into "hour partitions". In that case the name before the suffix is the name of the node that caused the merge. The following suffixes are provided:

Partition Suffix	Meaning
`_PS`	initial “seconds” partitions (only created by streaming imports)
`_PM`	“minute” partitions which are initially created by CSV imports and `INSERT INTO`, and are the result of merging `_PS` partitions
`_PH`	“hour” partitions (merged `_PS` and `_PM` partitions)
`_PD`	“day” partitions (merged `_PS`/`_PM`/`_PH` partitions)
`_PW`	“week” partitions (merged `_PS`/`_PM`/`_PH`/`_PD` partitions)
`_PF`	“month” or “final” partitions (merged `_PS`/`_PM`/`_PH`/`_PD`/`_PW` partitions)

The sequence number will always be 0 for merged partitions.

Contents of Partition Directories

Inside partition directories, you can have several files on different levels:

Filename	Meaning
`partition.smd`	general partition information (status, number of rows, versions)
colName	column store file of the partition (name according to column name)
`*.pbi`	bitmap index files (prefix has column name)
`*.sbi`	bitmap index (old format)
`*.map`	value map for variable length data such as non-hashed strings and multivalues
`*.hs`	hashed string lookup data
`*.hsm`	hashed string lookup map
`nrows.bm`	file to deal with rows having only NULL values

Note that the system table `ps_info_mapped_file` (see section 26.4, page 320) lets you see which of these files is currently mapped into the database.

Partition State

A partition can have one of the following states:

Partition State	Meaning
`incomplete`	The partition gets created and is not ready for usage yet.
`active`	The partition is active
`disabled-by-merge`	The partition is disabled by a merge (see section 5.1.1, page 31)
`disabled-by-unload`	The partition is disabled by an unload command (see section 16.4.2, page 204)
`offline`	The partition is not activated yet

Schema/Metadata Versioning

Cisco ParStream can modify its schema by calling statements such as `CREATE TABLE` and `ALTER TABLE` to add and modify tables (see chapter 24, page 277) or by calling `CREATE FUNCTION` to register user-defined functionality (see section 20, page 232). These modifications illustrate the versioning concept of the Cisco ParStream schema/metadata.

To handle different schema states, Cisco ParStream internally uses metadata versions. Currently, these versions use integral numbers which increase with each schema modification. Thus, the first table definition, Cisco ParStream's metadata version has the value 1, with a seconds table definition the metadata version becomes 2, after adding a column to a table, the metadata version becomes 3, and so on.

Note that the fact that currently integral values for metadata versions are used, might change over time. Therefore, if exported, the version is exported as string.

The schema and all its versions are stored in journals, located in journal directories, and are defined by the global option `journaldir` (see section 13.2.1, page 122).

If data is written into partitions, the metadata version used is stored within the partitions.

Note that the version stored for a partition is the version with the last definition for the table.

- You can query the metadata version of existing tables with the system table `ps_info_table` (see section 26.3, page 309).
- You can query the metadata version of loaded partitions with the system table `ps_info_partition` (see section 26.4, page 316).

Bitmap Indices

Cisco ParStream uses bitmap indices, which are stored into the physical partitions as index files (e.g., with suffix `.pbi` or `.sbi`, see section 5.1.3, page 34). This is one of the basic features Cisco ParStream uses to produce its excellent performance.

For a general introduction of the bitmap index concept, see, for example, `http://en.wikipedia.org/wiki/Bitmap_index`. That page also provides links to get additional information.

This section presents a few simple examples, which are important to aid in understanding other topics in this manual. You can use the bitmap optimizations by specifying a specific index. See section 24.2.6, page 293 for details.

Equal Index

The equal bitmap index is suited for queries that use one or a small number of values. The indices of different fields can be combined.

row	field1, field2	a	b	c	d	e	1	2	3	4
1	a, 1	x					x			
2	b, 2		x					x		
3	c, 3			x					x	
4	a, 4	x								x
5	b, 1		x				x			
6	a, 2	x						x		
7	c, 3			x					x	
8	a, 4	x								x
9	a, 1	x					x			
10	a, 2	x						x		
11	b, 3		x						x	
12	c, 4			x						x
13	d, 1				x		x			
14	e, 2					x		x		

Now let's look what happens on queries on this index type.

If we select one index:

```
SELECT * WHERE field1 = 'b';
```

the index `b` is used to select rows 2, 5, 11:

```
"b";2
"b";1
"b";3
```

But if we select **two** indices:

```
SELECT * WHERE field1 = 'b' AND field2 = 2;
```

the `AND` of the two indices is computed:

row	b	2	result
1			
2	x	x	x
3			
4			
5	x		
6		x	
7			
8			

9			
10		x	
11	x		
12			
13			
14		x	

and row 2 is returned:

```
"b";2
```

Range Index

The range bitmap index works the same as the equal indices but with the columns representing values within a range:

row	field1, field2	<= a	<= b	<= c	<= d	<= e	1	2	3	4
1	a, 1	x	x	x	x	x	x			
2	b, 2		x	x	x	x		x		
3	c, 3			x	x	x			x	
4	a, 4	x	x	x	x	x				x
5	b, 1		x	x	x	x	x			
6	a, 2	x	x	x	x	x		x		
7	c, 3			x	x	x			x	
8	a, 4	x	x	x	x	x				x
9	a, 1	x	x	x	x	x	x			
10	a, 2	x	x	x	x	x		x		
11	b, 3		x	x	x	x			x	
12	c, 4			x	x	x				x
13	d, 1				x	x	x			
14	e, 2					x		x		

Binned Index

Two or more values are binned into one index. The distribution of values into one index can be equally distributed or custom defined. Custom defined bins should be used if you are always querying the same values.

row	field1, field2	a	b, c	d, e	1	2	3	4
1	a, 1	x			x			
2	b, 2		x			x		

3	c, 3		x				x	
4	a, 4	x						x
5	b, 1		x		x			
6	a, 2	x				x		
7	c, 3		x				x	
8	a, 4	x						x
9	a, 1	x			x			
10	a, 2	x				x		
11	b, 3		x				x	
12	c, 4		x					x
13	d, 1			x	x			
14	e, 2			x		x		

Sorting criteria

An index for unsorted data is larger than an index for sorted data. This effect is influenced by the selected compression algorithm. But even using a very simple RLE (Run Length Encoding) produces a much smaller index on sorted data. You can sort on additional fields, but the effect on the index is not as big.

row (in CSV)	field1, field2	a	b	c	d	e	1	2	3	4
1	a, 1	x					x			
9	a, 1	x					x			
6	a, 2	x						x		
10	a, 2	x						x		
4	a, 4	x								x
8	a, 4	x								x
5	b, 1		x				x			
2	b, 2		x					x		
11	b, 3		x						x	
3	c, 3			x					x	
7	c, 3			x					x	
12	c, 4			x						x
13	d, 1				x		x			
14	e, 2					x		x		

See section 15.4, page 165 for an approach to use this benefit.

Examples for Bitmap Indices

IPv4 Address

The IPv4 Address (see `http://en.wikipedia.org/wiki/Ipv4`) is a 32-bit, unsigned integer value.

Cisco ParStream allows you to import IPv4 Addresses in several formats:

Import as 4 uint16/uint8 fields according to the dot notation

Warning:

If one or more parts of the address can be set to 255 (0xFF), uint16 must be used, because UINT8_MAX is used as NULL inside Cisco ParStream.

Query examples: Find all rows with an IP address

```
SELECT * FROM ip WHERE ip_1=192 AND ip_2=168 AND ip_3=178 AND ip_4=32;
```

Import as uint64/uint32

Warning:

If the address 255.255.255.255 (0xFFFFFFFF) is needed as a representation, uint64 must be used, because UINT32_MAX is used as NULL inside Cisco ParStream.

Sales Volume

If you have a large number of companies stored inside your table and use an equal index, you will get one index per value inside your data. However, you use limiting values in your query most of the time.

Query examples: Selecting all companies where the sales volume is greater or equal to the limiting value of 10,000:

```
SELECT * FROM company WHERE sales_volume >= 10000;
```

Because your query hits exactly a limiting value, Cisco ParStream does an AND on two bitmap indices (first `>= 10000 => < 20000`, second: `>= 20000`) and can return the result set after one bitmap operation. If you have more indices configured, additional bitmap operations are necessary.
Selecting all companies where the sales volume is greater or equal to the non limiting value

```
SELECT * FROM company WHERE sales_volume >= 15000;
```

The same code is executed, but with an additional filter step which excludes the values < 15000.

Clustering and Distribution

Cisco ParStream is a database designed for big data scenarios. This means that it must be able to scale and to be highly available:

- Scalability can be reached by being able to distribute requests over multiple servers and providing load balancing.
- High availability can be reached by providing backup servers and providing failover support.

Both features are supported by the "*clustering*" abilities of Cisco ParStream.

High Availability and Scalability

To support high availability and scalability, Cisco ParStream uses the following concepts:

- Because any Cisco ParStream database provides the ability to run with multiple servers, you can have both failover and load balancing, where imports and merges are even possible when nodes are temporarily offline. You can even add new servers (cluster nodes) at runtime.
- *Partition Distribution* lets you define how values are distributed in the cluster according to the value of a specific column. This provides both failover and load balancing.

Note the following general remarks:

- High availability can be achieved by using partition distribution lists with backup servers. Depending on how many redundant backup servers are used for each partition hash, one or more query-slaves can become inoperative without affecting the availability. Nevertheless, the query performance will begin to degrade, if the optimal parallelism cannot be achieved with the remaining servers.
- Replication of data can be achieved by using partition distribution lists with backup servers. Depending on how many redundant backup servers for each partition hash are used, one or more query-slaves can become inoperative without affecting the availability.
- For the configuration of partition distribution lists, see section 6.3, page 53.

Running Cisco Parstream with Multiple Server Nodes

- **Enhanced redundancy and fault tolerance:**
  As long as at least half of the nodes are operational, the cluster will work continuously. This means, it can import data and answer queries. However, queries can be processed successfully only if the operational nodes have all partitions that are required to answer that query.
- **Continuous import:**
  Even when some cluster nodes are down, it is now possible to continuously import data as long as at least one target node for every partition distribution value is still online. If an importer finds that for a given partition distribution value there is no distribution target node left online, it will temporarily stop importing and wait until one of the offline nodes rejoins the cluster.
- **Automatic resynchronization of nodes after a downtime:**

When a node receiving partitions from an importer fails, it will miss data which was imported during its downtime. Therefore, before the node is reintegrated into the cluster, it will be automatically synchronized, i.e., it downloads the missing data partitions from one of the other nodes. This is automatically initiated by the leader node.

- **Adding new servers/nodes:**
  After the initial setup and while the cluster is up and running, you can add new nodes to distribute newly imported data over more servers.

Note the limitations at the end of this section (see section 6.2.6, page 52).

See section A.1, page 392 for a complete example.

Terminology and Basic Concepts

Note the following basic concepts and terminology for clustering:

- A cluster consists of **cluster nodes** or, in this context, just **nodes**, which represent a logical process instance in the cluster that can fulfill query requests, import data, and/or manage the cluster. Nodes can run on the same or different hosts, but when they use the same host the server nodes have to use different ports (note that up to 5 consecutive ports are used; see section 13.3.1, page 135).
- A cluster node can either be
  - a **query node** (server), which you can use to send query statements, or
  - an **import node** (importer), which you can use to import new data.

  A cluster node can't be both.
- Each cluster node can have different states:
  - **offline**: The node is not available for cluster management or for queries.
  - **online**: The node is available for cluster management but not for queries.
    This state is used to allow cluster nodes to synchronize data and metadata so that they are in a valid state, when they become active
  - **active**: The node is available for both cluster management and for queries.
- At any time a cluster has one **leader node**, also referred to as just **leader**, which organizes the cluster:
  - Informs all nodes about the current **state** of the cluster
  - Initializes **merges** (see section 14.1, page 152)
  - Deals with partially **failed imports** or **failed merges**

  The functionality of the leader node is not used to query data or to import data (unless the data import partially failed). Normal queries and imports are completely distributed and scale well.
- A query node can be a **follower node**, also referred to as just a **follower**, if it is used as fallback for the leader. Any online query node can become a follower. The minimum number of follower nodes that must exist to run the cluster is $n/2$ query nodes.
- The question which query node becomes a leader (or follower) is determined according to a so-called **rank**, which is an integral value. The lower the rank, the more likely the node is to become a leader. Initially, the query node with the lowest rank becomes the leader and the nodes with the

next lowest ranks become followers. If due to a failure other nodes become leaders or followers they keep those roles even if the failed nodes become available again.

- All nodes, in the cluster, including import nodes, must have a
  - **unique name**
  - **unique rank**
- The query nodes of the cluster are usually specified and elected during the **initial start-up** of the cluster. After the cluster is initialized the set of query nodes in a cluster is rather stable, though you may have "known" nodes that are (temporarily) unavailable.
- **New query nodes** can be added while the cluster is running. This does not lead to a redistribution of existing data. Instead, *newly* imported data will be distributed to this node, provided new values get imported and dynamic partition distribution is used.
- **New import nodes** can be added/used at any time after the cluster is initialized and running.

Several options allow to influence the behavior of clusters. See section 13.2.2, page 130 for details.

Setting-Up and Starting a Cluster

To start a cluster you need the necessary configurations (see section 6.2.3, page 45) and then have to start all query nodes (servers) within a given timeframe. During this initial timeframe all nodes get in contact with each other to find out which leader node manages the cluster (as long it is online). This is described in the next subsection.

Note that in the cluster example (see section A.1, page 392) there is a shell script `run_servers.sh`, which you can use as example to start and run multiple servers of a cluster. It starts multiple servers, checks whether and when they are up and running, prints possible errors, and stops the servers with <return> or Ctrl-C. Note that stopping the servers is done just by killing them all with `kill -9`, which you should not do in your mission-critical database control scripts (see section 6.2.6, page 51). See section 9.2, page 78 for how to use user authentication in a cluster.

Leader Election

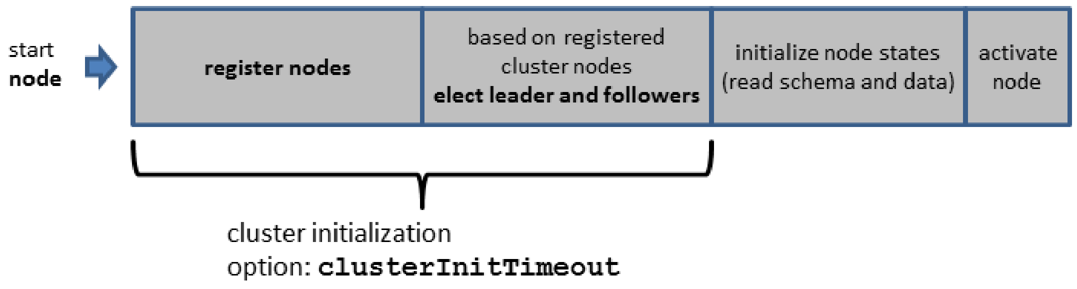
As just introduced as terminology and basic concept, the cluster management uses a "leader-follower" approach:

- A *leader* manages the relevant cluster state data and propagates it to the followers.
- A *follower* will take over in case of the failure of the leader.

That is, while the cluster is in operation, exactly one of the nodes is the leader. The leader is responsible for managing cluster metadata and distributing it to the other nodes, for dealing with partially failed imports and for merges. For redundancy purposes, the leader has exactly $n/2$ followers which replicate all the metadata so that they can take over when the leader goes down, where n is the known number of query nodes in the cluster.

In order to determine the leader of a cluster the option `rank` (see section 13.3.1, page 134 and section 13.4.2, page 147) is used. When the cluster starts for the very first time, a **leader election** takes place. During this election phase, each node announces its rank to all other nodes. Then an *elimination algorithm* is used: If two nodes compare their rank, the node with the higher rank gets eliminated for the leader election. This is done until the leader election phase ends.

The following figure shows what happens when a cluster node is started:


start
node
register nodes
based on registered cluster nodes
elect leader and followers
initialize node states (read schema and data)
activate node
cluster initialization
option: clusterInitTimeout

Thus, in each started query node you have the following phase:

1. The first period is used to let the nodes register to each other. Thus, a node sends out and receives registration requests. Each node with a rank higher than another node gets eliminated, so that finally the leader node becomes clear.

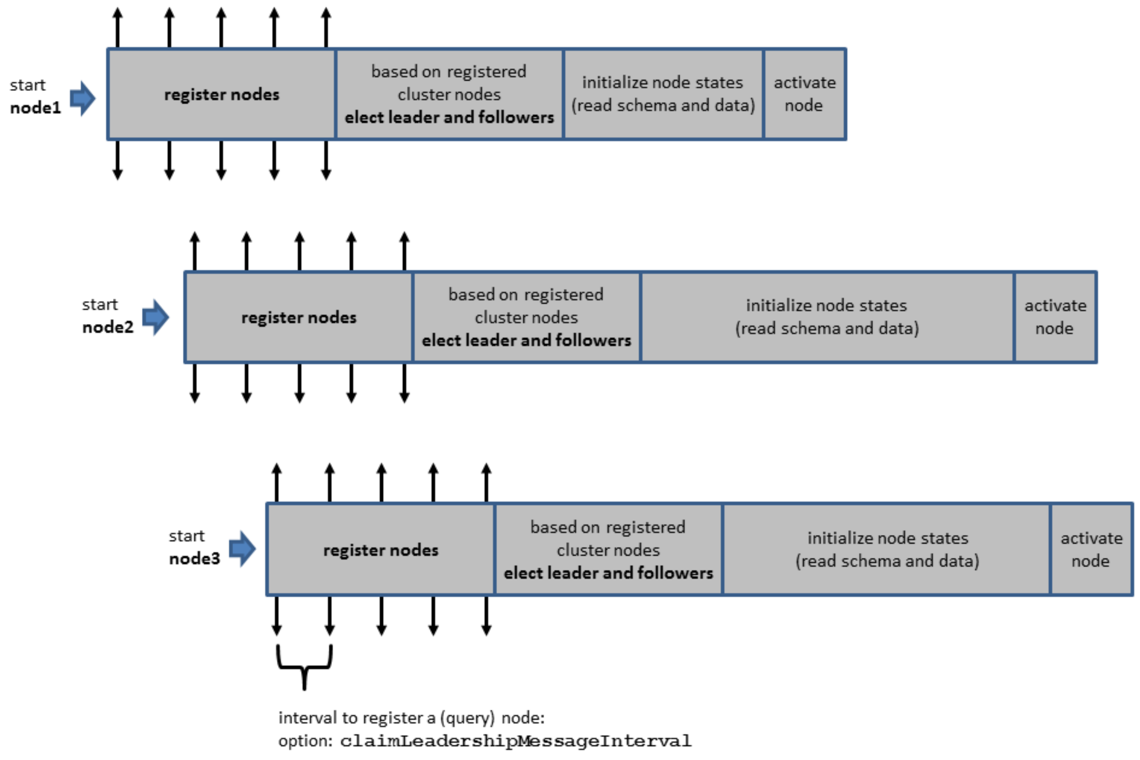
2. Then, each node registers itself to the leader node and the leader distributes the initial state of the cluster.

3. Then, each cluster node has time to establish its local state.

4. Finally, the cluster node gets activated.

You can control, how long the first two phases, which are used for *cluster initialization*, take with the global option `clusterInitTimeout` (see section 13.2.2, page 131).

The following figure shows how multiple cluster nodes register to each other happens when a cluster node is started:


start
node1
register nodes
based on registered cluster nodes
elect leader and followers
initialize node states (read schema and data)
activate node
start
node2
register nodes
based on registered cluster nodes
elect leader and followers
initialize node states (read schema and data)
activate node
start
node3
register nodes
based on registered cluster nodes
elect leader and followers
initialize node states (read schema and data)
activate node
interval to register a (query) node:
option: claimLeadershipMessageInterval

Thus, to initially start a cluster:

- You have to **start all servers (query nodes) within the first half of `clusterInitTimeout`** period so that they have time to register to each other. Thus, this option should be set to a value that is twice the time you need to start all cluster nodes plus some seconds for the handshake.
- Option `clusterInitTimeout` also applies as default to cluster reinitializations (cluster restarts). However, you can specify an alternative period with option `clusterReinitTimeout` for such a reinitialization (see section 6.2.2, page 45).
- You can control the interval for sending out registration messages to the other nodes with the option `claimLeadershipMessageInterval`. The default value is every five Seconds with a maximum of an eighth of the whole cluster initialization period to ensure that at least four registration messages go out during the registration phase.

As written, the node with the lowest rank will initially become the leader. However, if a leader goes offline/down, another node will become the leader and will remain to be the leader until it gets offline. Thus, the rank is only a hint for becoming the leader, but any query node can become a leader and then remains to be a leader as long as it is online. For this reason, it can happen that not the query node with the lowest rank is the leader. This strategy was chosen to minimize leader changes.

Again, please note, that the configured rank must be unique within a cluster. If the rank is ambiguous or not specified, you can't start a node inside the cluster. This also applies to import nodes.

Cluster Reinitialization and Leader Re-Election

If the leader is no longer available, leader re-election is triggered, which roughly uses the same mechanisms as described above. However, unlike the first initialization the actual current cluster nodes are already known by the former leader and all followers. For this reason, you can give `clusterReinitTimeout` (see section 13.2.2, page 131) a smaller duration than for `clusterInitTimeout` (the default is that both intervals are equal).

During re-election the preference is not to change leadership (even if the former leader was not the initial leader). Thus, leadership only changes if the leader is not available.

To prevent from a “split brain” situation, where two leaders in a cluster evolve, a new leader can only be successfully elected if more than half of the query nodes of the cluster accept the new leader. This means that

- A cluster reinitialization or leader re-election can only be successful if at least $n/2+1$ query nodes are available, where n is the number of known query nodes.
- For a cluster with 2 query nodes, reinitialization or re-election can only be successful, if both nodes are available.

If leader re-election fails (not enough nodes available after the first half of `clusterReinitTimeout` seconds), another trial is triggered. Thus, leader re-election is always performed in an endless loop until it is successful.

Runtime Split-Brain Prevention

As explained above a leader will only establish when a majority of nodes registers with it after the election. This way only one leader can be established in this fashion. However it is possible that an established leader loses contact with a number of nodes such that it no longer has a majority of nodes registered with it. Subsequently a new leader could establish with a new majority, while the old leader is still running. To prevent this a running leader will automatically resign leadership when it no longer has contact to a majority of nodes (including itself).

Furthermore it has to be ensured that during a change of leadership no leader-relevant information such as distribution tables, sync states, etc. is lost. To ensure this the leader always chooses exactly $n/2$ nodes to be followers, such that this information always resides on a majority of nodes. Together with itself this means that this information resides on a majority of nodes, and thus the majority establishing the new leader will always contain a node with the newest version of this information which will then become leader.

Configuration of the Clusters

Activating Clustering

By default, clustering is always enabled for Cisco ParStream servers. You have to start each process (server and importer) with the same cluster ID (unique in your network), which is usually defined in an corresponding INI file:

```
clusterId = MyTestCluster
```

A server or importer can not connect with a server having a different ID. (Importers will wait forever when the cluster ID is wrong, expecting/hoping that, because no server is available at the moment, this will change soon.)

Next, you have to configure a registration port:

```
registrationPort = 9040
```

The leader will open this TCP port to accept connections from the other nodes.

There are a lot of additional options you can set, which are described in section 13.2.2, page 130.

Note that for very small clusters having only one or two query nodes, or if (almost) no redundancy is required, you have to set a couple of options so that operations are not (temporarily) blocked due to missing query nodes.

Update Redundancy

In case of clusters where the redundancy is higher than 1, you have to set the global option `minUpdateRedundancy` (see section 13.2.2, page 130). For example, to have all data on 2 nodes (one redundant node), you have to set:

```
minUpdateRedundancy = 2
```

The default minimum update redundancy is 1.

This option influences the merge and import behavior. If the affected distribution group has less nodes available than the number specified in `minUpdateRedundancy`, then

- merges are skipped and
- imports are blocked until enough nodes are available again (a retry of the import is scheduled after 1, 2, 4, 8, ... seconds).

Specify Servers

For each query node, add a server section to your INI files. The section must be named `[server.`*`servername`*`]` and must contain at least the node's rank, hostname, and port number.

For example:

```
[server.srv1]
rank = 30
port = 9010
host = castor.example.com
datadir = ./partitions

[server.srv2]
rank = 40
port = 9015
host = pollux.example.com
datadir = ./partitions
```

Make sure that each node has its own distinct rank, otherwise the leader election will fail.

Also note that servers sharing the same directory for their data results in undefined behavior. Thus, you have to ensure that each server has its own physical database directory. You can do that by

- running the servers on different hosts (as in the example above) or
- by specifying different directories as `datadir`.

Specify Importers

For each import node, in addition to general cluster settings you have to configure a section named `[import.`*`importername`*`]` as follows:

```
[import.imp1]
host = calypso.example.com
sourcedir = ./csv
targetdir = ./import
rank = 99
```

Note again that even import nodes need a unique rank, although they never can become a leader.

Note also that you don't have to start and/or register importers during the initial cluster setup or leader election. You can start importers (with or without a new rank) at any time the cluster is up and running.

Configuring the Leader Election Ports

For the leader election, Cisco ParStream technically establishes communication channels between all cluster nodes using TCP ports.

For this, all nodes in the cluster need a common unique registration port, which is used by each leader to manage the cluster. It should be set using the global option `registrationPort`. For example:

```
registrationPort = 9040
```

In addition, each node needs an individual "leaderElectionPort" for the leader election. For servers, `port+4` will be used (see section 13.3.1, page 135).

For importers, you have to set it as `leaderElectionPort`:

```
[import.imp1]
leaderElectionPort = 4712
```

Note the following when using TCP channels for leader election:

- In this mode a lot of initial communication channels are opened during the leader election so that each node can communication with other nodes.
- For this reason, **all processes have to know all ports of all servers in this mode**. That is, if you start your processes with passing the ports via commandline options, you have to specify all ports in all processes.
- Running two clusters with the same ports (but different cluster IDs) is not possible.

For example, when starting a cluster with servers `srv1`, `srv2`, and `srv3`, all servers have to know all ports of all other servers:

```
parstream-server srv1 --clusterId=MyClusterId
  --server.srv1.port=8855 --server.srv2.port=8860 --server.srv3.port=8865
  --registrationPort=9040

parstream-server srv2 --clusterId=MyClusterId
  --server.srv1.port=8855 --server.srv2.port=8860 --server.srv3.port=8865
  --registrationPort=9040

parstream-server srv3 --clusterId=MyClusterId
  --server.srv1.port=8855 --server.srv2.port=8860 --server.srv3.port=8865
  --registrationPort=9040
```

In addition, each importer has to know all ports of all servers and define a `leaderElectionPort`:

```
parstream-import imp1 --clusterId=MyClusterId
  --server.srv1.port=8855 --server.srv2.port=8860 --server.srv3.port=8865
  --registrationPort=9040 --leaderElectionPort=8890
```

A corresponding INI file shared by all nodes might contain the following rows:

```
clusterId = MyClusterId
registrationPort = 9040

[server.srv1]
port = 8855
...

[server.srv2]
port = 8860
...

[server.srv3]
port = 8865
...

[import.imp1]
leaderElectionPort = 8890
...
```

Configuring the Table Distribution

For each table, the distribution must be configured as described in Section “Partition Distribution” (see section 6.3, page 53).

State of a Cluster

Note that a Cisco ParStream database using a cluster with distributed data and failover support has a distributed state. In fact the state of the database as a whole consists out of:

- The **INI files** of each query node and each import node, containing basic configurations.
- The **partitions** distributed over and used by all query nodes, containing all imported data.
- The **journal files** of each query node and each import node, containing the (distributed) schema. They contain all table definitions and modifications, distribution lists, and data necessary to synchronize nodes that were temporarily offline.

  The files are located in a node specific sub-directory of the journal directory, which is specified by the `journaldir` option (see section 13.2.1, page 122). Thus, multiple nodes can share the same journal directory. The default value for the `journaldir` option is `./journals`. Journal sub-directories might have a prefix such as `import`.

To be able to query the state of a cluster, a couple of system tables are provided:

- **`ps_info_cluster_node`** (see section 26.4, page 317
  allows to list all nodes, their type, and whether they are online, active, leader, or follower (see section 6.2.1, page 41).
  For example:

```
SELECT name, host, port, type, leader, follower, active
       FROM ps_info_cluster_node
       ORDER BY leader DESC, follower DESC, type DESC;
```

  might have the following output:

```
#name;host;port;type;leader;follower;active
"srv1";"localhost";9110;"query";1;0;1
"srv2";"localhost";9120;"query";0;1;1
"srv3";"localhost";9130;"query";0;1;1
"srv4";"localhost";9140;"query";0;0;1
"srv5";"localhost";9150;"query";0;0;1
"imp1";"";0;"import";0;0;1
```

  Note that during a (re-)election the result might be something like (after establishing a connection, being online is enough to get these information):

```
#name;host;port;type;leader;follower;active
"srv1";"localhost";9110;"query";0;0;0
"srv2";"localhost";9120;"query";0;0;0
"srv3";"localhost";9130;"query";0;0;0
"srv4";"localhost";9140;"query";0;0;0
"srv5";"localhost";9150;"query";0;0;0
"imp1";"";0;"import";0;0;0
```

- **`ps_info_partition_sync_backlog`** (see section 26.4, page 318)
  allows to query from the leader information about open (re-)synchronizations.

For example:

```
SELECT node_name,type,table_name,relative_path FROM
    ps_info_partition_sync_backlog;
```

might have the following output:

```
#node_name;type;table_name;relative_path
"srv3";"import";"Hotels";"0/10865448114464928962Z_2013-09-09T17:45:34_imp1_PM"
"srv3";"import";"Hotels";"0/7295227430346185111Z_2013-09-09T17:45:34_imp1_PM"
"srv3";"import";"Hotels";"2/590598305913432875Z_2013-09-09T17:45:34_imp1_PM"
"srv3";"import";"Hotels";"2/6332577679151947647Z_2013-09-09T17:45:34_imp1_PM"
"srv3";"import";"Hotels";"2/921786917744982206Z_2013-09-09T17:45:34_imp1_PM"
"srv3";"import";"Hotels";"4/17149388985709726001Z_2013-09-09T17:45:34_imp1_PM"
```

signaling that node “`srv3`” has open synchronizations for an import of table “Hotels” with the listed physical partitions.
An empty result signals no open synchronizations.
Note, that you have to send this request **to the leader**. Otherwise, the result will be empty although open synchronizations exist or not up-to-date.[1]

- **`ps_info_partition_distribution`** (see section 26.4, page 323)
  allows to query from the leader (or a follower) information about which value gets distributed to which node.

Adding Cluster Nodes

You can add new query nodes while the cluster is running. This does not lead to a redistribution of existing data. Instead, *newly* imported data will be distributed to this node, provided new values get imported and dynamic partition distribution is used.

To add/register a new node, you have to start the new node as usual with the command `parstream-server`, but with the option `--registerNode` For example, if you want to add `srv4` as new cluster node start this server with the following command:

```
# start server srv4, registering it as new cluster node
parstream-server --registerNode srv4
```

Of course, as usual for the new server/node the corresponding INI file settings have to be provided.

Note that **this call is a pure registration**. After the registration, the process will immediately terminate. If the registration was successful, the exit status of the registration is 0 (`EXIT_SUCCESS`). You can find further information in the log output, such as (the detailed wording might change):

```
[2013-11-04T13:56:40]:srv4-135640:INFO(77063) Registration of cluster node
    'srv4' done
[2013-11-04T13:56:40]:srv4-135640:PROT(77018) node 'srv4' registered in
    cluster 'cluster_xyz', terminate now
```

[1]You can also send the request to other nodes. However, followers might return a state that is not up-to-date.

Note that the new node gets only registered, not started. To use the server in the cluster, you have to start it again this time as usual (without `--registerNode`) to let it become active and online:

```
# start server for node srv4 after registration
parstream-server srv4
```

Note the following limitations:

- Adding a node to a cluster that has a table with `EVERYWHERE` distribution (see section 6.3.1, page 58) is currently not supported.

Stopping, Restarting, and Upgrading a Cluster

Cluster Control Commands

The following subsection describes how to maintain a cluster. For this purpose cluster control commands are used which are described in section 16.4.1, page 200.

Stopping and Restarting a Cluster

To stop and restart a running cluster, it is necessary to ensure that no inconsistencies are created or unnecessary cluster reorganizations happen (for example by stopping the leader before other nodes, so that a leader re-election is initiated).
The usual way to shut down a cluster is the high-level cluster shutdown command

```
  ALTER SYSTEM CLUSTER SHUTDOWN;
```

This is roughly equivalent to the following manual sequence:

1. Find out, which nodes are leader and followers:

   ```
   -- find out leader:
   SELECT name, host, port FROM ps_info_cluster_node
          WHERE leader = 1;
   -- find out followers:
   SELECT name, host, port FROM ps_info_cluster_node
          WHERE follower = 1;
   ```

2. Stop import(er)s.

3. Stop query nodes that are neither leaders nor followers.

4. Stop query nodes that are followers.

5. Stop the leader node.

To restart the cluster, you should use the opposite order:

1. Restart the former leader node.

2. Restart the query nodes that have been followers.
3. Restart all remaining query nodes.
4. Restart importers.

Checking for Open Synchronizations

In order to check for open synchronizations, you should send

```
SELECT * from ps_info_partition_sync_backlog;
```

to the leader. As long as there are entries in the table, there are open synchronizations left to be processed. If you want to make sure that there are no more open synchronizations, you should deactivate imports and merges and wait until `ps_info_partition_sync_backlog` has no more entries.

Backup a Cluster

As explained in section 6.2.4, page 49, you should always backup all the INI files, the partitions, and the journal files if you want to save the state of the cluster. In addition, we recommend to stop the cluster at least partially so that no merges are running (see section 16.4.1, page 201), no imports are running (see section 16.4.1, page 200), and all open synchronizations are done (see section 6.2.6, page 52),

```
ALTER SYSTEM CLUSTER DISABLE MERGE;
ALTER SYSTEM CLUSTER DISABLE IMPORT;

-- should return zero before backup of the node is started
SELECT COUNT(*) FROM ps_info_partition_sync_backlog GROUP BY node_name;
```

Upgrading a Cluster

Sometimes it is necessary to upgrade a Cisco ParStream cluster to a new version. For performing a version upgrade in a cluster environment, please perform the following steps.

- Stop the cluster (see section 6.2.6, page 51).
- Install the new Cisco ParStream package.
- Make any necessary configuration changes (Check the incompatibilities section in the Release notes).
- Restart the cluster (see section 6.2.6, page 51).

Limitations for Clustering

Note the following limitations for the clustering of Cisco ParStream databases:

- Currently, the set of cluster nodes should usually be defined at the first startup. While you can add nodes later on, you can't dynamically remove nodes (nodes can only be down, which means that they are still considered for new data, distribution values, and merges).
- The relocation of cluster nodes to a different host requires manual activity i.e. to adjust the configured host names. You especially have to transfer the state (partitions and journal files)
- To re-setup a cluster from scratch you have to remove both partition and journal files/directories.
- The partition directories of different servers can't be shared. Thus, you have to ensure that the partition directories (option datadir) are on different hosts or have different names.

Partition Distribution

Cisco ParStream provides load balancing in form of partition distribution.

It means that the data gets partitioned across multiple servers/nodes so that each server/node is responsible for a different set of data. A query sent to one server, say `srv1`, will try to distribute the request over the servers assigned as primary (or fallback) query nodes according to the current distribution table. The other servers send their partial results to `srv1`, where it is consolidated into the final result, which is send back to the client.

In Cisco ParStream, the values are distributed according to the current **distribution table**, that each node internally has. Each table has exactly one column for which it manages a list of query nodes, which defines where the data is stored for each possible value for that column. All lists for all tables compose the **distribution table**. For any query sent to a query node, the query node hands the request over to the primary or (if not available) to the fallback nodes, which are listed for the value in the distribution table.

The distribution table is a dynamic table that grows with every new value not already registered, appending its distribution (primary and fallback nodes) determined by a shifting round robin algorithm.

In addition, note the following:

- The distribution criterion has to be one of the table's partitioning columns.
- Currently, only integral and date/time column types are allowed as distribution values. See section 6.3.1, page 59 for details about how to distribute over (non-hashed) strings and other types.
- It is not possible to configure a function partition value as distribution criterion. However you can use an ETL column instead.
- It is possible to configure more than one server for a given value of the distribution criterion. The first configured server is the primary server for this value. All other servers are backup servers for this partition. If the primary server is down, one of the backup servers will be used for this partition. The querying client will not take notice of this fallback situation. If the query node the client connects to cannot find a query node for a partition (i.e. neither the primary node the backup nodes are available), the request cannot be fulfilled and an error is send to the client.

The following subsections explain these features in detail with typical examples. See chapter 24, page 277 for details of the exact syntax.

Dynamic Partition Distribution

You can distribute your partitions dynamically. That means, that you don't have to specify for each possible value the nodes where it gets distributed. Instead, new values, for which the distribution is not clear yet, are distributed according to a certain policy (which is described below).

To be able to use dynamic partition distribution when switching from older Cisco ParStream versions you can define an initial static distribution for a limited set of values.

You can specify a `DISTRIBUTION` clause in one of the following formats:

- **`DISTRIBUTE OVER`** ***`col`***
  distribution according to column *col* with 1 fallback node
- **`DISTRIBUTE OVER`** ***`col`*** **`WITH REDUNDANCY`** ***`num`***
  distributed according to column *col* with *num*-1 fallback nodes
- **`DISTRIBUTE OVER`** ***`col`*** **`WITH INITIAL DISTRIBUTION`** ***`dlist`***
  distributed according to column *col* with 1 fallback node and an initial static distribution *dlist*
- **`DISTRIBUTE OVER`** ***`col`*** **`WITH REDUNDANCY`** ***`num`*** **`WITH INITIAL DISTRIBUTION`** ***`dlist`***
  distributed according to column *col* with *num*-1 fallback nodes and an initial static distribution *dlist*
- **`DISTRIBUTE OVER`** ***`col`*** **`BY COLOCATION WITH`** ***`tab`***
  distributed according to the distribution that the value of column *col* would have in table *tab*
- **`DISTRIBUTE EVERYWHERE`**
  distributed over all query nodes

If the `REDUNDANCY` is not specified, the default value 2 is used, so that we have 1 fallback node. The maximum redundancy allowed is 9.

In addition you can explicitly specify the default policy for new values, `BY ROUND_ROBIN`, which isn't necessary yet because this is the only policy yet.

Note the following restrictions:

- If old imported data violates any internal distribution rules, you run into undefined behavior. Note that you can query the existing distribution policy (see section 6.3.1, page 59).

The clause will be explained in details in the following subsections. See section 27.7.2, page 365 for the grammar specified in Backus-Naur-Form (BNF).

Note that you can use this approach to benefit from the Separation Aware Execution optimizations (see section 15.15, page 186 for details).

Simple Dynamic Distribution

The simple example for a dynamic partition distribution is just to specify the column, which is used as distribution criterion.

For example:

```
CREATE TABLE MyTable (
   zip INT16 ... INDEX EQUAL ...
   ...
)
PARTITION BY zip, ...
```

```
DISTRIBUTE OVER zip
...
```

Note the following:

- The distribution column (specified with `DISTRIBUTE OVER`) has to be a partition column (specified in the column list for `PARTITION BY`) that has to have an `EQUAL` index.
- Currently, only integral and date/time types are supported. See section 6.3.1, page 59 for details about how to distribute over strings and date/time types.

Note that for performance reasons the column to distribute over should have a small/limited number of possible values. For this reason, you might use an ETL column as distribution column. For example:

```
CREATE TABLE MyTable (
   userid UINT32,
   ...
   usergroup UINT64 INDEX EQUAL CSV_COLUMN ETL
)
PARTITION BY usergroup, zip
DISTRIBUTE OVER usergroup
...
ETL ( SELECT userid MOD 20 AS usergroup
             FROM CSVFETCH(MyTable)
);
```

By default, the distribution uses a default redundancy of 2, which means that 1 fallback node is required. You can change this default by specifying a different `REDUNDANCY`:

```
CREATE TABLE MyTable (
   zip INT16 ... INDEX EQUAL ...
   ...
)
PARTITION BY zip, ...
DISTRIBUTE OVER zip WITH REDUNDANCY 3
...
```

Note that the value specified as `REDUNDANCY` has to be possible. If you specify a `REDUNDANCY` of `3`, an import requires 3 query nodes. Thus, the redundancy should not exceed the number of query nodes in the cluster. In fact, if you have a cluster with only 1 query node, **you have** to set the `REDUNDANCY` to 1.

Note also that the importer blocks until at least one of the assigned distribution nodes for a value encountered during a CSV import is available.

Details of the Distribution Policy

To provide distributions for new values, Cisco ParStream uses an approach that should provide failover and load balancing without much internal effort. The principal algorithm is as follows:

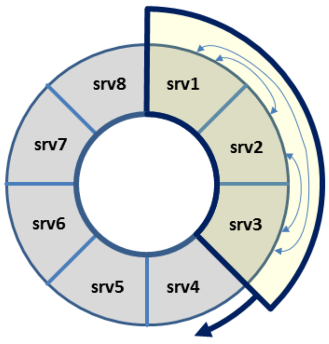
- If the redundancy covers all existing query nodes all permutations of distribution lists are used. For example, if we have a redundancy of 3 with 3 servers, each new distribution value gets the next entry of the following list:

  ```
  srv1 srv2 srv3
  srv2 srv3 srv1
  srv3 srv1 srv2
  srv1 srv3 srv2
  srv3 srv2 srv1
  srv2 srv1 srv3
  ```

- If the redundancy is smaller than the current number of nodes, we iterate a window of *numberOfRedundant* servers over the list of all servers and use all permutations in that window. That is, if we have 8 servers and a redundancy of 3, then we iterate with the following window states of 3 servers over all 8 servers:

  ```
  srv1 srv2 srv3
  srv2 srv3 srv4
  srv3 srv4 srv5
  srv4 srv5 srv6
  srv5 srv6 srv7
  srv6 srv7 srv8
  srv7 srv8 srv1
  srv8 srv1 srv2
  ```

  and use all 6 permutations in each window (as demonstrated by the following figure):

  That is, for a specific number of servers, we have
  numberOfServers ***** *redundancy***!**
  different distribution list entries (here: `8 * 3*2*1`, thus `48`).

Note that the possible distribution lists for new values change, when the number of nodes change. For example, the window `[ srv1 srv2 srv4 ]` is possible with 4 nodes, but not with 5 or more nodes. Thus, with 4 nodes new distributions might use this window, while after adding an additional node this window won't be used again for new distribution values. As a consequence, the number of different distribution lists can be higher with the number of nodes over time raised to 8 than with a stable number of 8 nodes from the beginning.

INITIAL Distributions

You can specify an initial static distribution for a limited set of values. This feature is especially provided to be able to deal with data imported in older versions where only a static distribution was possible. In addition, this allows to re-initialize the whole database with dynamic distribution with partitions that were already distributed (to query the existing distribution, see section 6.3.1, page 59).

To specify an initial static distribution, you have to use an `INITIAL DISTRIBUTION` clause. In that clause you can specify the query nodes, data should be distributed to, for each value.

For example:

```
CREATE TABLE MyTable (
   zip INT16 ... INDEX EQUAL ...
   ...
)
PARTITION BY zip, ...
DISTRIBUTE OVER zip WITH INITIAL DISTRIBUTION (
                                    (0  TO srv1,srv2,srv3),
                                    (1  TO srv2,srv3,srv1),
                                    (2  TO srv3,srv1,srv2),
                                    (3  TO srv1,srv3,srv2),
                                    (4  TO srv2,srv1,srv3),
                                    (5  TO srv3,srv2,srv1),
                                  )
```

Note that in the lists of servers you should always use **all permutations** to avoid bad load balance situations when a server is not available. This ensures that during a failure of one server, the requests are equally distributed among all other servers. For example, if with the configuration above `srv1` fails, the requests are distributed to `srv2` for value `0` and to `srv3` for value `3`. A setup such as:

```
...
DISTRIBUTE OVER zip WITH INITIAL DISTRIBUTION (
                                    -- bad distribution !
                                    (0  TO srv1,srv2,srv3),
                                    (1  TO srv2,srv3,srv1),
                                    (2  TO srv3,srv1,srv2),
                                    (3  TO srv1,srv2,srv3),
                                    (4  TO srv2,srv3,srv1),
                                    (5  TO srv3,srv1,srv2),
                                  )
```

would be a lot worse because with `srv1` not being available all requests are distributed to `srv2` only.

You can assign the same nodes for multiple values (but again keep in mind to specify all permutations):

```
CREATE TABLE MyTable (
   zip INT16 ... INDEX EQUAL ...
   ...
)
```

```
PARTITION BY zip, ...
DISTRIBUTE OVER zip WITH INITIAL DISTRIBUTION (
                                    (0,6,12,18 TO srv1,srv2,srv3),
                                    (1,7,13,19 TO srv2,srv3,srv1),
                                    (2,8,14,   TO srv3,srv1,srv2),
                                    (3,9,15    TO srv1,srv2,srv3),
                                    (4,10,16   TO srv2,srv3,srv1),
                                    (5,11,17   TO srv3,srv1,srv2),
                                  )
```

Note that the initial distribution is independent from the specified or default redundancy. That is, the previous statement would lead to a redundancy of 3 for each value from 1 to 19 and to a redundancy of 2 for all new values.

To specify a redundancy of 3 for all values, you have to specify:

```
CREATE TABLE MyTable (
   zip INT16 ... INDEX EQUAL ...
   ...
)
PARTITION BY zip, ...
DISTRIBUTE OVER zip WITH REDUNDANCY 3 WITH INITIAL DISTRIBUTION (
                                    (0,3,6,9,12,15,18  TO srv1,srv2,srv3),
                                    (1,4,7,10,13,16,19 TO srv2,srv3,srv1),
                                    (2,5,8,11,14,17    TO srv3,srv1,srv2),
                                  )
```

EVERYWHERE Distributions

You can specify that a distribution provides full redundancy. That is, all data is distributed over all nodes.

For example:

```
CREATE TABLE MyTable (
   zip INT16 ... INDEX EQUAL ...
   ...
)
PARTITION BY zip, ...
DISTRIBUTE EVERYWHERE
...
```

This feature will especially be useful when small tables are used in `JOIN`s of distributed tables. By replicating the data to all query nodes, these small tables are always locally available when queries for the distributed table are processed. This leads to a better performance because the whole query is processed inside one node.

Note that adding a node to a cluster (see section 6.2.5, page 50) that has a table with `EVERYWHERE` distribution is currently not supported.

COLOCATION Distributions

You can also specify that a distribution according to one column in one table has to follow the distribution of another column in another table. This is useful for JOINs where it is an performance improvement that typically the data of the joined tables is co-located on the same query node.

For example:

```
CREATE TABLE MyTable1 (
   no INT16 ... INDEX EQUAL ...
   ...
)
PARTITION BY no, ...
DISTRIBUTE OVER no
...

CREATE TABLE MyTable2 (
   zip INT16 ... INDEX EQUAL ...
   ...
)
PARTITION BY zip, ...
DISTRIBUTE OVER zip BY COLOCATION WITH MyTable1
...
```

Here, the values of MyTable2 are distributed according to the distribution the value of zip would have as distribution value of table MyTable1. That is, the zip code 34500 in MyTable2 has the same distribution as the value 34500 as no would have in MyTable1.

Note:

- The distribution columns of both tables have to have the same type.
- The basic table referred to with COLOCATION WITH has to be a table that has a directly specified round robin policy specified with DISTRIBUTE OVER. That is, the table referred to is not allowed to have a DISTRIBUTE EVERYWHERE or DISTRIBUTE ... BY COLOCATION distribution policy. However, a basic table might have multiple tables that co-locate their distribution with it.

Querying the Current Distribution

You can query the current distribution via the system tables ps_info_table (for the distribution configuration) and ps_info_partition_distribution (for the distribution of the imported values). See section 26.4, page 323 for details.

Distribution over String and Date/Time Values

There are restrictions regarding the type of the distributed column: You need an integral or date/time type. However, indirectly you can still use other types for distribution. Note also that for hashed string you have to pass NULL instead of the empty string as distribution value.

To be able to use other types, the general approach is to provide an ETL column (see section 10.6, page 104), which is filled by a function transforming another import column into an integral value. If you need a limited value range, you can also use `MOD`. Note however, that you can't MOD for non-numeric types (except `DATE` and `SHORTDATE`).

Thus, you have the following options:

- For **non-hashed strings** (type `VARSTRING` without compression `HASH64`) you should provide a `ETL` statement calling `HASH64()` (see section 25, page 303) for the string value.
  For example:

  ```
  CREATE TABLE Hotels
  (
    City VARSTRING,
    ...
    distr UINT64 INDEX EQUAL CSV_COLUMN ETL,
  )
  PARTITION BY distr, Hotel
  DISTRIBUTE OVER distr
  ETL (
    SELECT HASH64(City) AS distr
           FROM CSVFETCH(Hotels)
  );
  ```

  If you need a limited value range, you have to call `MOD` on the value returned by `HASH64()`.
  For example:

  ```
  CREATE TABLE Hotels
  (
    City VARSTRING,
    ...
    distr UINT64 INDEX EQUAL CSV_COLUMN ETL,
  )
  PARTITION BY distr, Hotel
  DISTRIBUTE OVER distr
  ETL (
    SELECT HASH64(City) MOD 7 AS distr
           FROM CSVFETCH(Hotels)
  );
  ```

- For **floating-point types**, you can call `TRUNC()` or `FLOOR()` (and `MOD`).
- For **hashed strings** (type `VARSTRING` with compression `HASH64`) you also can use the `HASH64()` function, which yields the hash value of the string (see section 25, page 303). But if you need a limited value, you can call `MOD` on the hashed string directly.

Dynamic Columns

This section describes the usage of the Cisco ParStream database feature for dealing with dynamic columns.

Corresponding example code can be found in `examples/dynamiccolumns` (see section A.3, page 392).

Motivation for Dynamic Columns

One problem of databases running against a fixed schema is that adding a new column is a significant change. When tracking data and states, what or how something is being measured might change over time. For example, sensors might be added, enabled, modified, disabled, or removed. Having a static column for each device or point of measurement would lead to a lot of schema changes.

A better approach to have a specific column for each specific point of measurement is to collect data in a generic way, having a key identifying what or how something was measured and a corresponding value. However, the SQL expressions to analyze data for such a generic approach become very complex, using JOINs and other features.

For this reason, Cisco ParStream provides an approach to capture/import data in a raw generic format, while allowing to map raw data to **Dynamic Columns** when analyzing/processing data.

For example, you can measure raw data over time as follows:

Timestamp	Region	ID	Attribute	val_uint8	val_int64	val_double
2015-07-07 12:00:00	1	us16x	power	77		
2015-07-07 12:00:00	1	us771	humidity	67		
2015-07-07 12:00:00	1	us771	pressure			1234.3
2015-07-07 12:01:02	2	ca224	rpm		86500	
2015-07-07 12:02:22	1	us771	humidity	64		
2015-07-07 12:02:22	1	us771	pressure			1237.1
2015-07-07 12:02:22	1	us771	rpm		86508	
2015-07-07 12:02:22	2	ca224	rpm		86433	
2015-07-07 12:03:44	1	us771	pressure			1240.7
2015-07-07 12:03:44	2	ca224	rpm		86622	
2015-07-07 12:04:00	1	us16x	power	99		
2015-07-07 12:04:00	1	us990	humidity	63		
2015-07-07 12:04:00	1	us16y	power	98		

Here, for certain points in time, we then can specify

- the `Region` code and `ID` of a sensor, representing *where*, *how*, and/or *by what* a value was measured,
- the `Attribute` that was measured, and
- the value the attribute had when it was measured, using different value columns for different data types.

Using the dynamic columns feature, you can deal with such a raw table as if it would have the different attributes imported as individual columns:

Timestamp	Region	ID	Humidity	Power	Pressure	RPM
2015-07-07 12:00:00.000	1	us16x		77		
2015-07-07 12:00:00.000	1	us771	67		1234.3	
2015-07-07 12:01:02.000	2	ca224				86500
2015-07-07 12:02:22.000	1	us771	64		1237.1	86508
2015-07-07 12:02:22.000	2	ca224				86433
2015-07-07 12:03:44.000	1	us771			1240.7	
2015-07-07 12:03:44.000	2	ca224				86622
2015-07-07 12:04:00.000	1	us16x		99		
2015-07-07 12:04:00.000	1	us16y		98		
2015-07-07 12:04:00.000	1	us990	63			

The resulting "*dynamic table*" has a column for each attribute with the corresponding type that was used to store the values. Rows with values for different attributes of the same timepoint and sensor are combined.

Similarly, if you just care for the values of a specific region, you can get:

Timestamp	Region	Humidity	Power	Pressure	RPM
2015-07-07 12:00:00.000	1	67	77	1234.3	
2015-07-07 12:01:02.000	2				86500
2015-07-07 12:02:22.000	1	64		1237.1	86508
2015-07-07 12:02:22.000	2				86433
2015-07-07 12:03:44.000	1			1240.7	
2015-07-07 12:03:44.000	2				86622
2015-07-07 12:04:00.000	1	63	99		
2015-07-07 12:04:00.000	1	63	98		

As you can see, the first two rows were joined because they have the same region and distinct attributes. In fact, because two different sensors in the same region did yield different values at the same time, we have one joined row in the dynamic table for region "1" of 12:00:00. In addition, the last three rows were combined into two rows because we join both values for power with the value for humidity.

These dynamic tables are created on the fly as temporary tables and can be used as a table in arbitrary queries, which makes handling of this generic approach a lot easier.

Using Dynamic Columns

To be able to use the dynamic columns approach, you have to

- Define a generic **raw table** that can be used with dynamic columns.
- Use the `DYNAMIC_COLUMNS` operator to use the corresponding "**dynamic table**," temporarily generated from the source table and its data.

Defining Generic Raw Tables for the Dynamic Columns Approach

To define a generic table that can be used for the dynamic columns feature, the table has to have the following special constraints:

- There must be exactly one column marked with `DYNAMIC_COLUMNS_KEY`, defining the **dynamic columns key**. This column stores the name that is used as column name when using the table as dynamic table. The column type must be a hashed string.
- There must be one or more columns marked with `DYNAMIC_COLUMNS_VALUE`, that can be used for **dynamic columns values** associated with the dynamic columns key. Here, multiple value columns are possible to be able to handle different value types, index configurations. etc. Thus, for each possible value type there should be a corresponding value column. However, multiple keys can share the same value column, if the value type is the same.

In addition, the following constraints apply:

- The table must have an `ORDER BY` clause, which specifies at least one physical column, which is neither a dynamic columns key nor a dynamic columns value. Cisco ParStream uses the columns listed there as `JOIN_COLUMNS` in a dynamic table to be able to decide how to construct the logical rows inside a dynamic columns operator (without such join columns, Cisco ParStream would always combine all rows with all other rows, which would not be useful).

  The first ORDER BY column should be *THE* most coarse-grained attribute you want to analyze over, because this column always has to be used in the `JOIN_COLUMNS` of a dynamic table using this raw table. In most cases, it will be a timestamp attribute if you aggregate data over time (as shown in the initial example on page 62 in Section 7.2).

Here is a first example, specifying a table for the motivating example above (with slightly more technical names such as `ts` instead of `Timestamp`):

```
CREATE TABLE RawData
(
  ts TIMESTAMP NOT NULL INDEX EQUAL,
  region UINT64 NOT NULL INDEX EQUAL,
  id VARSTRING(255) NOT NULL COMPRESSION HASH64 SEPARATED BY region INDEX
    EQUAL,
  attribute VARSTRING(255) NOT NULL COMPRESSION HASH64 INDEX EQUAL
    DYNAMIC_COLUMNS_KEY,
  val_uint8 UINT8 INDEX EQUAL DYNAMIC_COLUMNS_VALUE,
  val_int64 INT64 INDEX EQUAL DYNAMIC_COLUMNS_VALUE,
  val_double DOUBLE INDEX EQUAL DYNAMIC_COLUMNS_VALUE,
)
PARTITION BY region, id
DISTRIBUTE OVER region
ORDER BY ts, id
ETL (SELECT LOWER(attribute) AS attribute FROM CSVFETCH(RawData))
```

Here, over time (defined in `ts`), we collect and analyze different data, identified by the name in the key column `attribute`. The data is provided by a sensor identified with `id`. These sensors are located in different regions, defined by integral values, which are used to distribute the data over different servers. For the values of the dynamic columns, we provide three possible types: one for small unsigned integral values, one for large signed integral values, and one for floating-point values.

Note the following:

- Any source table for the dynamic columns feature is still an ordinary table that can be used directly as if the `DYNAMIC_COLUMNS_KEY` and `DYNAMIC_COLUMNS_VALUE` attributes would not have been set.
- There is only one `DYNAMIC_COLUMNS_KEY` column allowed per table.
- The `DYNAMIC_COLUMNS_KEY` column must be defined as `VARSTRING` with `COMPRESSION HASH64 INDEX EQUAL`. In addition, the following constraints for key columns apply:
  - The column may not be a primary key.
  - The column may not be defined as `UNIQUE` column.
  - The column may not be defined as `SKIP`ed column.
- We strongly suggest that the `DYNAMIC_COLUMNS_KEY` column has a `NOT NULL` constraint to ensure that no rows without any key may get imported.
- We also suggest that the columns to partition or join over are also `NOT NULL`, because joining with NULL values sometimes results into surprising results.
- The values provided for the dynamic columns keys should be valid column names and differ not only regarding case-sensitivity. Ideally, they should follow a common convention such as using uppercase or lowercase letters only (see section 7.2.2, page 68 for details). Otherwise, we have a mixture of case-sensitive values that define different columns although column names are normally case-insensitive. This results into undefined behavior when using dynamic columns queries (see below).

  To ensure that no different spelling of the same attribute causes this undefined behavior, we strongly recommend to use an ETL statement that converts all attributes to lowercase or uppercase letters as it is done in this example:

  ```
  ETL (SELECT LOWER(attribute) AS attribute FROM CSVFETCH(RawData))
  ```

  (See section 10.6, page 105 for details of modifying ETL statements).
- A `DYNAMIC_COLUMNS_VALUE` column must not have a default value other than NULL. For this reason, it also must not have a `NOT NULL` constraint.
- Neither a `DYNAMIC_COLUMNS_KEY` nor a `DYNAMIC_COLUMNS_VALUE` column may be used as partitioning columns (used in the `PARTITION BY` clause of the raw table) or as first field in the `ORDER BY` clause of the raw table.

Using Dynamic Tables

To be able to use tables supporting the dynamic columns approach as dynamic tables (in the mode where the generic key entries are handled as columns), you have to use the `DYNAMIC_COLUMNS` operator. It yields a dynamic table as temporary table, which can be used like any other table. However, when using the `DYNAMIC_COLUMNS` operator you have to specify some attributes and some restrictions apply.

For example, with the following SQL expression, you can convert the raw table defined above to a dynamic table having all values assigned to a specific timestamp:

```
DYNAMIC_COLUMNS(ON RawData PARTITION BY id JOIN_COLUMNS(ts))
```

Here, we create a dynamic table based on the data in `RawData`, which is partitioned by column `id`, and join the data in each partition over column `ts`. Because the raw table has the distribution column `region`, the `PARTITION BY` clause is necessary (using the distributed column or a column separated by it). The `JOIN_COLUMNS` clause has to have at least the first column in the `ORDER BY` clause of the raw table. The resulting table will have all columns used in the `PARTITION BY` and the `JOIN_COLUMNS` clauses plus the columns derived from the dynamic column key values.

As a result, a query such as

```
SELECT * FROM DYNAMIC_COLUMNS(ON RawData PARTITION BY id JOIN_COLUMNS(ts))
         ORDER BY ts
```

will generate a table having a column for each attribute imported as dynamic columns keys. In this table, for each point in time, the rows will have the corresponding values (or NULL).

For example (according to the motivating example above, see section 7.1, page 61), if you import the following data (e.g., via CSV import):

```
# ts; region; id; attribute; values (different types)
2015-07-07 12:00:00; 01; us16x; power    ;77;     ;
2015-07-07 12:00:00; 01; us771; humidity ;67;     ;
2015-07-07 12:00:00; 01; us771; pressure ;  ;     ;1234.3
2015-07-07 12:01:02; 02; ca224; rpm      ;  ;86500;
2015-07-07 12:02:22; 01; us771; humidity ;64;     ;
2015-07-07 12:02:22; 01; us771; pressure ;  ;     ;1237.1
2015-07-07 12:02:22; 01; us771; rpm      ;  ;86508;
2015-07-07 12:02:22; 02; ca224; rpm      ;  ;86433;
2015-07-07 12:03:44; 01; us771; pressure ;  ;     ;1240.7
2015-07-07 12:03:44; 02; ca224; rpm      ;  ;86622;
2015-07-07 12:04:00; 01; us16x; power    ;99;     ;
2015-07-07 12:04:00; 01; us990; humidity ;63;     ;
2015-07-07 12:04:00; 01; us16y; power    ;98;     ;
```

the query

```
SELECT * FROM DYNAMIC_COLUMNS(ON RawData PARTITION BY id JOIN_COLUMNS(ts))
         ORDER BY ts, id
```

will yield the following data (spaces added to make it more readable):

```
#ts;id;humidity;power;pressure;rpm
2015-07-07 12:00:00.000; "us16x"; <NULL>; 77;     <NULL>; <NULL>
2015-07-07 12:00:00.000; "us771"; 67;     <NULL>; 1234.3; <NULL>
2015-07-07 12:01:02.000; "ca224"; <NULL>; <NULL>; <NULL>; 86500
2015-07-07 12:02:22.000; "ca224"; <NULL>; <NULL>; <NULL>; 86433
2015-07-07 12:02:22.000; "us771"; 64;     <NULL>; 1237.1; 86508
```

```
2015-07-07 12:03:44.000; "ca224"; <NULL>; <NULL>; <NULL>; 86622
2015-07-07 12:03:44.000; "us771"; <NULL>; <NULL>; 1240.7; <NULL>
2015-07-07 12:04:00.000; "us16x"; <NULL>; 99;     <NULL>; <NULL>
2015-07-07 12:04:00.000; "us16y"; <NULL>; 98;     <NULL>; <NULL>
2015-07-07 12:04:00.000; "us990"; 63;     <NULL>; <NULL>; <NULL>
```

Because data with the attributes `humidity`, `pressure`, `rpm`, and `power` was inserted, the resulting dynamic table has these columns. If we add additional data afterwards with a new attribute, the resulting dynamic table would automatically have one more column, if the same query is processed again.

The rows contain all values we have for a particular sensor for a particular point in time, because we join explicitly over `ts`. However, different sensors always have different rows, because we partition over the sensors (column `id`). Thus, for each attribute there is only a column with multiple entries if multiple rows have values for different attributes of the same ID.

To see and join over both ID and region, you have to request the following:

```
SELECT ts, id, region, *
       FROM DYNAMIC_COLUMNS(ON RawData
                            PARTITION BY id
                            JOIN_COLUMNS(ts, region))
       ORDER BY ts, id
```

or:

```
SELECT ts, id, region, *
       FROM DYNAMIC_COLUMNS(ON RawData
                            PARTITION BY id, region
                            JOIN_COLUMNS(ts))
       ORDER BY ts, id
```

which will both yield the following data:

```
#ts;id;region;humidity;power;pressure;rpm
2015-07-07 12:00:00.000; "us16x"; 1; <NULL>; 77;     <NULL>; <NULL>
2015-07-07 12:00:00.000; "us771"; 1; 67;     <NULL>; 1234.3; <NULL>
2015-07-07 12:01:02.000; "ca224"; 2; <NULL>; <NULL>; <NULL>; 86500
2015-07-07 12:02:22.000; "ca224"; 2; <NULL>; <NULL>; <NULL>; 86433
2015-07-07 12:02:22.000; "us771"; 1; 64;     <NULL>; 1237.1; 86508
2015-07-07 12:03:44.000; "ca224"; 2; <NULL>; <NULL>; <NULL>; 86622
2015-07-07 12:03:44.000; "us771"; 1; <NULL>; <NULL>; 1240.7; <NULL>
2015-07-07 12:04:00.000; "us16x"; 1; <NULL>; 99;     <NULL>; <NULL>
2015-07-07 12:04:00.000; "us16y"; 1; <NULL>; 98;     <NULL>; <NULL>
2015-07-07 12:04:00.000; "us990"; 1; 63;     <NULL>; <NULL>; <NULL>
```

Note that you can specify `region` both as additional partition column or as additional join column.[1]

[1] There is small difference between using `region` as partitioning or join column, though: For `PARTITIONING BY` we apply the rules for `GROUP BY`, while for `JOIN_COLUMNS` we apply the rules for `JOIN`. This has different effects if input rows contain NULL as values for these partitioning or join columns, which can become pretty confusing. For this reason, we recommend to declare partitioning and joining columns to be `NOT NULL`, so that both forms have the same effect.

To partition over regions only, the query should look like the following:

```
SELECT ts, region, * FROM DYNAMIC_COLUMNS(ON RawData PARTITION BY region
    JOIN_COLUMNS(ts)) ORDER BY ts, region
```

It joins the data of rows that have the same timepoint and the same region. As a result, we now also join the first two rows and combine the last three rows into two rows:

```
#ts;region;humidity;power;pressure;rpm
2015-07-07 12:00:00.000; 1; 67;     77;     1234.3; <NULL>
2015-07-07 12:01:02.000; 2; <NULL>; <NULL>; <NULL>; 86500
2015-07-07 12:02:22.000; 1; 64;     <NULL>; 1237.1; 86508
2015-07-07 12:02:22.000; 2; <NULL>; <NULL>; <NULL>; 86433
2015-07-07 12:03:44.000; 1; <NULL>; <NULL>; 1240.7; <NULL>
2015-07-07 12:03:44.000; 2; <NULL>; <NULL>; <NULL>; 86622
2015-07-07 12:04:00.000; 1; 63;     98;     <NULL>; <NULL>
2015-07-07 12:04:00.000; 1; 63;     99;     <NULL>; <NULL>
```

Note that because two different sensors in the same region did yield a value for the *same* attribute `power` at the same time 12:04:00, we now have two joined rows in the dynamic table combining these values with the humidity value for the same timepoint and region.

To get same column names as the motivating example (see section 7.1, page 62), we can use aliases for both the regular static columns and the dynamic columns in the dynamic table as follows:

```
SELECT "Timestamp", Region, *
FROM DYNAMIC_COLUMNS(ON RawData
                     PARTITION BY region
                     JOIN_COLUMNS(ts)
                     DYNAMIC_COLUMN_NAMES(humidity AS Humidity,
                                          "power" AS "Power",
                                          pressure AS Pressure,
                                          rpm AS RPM)
                     STATIC_COLUMN_NAMES(ts AS "Timestamp",
                                         region as Region)
) ORDER BY "Timestamp", Region
```

Note that as keywords such as `power` and `timestamp` have to get quoted here.

The output would be then as follows (spaces added to make it more readable):

```
#Timestamp;Region;Humidity;Power;Pressure;RPM
2015-07-07 12:00:00.000; 1; 67;     77;     1234.3; <NULL>
2015-07-07 12:01:02.000; 2; <NULL>; <NULL>; <NULL>; 86500
2015-07-07 12:02:22.000; 1; 64;     <NULL>; 1237.1; 86508
2015-07-07 12:02:22.000; 2; <NULL>; <NULL>; <NULL>; 86433
2015-07-07 12:03:44.000; 1; <NULL>; <NULL>; 1240.7; <NULL>
2015-07-07 12:03:44.000; 2; <NULL>; <NULL>; <NULL>; 86622
2015-07-07 12:04:00.000; 1; 63;     98;     <NULL>; <NULL>
2015-07-07 12:04:00.000; 1; 63;     99;     <NULL>; <NULL>
```

Note that in dynamic tables it can happen that rows exist even if the columns selected have no valid entries. For example, if you only select for entries with the attribute `rpm`:

```
SELECT ts, rpm FROM DYNAMIC_COLUMNS(ON RawData
                                    PARTITION BY id
                                    JOIN_COLUMNS(ts))
               ORDER BY ts, rpm
```

the output will be as follows:

```
#ts;rpm
2015-07-07 12:00:00.000;<NULL>
2015-07-07 12:00:00.000;<NULL>
2015-07-07 12:01:02.000;86500
2015-07-07 12:02:22.000;86433
2015-07-07 12:02:22.000;86508
2015-07-07 12:03:44.000;<NULL>
2015-07-07 12:03:44.000;86622
2015-07-07 12:04:00.000;<NULL>
2015-07-07 12:04:00.000;<NULL>
2015-07-07 12:04:00.000;<NULL>
```

To skip the rows with NULL values, use a corresponding WHERE condition. For example:

```
SELECT ts, rpm FROM DYNAMIC_COLUMNS(ON RawData
                                    PARTITION BY id
                                    JOIN_COLUMNS(ts))
               WHERE rpm IS NOT NULL
               ORDER BY ts, rpm
```

Here, the output is as follows:

```
#ts;rpm
2015-07-07 12:01:02.000;86500
2015-07-07 12:02:22.000;86433
2015-07-07 12:02:22.000;86508
2015-07-07 12:03:44.000;86622
```

Note that such a `WHERE` clause is especially necessary if you only are requesting for columns that were added recently. For all data, before the new dynamic column was added, you would otherwise get rows with NULL values.

Dynamic Column Names

There are certain requirements and constraints regarding dynamic column names, because on one hand the dynamic columns key values can in principle be any string, while column names have some restrictions.

Note the following:

- If a dynamic columns key is not a valid column name (see section 24.2.1, page 281), it is not listed in dynamic tables.
- The dynamic columns are automatically sorted according to their case-sensitive name (using ASCII order)
- Because in general column names are case-insensitive in Cisco ParStream, it is **undefined behavior** if column names differ but are case-insensitively equal. (e.g. having the value/name 'Height', 'height', and 'HEIGHT'). It is not guaranteed whether a * yields multiple or just one column and what happens if one of the columns is explicitly requested.
- A static column name hides a dynamic column with the same name. Use aliasing to make them visible (see section 7.2.3, page 71).

NOTE: For this reason, it is a strong recommendation to **establish/force a general convention for values of dynamic columns keys to be valid column names**:

- The string values should have at least two letters and no special characters.
- The string values should follow a general convention regarding case-sensitivity, such as using only capital letters or using only lowercase letters.
- To ensure that no different spelling of the same attribute causes undefined behavior, we strongly recommend to use an ETL statement that converts all attributes to lowercase or uppercase letters. For example:

```
CREATE TABLE RawData
(
   ...
   attribute VARSTRING(255) NOT NULL COMPRESSION HASH64 INDEX EQUAL
     DYNAMIC_COLUMNS_KEY,
   ...
)
...
ETL (SELECT LOWER(attribute) AS attribute FROM CSVFETCH(RawData))
```

You can check, whether dynamic column keys are valid or conflict by using the system table `ps_info_dynamic_columns_mapping` (see below).

System Table Support

We provide a system table `ps_info_dynamic_columns_mapping` (See section 26.4, page 316), which lists the actual dynamic columns combined with some information about whether column names are invalid or conflict with each other.

A request such as

```
SELECT * FROM ps_info_dynamic_columns_mapping ORDER BY table_name,
    dynamic_name
```

after importing the data listed above, will correspondingly have the following output:

```
#table_name;dynamic_name;key_column;value_column;is_valid;is_conflicting
```

```
"RawData";"HEIGHT";"attribute";"val_uint8";"TRUE";"TRUE"
"RawData";"Height";"attribute";"val_uint8";"TRUE";"TRUE"
"RawData";"height";"attribute";"val_uint8";"TRUE";"TRUE"
"RawData";"humidity";"attribute";"val_uint8";"TRUE";"FALSE"
"RawData";"x";"attribute";"val_int64";"FALSE";"FALSE"
```

Thus, column `humidity` is the only dynamic columns name, where no problems exist, because it is a valid column name and not conflicting.

Whether columns play a role regarding dynamic columns is also listed in the attribute `dynamic_columns_type` of the `ps_info_column` system table. See section 26.3, page 309 for details.

Dealing with Multiple Values for a Measurement

A row may have multiple values for multiple or different `DYNAMIC_COLUMNS_VALUE` columns. In that case, the first row having exactly one `DYNAMIC_COLUMNS_VALUE` value defines the dynamic mapping used.

For example, if the following data is imported:

```
# ts; reg.; id; attribute; values (different types)
2015-07-08 15:01:00; 01;   us771; humidity;   22;    ; 44.0
2015-07-08 15:02:00; 01;   us771; humidity;   67;    ;
2015-07-08 15:03:00; 03;   pow16; humidity;     ; 142;
2015-07-08 15:04:00; 03;   pow16; humidity;  133;    ;
```

the second data row defines that in the corresponding dynamic table the column `humidity` maps to the value in the first value column. As a consequence, the values of the dynamic table are:

```
#ts;id;humidity
2015-07-08 15:01:00.000;"us771";22
2015-07-08 15:02:00.000;"us771";67
2015-07-08 15:03:00.000;"pow16";<NULL>
2015-07-08 15:04:00.000;"pow16";133
```

As you can see, because the second data row maps `humidity` to the first value column, all values are taken from that column (even from the first row, which did define two values). In cases where no value exists, the corresponding rows only contain NULL values (that way you can see that there may be some conflicts).

If there is only input defining multiple values for a key, there will be no clear mapping yet. In that case, the *row* (with the timestamp) will exist in the dynamic table, but without any column listing the corresponding value. Thus, all visible dynamic columns will have null values in the row. In that case, the column will also have no entry yet in `ps_info_dynamic_columns_mapping`.

Details of `DYNAMIC_COLUMNS` Operators

The `DYNAMIC_COLUMNS` operator might have the following clauses (in that order):

- An optional **PARTITION BY** clause
- A mandatory **JOIN_COLUMNS** clause
- An optional **DYNAMIC_COLUMN_NAMES** clause
- An optional **STATIC_COLUMN_NAMES** clause

The clauses have the following meaning:

- The optional `PARTITION BY` clause is necessary for distributed tables (see section 6.3, page 53) that have no `EVERYWHERE` distribution. In that case, the PARTITION BY clause must contain at least the distribution column or a column separated by the distribution column.

  The effect of the `PARTITION BY` clause is that the data is split into buckets depending on the `PARTITION BY` field similar to `GROUP BY` semantics. This means that NULL values may be grouped together and are not added implicitly to the `JOIN_COLUMNS`.

  If you have an `EVERYWHERE` distribution or a table with multiple partitioning columns, using a `PARTITION BY` clause increases the amount of parallelization used internally. For example, if a table has a partitioning column `id` a query such as

```
SELECT ts, id, humidity, pressure
       FROM DYNAMIC_COLUMNS(ON RawData
                            JOIN_COLUMNS(ts, id))
       ORDER BY ts, id
```

  will usually perform better if the fact that `id` is a partitioning column is taken into account in the `DYNAMIC_COLUMNS` clause:

```
SELECT ts, id, humidity, pressure
       FROM DYNAMIC_COLUMNS(ON RawData
                            PARTITION BY id
                            JOIN_COLUMNS(ts))
       ORDER BY ts, id
```

  All `PARTITION BY` fields must either be partitioning columns of the table or separated by one.

- The mandatory `JOIN_COLUMNS` clause as described above (see section 7.2.1, page 63) must contain the first `ORDER BY` column of the raw table and may contain additional columns. It is used to decide which data to join in the resulting dynamic table.

  Any physical column that shall be visible in the resulting dynamic table and is neither a dynamic key nor a dynamic value column nor part of the `PARTITION BY` clause must be listed here. All columns listed in this clause must be physical columns of the table and must be no `MULTI_VALUE` columns.

- The optional `DYNAMIC_COLUMN_NAMES` and `STATIC_COLUMN_NAMES` clauses allow to define alias names for the dynamic and for the static columns of the dynamic table. This allows for dynamic columns names of existing static columns, while still using the static columns under a different name. For example:

```
SELECT ts, raw_id, *
       FROM DYNAMIC_COLUMNS(ON RawData
                            PARTITION BY id
```

```
                                    JOIN_COLUMNS(ts)
                                    DYNAMIC_COLUMN_NAMES(humidity AS id, rpm AS dz)
                                    STATIC_COLUMN_NAMES(id AS raw_id))
          ORDER BY ts, raw_id
```

maps the dynamic columns with the name `humidity` to the name `id`, dynamic columns with the name `rpm` to the name `dz`, while giving the existing static column name `id` the name `raw_id`. If columns with dynamic aliases don't exist, they are simply ignored.

Note that all physical columns of the raw table that shall be visible in the resulting dynamic table must be columns in the `JOIN_COLUMNS` clause or the `PARTITION BY` clause.

Current Restrictions

The following general restrictions apply regarding the dynamic columns feature:

- Physical `MULTI_VALUE` columns can't be part of dynamic tables generated with the `DYNAMIC_COLUMNS` operator (although they can be part of the raw tables).
- All values of key columns should be transformed to lower case prior to importing them due to planned incompatible changes in future versions.

Database Configuration

Conf Directories and INI Files

Conf Directories

The Cisco ParStream database configuration can be split into multiple files to ease the setup of multiple servers. These files must be located in the same directory, the so-called *conf directory*.

On startup, Cisco ParStream checks the following locations to find the *conf directory*:

1. The containing directory if passed the `--inifile` *`filepath`* option
2. The directory passed with `--confdir` *`dir`*
3. The sub-directory `./conf` in the current working directory
4. The directory `conf` in the directory of the Cisco ParStream executable
5. The directory `/etc/parstream`

Within these directories, Cisco ParStream searches for the general configuration file `parstream.ini` and additional configuration files, which can be:

- INI files for server and distribution definitions

The general format of the INI files is described in the following sections of this chapter.

In the past (until Version 4.0) you could also define PSM files in the conf directory, which were used to define the schema used by the database. These files had the suffix `.psm` and contained in essence `CREATE TABLE` statements plus possible comments with leading `--` (see section 24.2, page 278). For backward compatibility these files are still processed the first time you start a server and no existing schema definition is found. After this initial usage, they are ignored. The details about how to create and define tables are described in Chapter 24, page 277.

INI File Format

Cisco ParStream configuration files are supplied in the INI file format. A single entry to set an option is formatted as follows:

```
name = value
```

The entry might be global or inside a section. At the beginning, the INI file has global scope (note: before Version 2.0 this only applies to the file `parstream.ini`; see the warning below).

Sections within a hierarchy are defined using square brackets:

```
[section]
name = value
```

Deeper levels of a hierarchy are denoted by a `.` character in the section name.

```
[section.subsection]
name = value
```

Since Version 2.0, you can switch back to the global scope with an empty section name:

```
[]
globaloption = value
```

Warning:

Before Version 2.0, if you have multiple INI files, global options should be set only at the beginning of the file `parstream.ini`. The reason is that after processing `parstream.ini` the order of the other INI files is undefined and further INI files continue with the scope the previous INI file ends with (this was fixed with Version 2.0).

Warning:

If you have any duplicate entries inside your INI files, Cisco ParStream will give you a warning, i.e.: `!! WARNING: duplicate config entry 'server.first.port': old value: '9951', using new value: '33333'` and use the last value read from the file or from the command line. Commandline parameters are preferred over INI file parameters.

Examples

Example: The port of a Cisco ParStream server named first, should be reachable on port 1234.

```
[server.first]
port = 1234
```

A # character at the beginning of a line denotes a comment. For example:

```
# comment name = value
```

For all these settings in INI files you can pass command line arguments when the server gets started.

These command line arguments overwrite the corresponding INI file setting.

See Starting the Server#commandline-args and Commandline Arguments for details.

Internationalization (I18N)

Different countries have different character sets, character encodings, and conventions. This section describe the features Cisco ParStream provide for internationalization.

In principle, Cisco ParStream supports different character sets (7-bit, 8-bit with different encodings, UTF8).

However, string sorting depends on which character set and encoding is used. For this reason, since Version 2.0, you can to specify the locale to use.

This is a global Cisco ParStream option, which can be specified in the corresponding INI file (see Global Options):

```
locale = C
```

or passed as an option (see Commandline Arguments):

```
--locale=C
```

The only portable locale is the locale called C, which is default. Other locales depend on your operating system. Usually you can call

```
$ locale -a
```

to query the supported locales on your platform. For example, the following might be supported:

```
locale = C     # ANSI-C conventions (English, 7-bit, provided on all platforms)
locale = POSIX   # POSIX conventions (English, 7-bit)
locale = en_US # English in the United States
locale = en_US.utf8 # English in the United States with UTF8 encoding
locale = de_DE # German in Germany
locale = de_DE.ISO-8859-1 # German in Germany with ISO-Latin-1 encoding
locale = de_DE.ISO-8859-15 # German in Germany with ISO-Latin-9 encoding (having the
    Euro-Symbol)
locale = de_DE.utf8   # German in Germany with UTF8 encoding
```

Note:

If in a cluster different locales are used so that sorting is not consistent, Cisco ParStream will run into undefined behavior.

Server Administration

To setup and start a Cisco ParStream database, a couple of steps are necessary:

- You should define a database configuration. This is usually done in INI files in the *conf directory* (see section 8.1.1, page 73).
- You have to start the servers.
- You might also want to import data, which is described in Chapter 10, page 88.

This chapter describes how to start a Cisco ParStream server with `parstream-server` and how to deal with authorization and logging.

Starting the Servers

Starting a Cisco ParStream Server

After installing Cisco ParStream and providing a corresponding configuration, the easiest way to start a Cisco ParStream server is to perform the following steps:

1. Set variable PARSTREAM_HOME:

   ```
   export PARSTREAM_HOME=/opt/cisco/kinetic/parstream-database
   ```

   and add Cisco ParStream libraries to your library search path:

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PARSTREAM_HOME/lib
   ```

2. Go to the directory, where your Cisco ParStream database configuration is located. Usually, this should be a directory containing:

   - a sub-directory `conf` which typically contains the INI file for the basic configuration
   - a sub-directory `partitions` or any other `datadir` specified in the INI file of the corresponding server.

3. The INI file has to contain the minimum options that are required to run a server:

   - a global `clusterId`
   - a global `registrationPort`
   - server-specific `host` and `port`
   - a server-specific `rank`

   Note that `clusterId`, `registrationPort` and `rank` are required because any Cisco ParStream server can sooner or later be extended to a multi-node cluster (see chapter 6, page 40).

   You might typically also set:

- a global `clusterInitTimeout`, which defines the time in seconds to wait for a cluster of multiple nodes to establish. If you know that the database is running with only one server/node you can/should set to just `1`.
- the server-specific `datadir`

For example, an INI file for a server called `first` might look as follows:

```
# basic cluster settings for single-node clusters:
clusterId = exampleCluster
registrationPort = 9040
clusterInitTimeout = 1

[server.first]
host = localhost
port = 9042

# server rank in multi-node clusters:
rank = 1

# directory that contains the data partitions:
datadir = ./partitions
```

4. Start the Cisco ParStream server `parstream-server`. The only required argument is the name of the server to start, which identifies the corresponding section in the INI files for the other settings.

   For example:

```
\$PARSTREAM_HOME/bin/parstream-server first
```

   Now, the server is ready to process queries sent to the configured host and port.

   If you add `$PARSTREAM_HOME/bin` to your search path, starting the server as follows is enough:

```
parstream-server first
```

   For more sophisticated ways to start a Cisco ParStream server (e.g. logging all output into a file), see the systemd daemon, the shell script `start.sh` in `$PARSTREAM_HOME/bin` or the `run_example.sh` scripts in the various examples, Cisco ParStream provides.

Note:

- The Cisco ParStream server uses multiple ports. For this reason, the server option `port` has to be set. This port and up to four consecutive ports will be used by Cisco ParStream. See section 13.3.1, page 135 for details.
- The server uses journal files to store cluster metadata. See section 6.2.4, page 49 for details.

Cisco ParStream Server Command Line Parameters

The Cisco ParStream server provides the ability to pass several commandline arguments, to configure and influence the behavior of the server.

On one hand you can pass several standard arguments:

Argument	Effect
`--help`	print syntax including all commandline options
`--version`	print version
`--confdir` *dir*	use `dir` as the conf-directory with the configuration (INI files)
`--configcheck`	just check the configuration without starting the server
`--inifile` *filepath*	use `filepath` as the configuration (INI) file
`--verbosity` *arg*	change verbosity (0: minimum, 5: maximum)

In addition, all values of the configuration file can be overwritten by command line parameters. The prefix of a parameter string depends on the section, in which it is defined.

For example, to overwrite the parameter `port` of the server with the name first with the value `9088`, the command line parameter must be formulated as follows:

```
--server.first.port=9088
```

User Authentication

Cisco ParStream provides the ability to enable user authentication to restrict access to the database. By default, users have to authenticate to the server by providing login credentials if they want to retrieve data . Access to the server is only granted if the authentication is successful.

Note the following:

- User authentication is only enabled for the external communication via the first two server ports (see section 13.3.1, page 135). For this reason, to secure the communication in a Cisco ParStream cluster, you should disable the access to all internal ports for the average user.
- Clients send pass phrases currently in clear text so that you should use secure network connections for connected clients.
- There is currently no authorization management established by Cisco ParStream providing different roles and responsibilities. Any successfully logged in user can perform any remote valid database operation.

While *pass word* is the widely established common term, we use *pass phrase* to remind people that the length of such a secret is the most important factor, and the effort needed to crack simple words or short secrets is reducing dramatically with every new hardware generation.

User Authentication Approach

The authentication approach Cisco ParStream uses is based on PAM (the *Pluggable Authentication Modules*), which is usually supported by all Linux distributions. Thus, this means that

- PAM provides user management and authentication. By default Cisco ParStream uses system users for authentication. However, you can configure Cisco ParStream to use any other PAM module that meets your requirements.
- Cisco ParStream allows to register these users as **database users** via DDL commands so that Cisco ParStream uses the PAM user with its pass phrase to grant access to the database. Doing this, you can even map a PAM user name to a different database login name.

Cisco ParStream authenticates users via an external application called `parstream-authentication`. This application is provided by an additional software package, that has to be installed separately. For each supported platform, a different package has to be installed (see section 2.6, page 11).

By default, the installation of Cisco ParStream creates a system and database user `parstream`. After you have followed the instructions in section 2.8.1, page 12 and set a pass phrase for the user `parstream`, you can log into the Cisco ParStream database using this user. For this to work, the Cisco ParStream database **has to be started** from this user. Currently it is only possible to log into Cisco ParStream with the system user that has started the database. However, you can create alias database users (see section 9.2.3, page 80) that share the same system user and pass phrase.

Thus, after installing Cisco ParStream to enable authentication for a new user, you have to

- Create a corresponding **PAM user** if not using the default Cisco ParStream PAM module. See the documentation of the selected PAM module how to create new users.
- Create a **database user** that maps to the PAM user, using the `CREATE USER` command (see section 9.2.3, page 80). As an effect, the database user has the pass phrase of the associated PAM user.
- Pass login name of the database user and the same pass phrase of the associated PAM user with each client that connects to Cisco ParStream:
  - See section 12.1.1, page 110 for authentication with `pnc`.
  - See section 12.1.2, page 113 for authentication with `netcat` (note however that you should prefer `pnc` because `netcat` doesn't hide the pass phrase input).
  - See section 19.5.22, page 229 for authentication with the Java Streaming Import Interface (JSII).
  - See section 18, page 214 for authentication with JDBC.

Note the following:

- Multiple database users can map to the same PAM user (but not vice versa).
- A database user can map to a PAM user that does not exist (yet). But, it can only be used to login with clients after the underlying PAM user was created. This way, you can temporarily remove a PAM user and create a new PAM user with the same user name to change the pass phrase.
- There is no authentication necessary to import data with the CSV importer. Authentication is only required for streaming imports and when retrieving data from clients.
- You can choose between different PAM service configurations. By default, the ParStream configuration is installed, which defines the following options with its default values:

```
[authentication]
pamService = parstream
```

```
authenticationWrapperExecutable =
    /opt/cisco/kinetic/parstream_authentication_1/parstream-authentication
```

You can define other service configurations using different names and specify its name here.

Migrating from Previous Database Versions

If you migrate from a previous Cisco ParStream database version, the system will **not** create the `parstream` database user upon first startup automatically. As authentication is on by default, you will not be able to authenticate and thus use the database. The following steps have to be taken to setup your database for authentication:

- Shutdown the database cluster
- Disable authentication for all database servers in the cluster. See section 9.2.4, page 81 for details.
- Start the database cluster. You can now log into the database without user authentication.
- Create the `parstream` database user. See section 9.2.3, page 80 for details.
- Shutdown the database cluster.
- Enable authentication for all database servers in the cluster.
- Start the database cluster. You can now log into the database with your newly created user.

This procedure has to be performed once for each migrated cluster.

Managing Database Users

Creating a Database User

To create a database user that maps to a PAM user `jdoe` you have to call:

```
CREATE USER 'jdoe'
```

The effect is, that provided the PAM user `jdoe` exists, the database user `jdoe` can login to the database when starting a client connection using the associated PAM user pass phrase.

Alternatively, you can map the PAM user name to a different database user name:

```
CREATE USER 'jdoe' WITH LOGIN 'john'
```

Here, the database user `john` will map to the PAM user `jdoe`, using its associated pass phrase to establish client connections.

Note:

- The database user is shared among all nodes of a cluster, thus it is up to the database administrator to ensure that all cluster nodes provide a corresponding PAM user consistently.
- Multiple database user can map to the same PAM user.

Removing a Database User

To drop a database user you have to call `DROP USER` with the database user name. For example:

```
DROP USER 'jdoe'
```

or:

```
DROP USER 'john'
```

Note:

- Any user who is logged into the database using its authentication credentials can drop any other user. If the user is currently connected to the database, the user can finish its queries, but can't login again.
- Even if the user is still connected, it is no longer listed in the list of database users see section 26.4, page 316.
- To avoid to be able to connect to the database again, you can't drop the last user.
- You can also drop the default user `parstream` provided it is not the last user.

Listing all Database Users

The system table `ps_info_user` lists all current database users:

```
SELECT * FROM ps_info_user
```

Users already dropped but still connected are not listed.

See section 26.4, page 316 for details.

Disabling Authentication

To disable user authentication at all, you have to set the global option `userAuthentication` (section 13.2.1, page 120):

```
userAuthentication = false
```

As with any option, you can pass this option via command line:

```
parstream-server --userAuthentication=false ...
```

DBMS Scheduler

The DBMS Scheduler allows users to define tasks that are executed periodically. A task consists of a unique name, an action to be performed, a timing that defines when the task is to be executed, an optional comment, and an ’enabled’ flag that defines whether the newly created job should be active immediately.

The task is executed by the cluster according to the defined timing. Should any errors occur during the execution of a job, an error message is logged in the leader’s log file. Please note that failing tasks will not be automatically disabled by Cisco ParStream.

If an action of a job uses a table that is removed via a `DROP TABLE` command, the job will be removed as well.

Listing all Jobs

All configured jobs along with their actions, timings, and whether they are enabled or disabled, are listed in the system table `ps_info_job`.

Creating a Job

You can create a job by using the CREATE JOB command followed by a unique identifier for the job name and a description of the job in the json format. By default, a job is enabled and starts running once the next configured point in time is reached.

To create a job named "exampleJob" that inserts the value 1 every second into a table named "exampleTable" with the only column "id", you would use the following statement:

```
CREATE JOB exampleJob { "action": "INSERT INTO exampleTable SELECT 1 AS id;",
    "timing": "* * * * *" };
```

Please note that the name of a job follows the rules of SQL identifiers. Hence, all name matching is done in a case-insensitive way unless the identifier is quoted.

The ’comment’ and ’enabled’ json fields are optional. The ’comment’ field allows you to add a description to the job. The ’enabled’ field defines whether the job should be enabled after creation. By default, all newly created jobs are enabled.

Disabling a Job

You can disable an enabled job by using the DISABLE JOB statement:

```
DISABLE JOB exampleJob;
```

A disabled job will no longer be executed automatically until it is re-enabled.

Enabling a Job

You can enable a disabled job by using the ENABLE JOB statement:

```
ENABLE JOB exampleJob;
```

Manually Running a Job

Apart from the automatic execution of a job according to its configured schedule, you can execute a job manually by using the RUN JOB statement:

```
RUN JOB exampleJob;
```

Removing a Job

You can remove a job from the system by using the DROP JOB statement:

```
DROP JOB exampleJob;
```

Stored Procedures

Stored procedures can be used to store SQL statements and reuse them by using a routine identifier.

Create a Stored Procedure

`CREATE PROCEDURE` defines a new stored procedure. A procedure is defined by a unique identifier, an argument list and a procedure definition.

The following example defines a procedure named `ageFilter` and has three parameters:

```
CREATE PROCEDURE ageFilter(
  firstNameFilter VARSTRING,
  minAge UINT8,
  maxAge UINT8)
LANGUAGE SQL AS
  SELECT age
  FROM persons
  WHERE first_name = firstNameFilter
    AND age >= minAge
    AND age <= maxAge;
```

The Cisco ParStream implementation of stored procedures has the following functional properties:

- The procedure identifier must be unique.
- No function overloading is possible.
- The parameter list can be empty.
- Default values for parameters are not supported.
- Routine characteristics are restricted to SQL only.
- Only single statements in the procedure definition are accepted.
- Only input parameters are supported.

Execute a Stored Procedure

To execute a previously defined stored procedure the `CALL` command can be used like:

```
CALL ageFilter('John', 18, 30);
```

where `ageFilter` is the routine identifier which was used to define the procedure in the `CREATE PROCEDURE` command. Additional arguments are defined in parentheses and separated by commas.

Drop a Stored Procedure

To remove no longer required procedures the following command can be used:

```
DROP PROCEDURE ageFilter;
```

Monitoring, Logging, and Debugging

Monitoring

To ease analysis of performance problems in Cisco ParStream, detailed event logs of the following processes are potentially recorded:

- Start of a server or importer
- Query execution (according to ExecTree option `MonitoringMinLifeTime`, see section 13.3.4, page 141).
- Import execution (according to ExecTree option `MonitoringImportMinLifeTime`, see section 13.3.4, page 141).
- Merge execution (according to ExecTree option `MonitoringMergeMinLifeTime`, see section 13.3.4, page 141).

Whenever execution of any of these exceeds the given threshold (or for server starts all the time) events are written to files in the directory specified by the global option `performanceLoggingPath` see section 13.2.1, page 129.

The following files are (potentially) written:

- `monitoring_`*`username`*`_`*`date`*`.log` recording per-execution query, merge and import events, persisted after each execution exceeding the given runtime limit.
- `serverStart_`*`username`*`_`*`date`*`.log` recording server start events, persisted after each server start.
- `execthreadpool_`*`username`*`_`*`date`*`.log` recording statistics and events of the global execution scheduler, persisted during server shutdown.

Here, *username* and *date* are replaced by the current username (usually the value of the `USER` environment variable) and date upon the time of persisting.

The detail level of the execution related event logs can be controlled by the ExecTree options `QueryMonitoringTimeDetailLevel`, `MergeMonitoringTimeDetailLevel`, `ImportMonitoringTimeDetailLevel`, for query, merge, and import execution, respectively (see section 13.3.4, page 141).

The detail level of the global scheduler event log can be controlled by the global/server option `executionSchedulerMonitoringTimeDetailLevel` (see section 13.2.1, page 129 and see section 13.3.2, page 139).

When you encounter performance problems you can use the mentioned options to generate event logs.

Logging

Cisco ParStream provides four different types of protocol messages, written by servers and importers:

Type	Meaning
`ERROR`	All errors that occur in Cisco ParStream server or importer
`PROT`	All types of information that are important to the user, e.g. version of Cisco ParStream, incoming requests, ...
`WARN`	Minor errors that occur and can be handled within Cisco ParStream server and importer
`INFO`	Minor information for the user, e.g. Cisco ParStream configuration

These messages have a unique output format:

```
[_timestamp_]:_Unique query ID_:_message type_(_message number_): _message
   text_ (optional: function that emit this message, file, line)
```

For example:

```
[2012-07-23T12:59:50]:server3-125949:PROT(77011): Starting to listen on port
   9042 for client connections
[2012-07-23T14:42:45]:server3-125949:ERROR(70024): Unknown command: slect *
   from Address (function: EQueryResultType
   parstream::ParStreamServer::executeCommand(const std::string&,
   parstream::ClientConnection&)   file: src/server/ParStreamServer.cpp,
   line: 1625)
```

All of these four message types are logged to `stderr`.

Verbosity

Using the global option `verbosity` (see section 13.2.1, page 120 you can enable more verbose output of servers and importers. For example:

```
verbosity = 2
```

The effect for higher verbosity is for example as follows:

- Servers print list of loaded dynamic libraries.

- Servers log complete results instead of just the beginning.
- Importers print more details about available tables.

Debugging

For debugging and analysis support, you can use global option and class specific settings.

Note: The class names internally used are not standardized and may change from version to version.

The global options `defaultDebugLevel` (see section 13.2.1, page 120) defines the default debug level.

Debugging Support at Start Time

You can specify default debug level in the global configuration section of an INI file:

```
defaultDebugLevel = 1
```

Value `0` means that no debug output is written. Values `1` (minimum) to `5` (maximum) enabled debug output of all internal behavior.

A class specific debug level may also be configured in the INI file in the subsection ”DebugLevel”. The class name is case sensitive.

For example:

```
[DebugLevel]
ClientConnection=5
PartitionManager=1
ExecThreadPool=2
```

Granularity of the class debug switches are different for each class. For example all execution nodes use the following debug levels:

Level	Debug Output
1	Class instance specific like construction, destruction and parameters
2	Blockspecific like block processing, finished, etc.
3	Rowspecific like processing or filtering of rows
4	Valuespecific like Calculation, Assignment, etc.

Debugging Support at Running Time

You can also change the general or specific debug level at runtime using an `ALTER SYSTEM SET` command.

For example, the following command changes the global debug level of the server that receives the command:

```
ALTER SYSTEM SET DebugLevel.defaultDebugLevel=3
```

For example, the following command changes a class specific debug level of the server that receives the command:

```
ALTER SYSTEM SET DebugLevel.ClientConnection=3
```

To reset a debug level back to default (i.e. tracking the see section 13.2.1, page 120 again):

```
ALTER SYSTEM SET DebugLevel.ClientConnection TO DEFAULT
```

Note that the default for defaultDebugLevel is 0.

Detailed Exception Information

You can use the debugging feature to enable more detailed reporting of process information in case of exceptions. If the debug level of the exception class is set to a value greater than 0, a file with process info is created every time an exception occurs.

The file is named after the pattern `ParStream-ProcInfo-<PID>-<TIMESTAMP>.txt` and is placed in the server's working directory. It contains the following information:

- timestamp
- version info
- process status
- open file descriptors
- memory maps

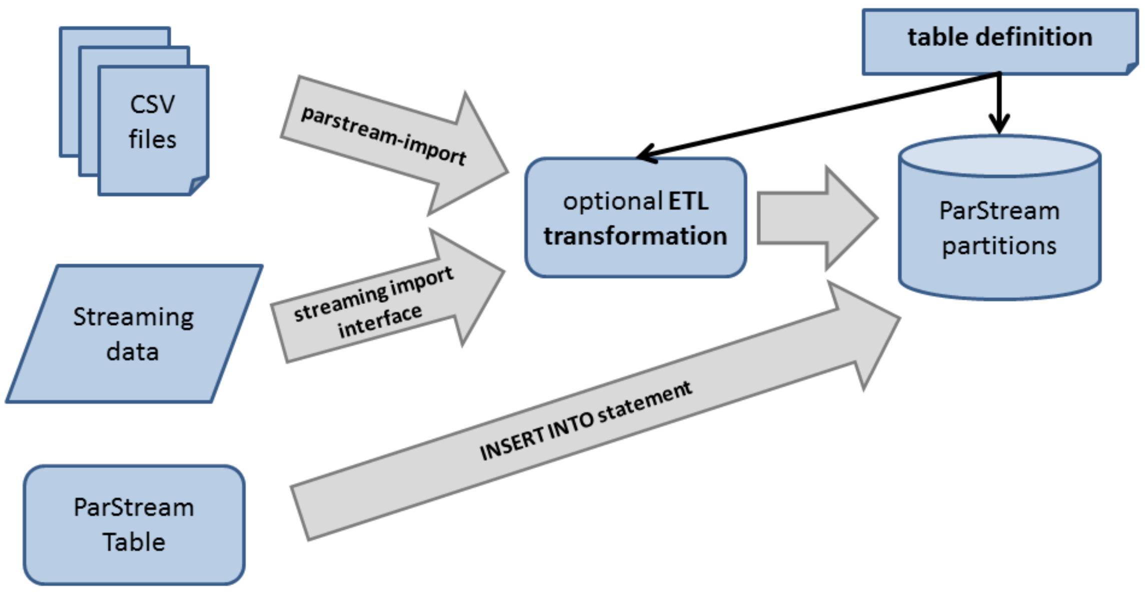
It must be enabled in the INI files as follows:

```
[DebugLevel]
Exception = 1
```

Importing Data

Cisco ParStream provides different efficient ways to import (huge amount of) data. The general import concepts, the executables, their options and possibilities, the format of import data and the format of the resulting partition data are introduced and explained in this chapter.

Overview of Data Import



In principle, Cisco ParStream can import data into the database as follows:

- The schema for the table to import has to be defined via `CREATE TABLE` statements and sent to one cluster node (See chapter 24, page 277 for details).
- The data to import can be provided by three ways:
  - You can use the **CSV importer** **`parstream-import`**, which reads data (in a loop) provided as CSV files.
    The format of CSV files is explained in section 10.3, page 89.
    `parstream-import` is described in section 10.5, page 99.
  - You can use the **Java Streaming Import Interface**.
    This allows to implement your own importing executables using a Java API provided by Cisco ParStream to import the data.
    The Java Streaming Import Interface is described in Chapter 19, page 216.
  - You can use the **INSERT INTO** statement.
    This allows to insert data inside Cisco ParStream from one table to another.
    The INSERT INTO feature is described in section 10.7, page 107.
- The imported data is stored into partitions, which discussed in section 5.1, page 29.
- You can import data, while tables are modified. See section 24.3.1, page 295 discussed in section 5.1, page 29.

When importing data from external sources, you can use so called **ETL statements** (“extract”, “transform”, and “load”) to transform the data you read into the format of the database columns For example, you can add additional columns for better partitioning or filter rows according to a WHERE clause. Note, however, that ETL import statements don’t apply to INSERT INTO statements. See section 10.6, page 104 for details.

General Import Characteristics and Settings

General Import Characteristics

Please note the following general characteristics of any import in Cisco ParStream:

- `NULL` handling:
  - Note that for strings and integral data types specific values represent `NULL`. For example, empty strings or the value 255 for type `UINT8`.
  - You can’t import rows containing only `NULL` values. Rows must have at least one column having a non-`NULL` value.
- When processing schema updates or servers are about to shut down, running imports are first finished. Thus, schema updates and regular shutdowns take effect after a running import is committed.

General Import Settings

The general ability to import can be controlled by the following options and commands:

- Using the global server option `enableImport` (see section 13.2.1, page 122) you can initially enable or disable imports. The default value is `true`.
- By calling `CLUSTER DISABLE IMPORT` or `CLUSTER ENABLE IMPORT` (see section 16.4.1, page 200) you can (temporarily) disable and enable imports in cluster scenarios.

General Format of CSV Import Files

One major source of input data for the Cisco ParStream database are comma separated value (CSV) files. The types of the values in each column are specified with the `CREATE TABLE` statement (see section 24.2, page 278). Every table is filled by a separate CSV file. That is, the CSV files serve basically as a human readable representation of the initial state of the table or additional state an import brings into the database.

Each row needs to have the same number of column entries, separated by the column separator (default: “`;`”).

The CSV File Format in General

CSV files have the following **general format**:

- Columns are separated by the column delimiter, which is the semicolon (“`;`”) by default.
- Rows are delimited by the end of an input row.
- The character `#` at the beginning is used for comments (introduces rows that are ignored).

Thus, the following CSV file content defines two rows with four columns:

```
# A simple CSV example having two rows with four columns:
# string; int; double; date
first line;42;12.34;1999-12-31
second line;77;3.1415;2000-12-24
```

Values, are validated during CSV import. Thus, the example above has to match a corresponding table definition such as the following:

```
CREATE TABLE MyTable (
  myString VARSTRING,
  myInt INT64,
  myFloat DOUBLE,
  myDate DATE,
)
...
```

If values don’t fit, the whole row is ignored and stored in a special file containing all rejected rows (see section 10.5.2, page 101). The same applies if the number of fields in the CSV input does not match the number of columns.

Basic Processing of CSV Fields

For the individual **values**, specified between the column delimiter or the beginning/end of row, the following rules apply:

- Leading and trailing spaces of values are **always** skipped.
- With leading and trailing spaces removed, the empty value, the value “`\N`”, and the value “`<NULL>`” always represents **NULL**. This is not the case if additional characters (except leading/trailing spaces) exist or the value is quoted. See section 10.3.2, page 93 for details.
- You can put the whole value (inside leading and trailing spaces) within **double quotes**. This applies to any type. By this feature, you can have leading or trailing spaces inside the value. You can also use double quotes to have the column separator as part of the value. Double quotes in the middle of a value have no special effect.
- A **backslash** (outside or within double quotes) disables any special handling of the following character. The following character is taken as it is. Thus:
  - “`\"`” makes the double quote character part of the value instead of beginning/ending a quoted value.
  - “`\;`” is not interpreted as column separator even without quoting the value.
  - “`\x`”, “`\t`”, and “`\n`” are just the characters “x”, “t”, and “n” (the leading backslash is ignored).
  - “`\\`” is just the backslash character as part of the value.

For example, if you have input for two columns, a line number and a string:

```
# line;string
1; nospaces
2;" withspaces "
3;"\"with quotes\""
4;\;
5;\"
6;"\""
```

the strings specified as second field are:

- in line 1: the string “`nospaces`” (leading and trailing spaces are skipped)
- in line 2: the string “ `withspaces` ” (leading and trailing spaces are not skipped)
- in line 3: the string “`"with quotes"`” (leading and trailing spaces are not skipped)
- in line 4: a semicolon.
- in line 5: just the character “`"`”
- in line 6: also just the character “`"`”

Note that queries for this data by default will escape the double quotes.

Skipped CSV Columns

To be able to process CSV files with more fields/columns than necessary, you can declare tables to have `SKIP`ed columns. By this, you define a column that from the point of the database schema does not exist, except that they are listed in the system table `ps_info_column` (see section 26.3, page 309).

For example:

```
CREATE TABLE MyTable (
  id UINT64,
  name VARSTRING,
  ignored VARSTRING SKIP TRUE,
  value DOUBLE,
)
...
```

defines `MyTable` to have 3 columns, `id`, `name`, and `value`, using the first, second, and fourth CSV input field of each row of the following CSV file:

```
# id; name; ignored; value
1; AA-143;some additional ignored information;1.7
1; AA-644;other additional ignored information;7.3
...
```

Note that you still have to specify the correct column type of skipped columns, because even ignored input fields have to match the format (simply, use type `VARSTRING` if the format doesn’t matter).

The Column Separator

You can use other characters than the semicolon as CSV column separator with the importer option `columnseparator` (see section 13.4.2, page 146).

For example, when a dot is used as separator:

```
[import.imp]
columnseparator = .
```

you have to specify floating-point values and other values using a dot within double quotes:

```
# A simple CSV example having two rows with four columns:
# string; int; double; date
first line.42."12.34".1999-12-31
second line.77."3.1415".2000-12-24
```

Again, you can also simply put double quotes around all values:

```
# A simple CSV example having two rows with four columns:
# string; int; double; date
"first line"."42"."12.34"."1999-12-31"
"second line".77"."3.1415"."2000-12-24"
```

This is especially necessary if you define a columns separator that also might be part of a value.

You can also define non-printable characters as column separators in different ways:

- You can either specify the column separator with a backslash. To specify other non printable character you can define it as the following escaped characters: `\a`, `\b`, `\f`, `\t`, `\v`, `\?`, or `\'`.
  For example, you can define a tabulator as column separator as follows:

  ```
  [import.imp]
  columnseparator = \t
  ```

- You can define the octal or hexadecimal value of the column separator: `\x##`, `0x##`, or a octal value defined with only a leading 0.
  For example, you can define a tabulator as column separator as follows:

  ```
  [import.imp]
  columnseparator = 0x09
  ```

The following characters are not allowed to be column separators: “`\`”, “`"`”, space, and newline.

Note that when using a comma as column separator, you can import multivalues only if they with all their elements are quoted as a whole, because they also use the comma as internal separator (see section 10.4.6, page 98).

Compression of Import Files

The CSV import files can be uncompressed, or they can be compressed by using either bzip2 or gzip. The compression type will be determined by examining the file extension `.bz2` or `.gz` respectively.

If none of those extensions is used, uncompressed text is assumed.

Note that `.tgz` is not supported.

Importing `NULL` Values

CSV values can be empty. In that case the values are always interpreted as `NULL` values.

Note that rows where all values are `NULL` are never imported.

In the following example, the last column of the first row and the second column of the second row are initialized by `NULL` values.

```
# two rows with 3 columns with NULL values:
first line;42;
second line;;date'2000-12-24'
```

In addition, one can use `\N` or `<NULL>` in CSV files as `NULL` values. This only applies if the whole value except leading and trailing spaces has the corresponding characters. Any additional character or double quotes let the value have the corresponding two or six characters.

To clarify that for strings and MultiValues:

- **Strings**: The empty string is interpreted as a `NULL` string (see section 23.5.1, page 273). Importing `\N` and `<NULL>` is interpreted as importing an empty/`NULL` string. Importing `"\N"` and `"<NULL>"` results in strings containing exactly that characters. Note that this means that rows having only empty string values are never imported.
- **MultiValues**: Importing `\N` and `<NULL>` or an empty CSV input value will be interpreted as `NULL`. Currently there is no difference between an empty multivalue and a multivalue containing exactly one `NULL` element. `IS NULL` is `TRUE` if the multivalue is empty. `IS NOT NULL` is the opposite (non-empty).

This table summarizes the special cases:

Type	`IS NULL`	`IS NOT NULL`	Valid `NULL` import
String	Any empty string	Any string that is not empty	empty string, with or without `""`, `\N`, `<NULL>`
Multivalue	empty	non-empty	empty multivalue, `\N`, `<NULL>`

CSV File Format of Specific Types

Importing Integers

Values for unsigned and signed integer typed columns will be parsed from integer values with decimal base.

Format	Examples
[−]*digits*	`42`
	`-12`
NULL	`<NULL>`
	`\N`

As usual, an empty CSV value also is interpreted as `NULL`.

Note that you have to put the value within double quotes if the minus character is defined as column separator (see section 10.3.1, page 92).

Importing Floating-Point Numbers

Floating-point numbers can be parsed for the value types FLOAT and DOUBLE from the following formats:

Format	Examples
[−]*digits* . *digits*[`e`(+/−)*digits*]	`-1.2`
	`3.24e+23`
NULL	`<NULL>`
	`\N`

As usual, an empty CSV value also is interpreted as `NULL`.

Note that you have to put the value within double quotes if the dot, minus, plus or 'e' character is defined as column separator (see section 10.3.1, page 92).

Importing Date and Time Formats

Date/time values have a default import format, you can use. However, since Version 2.2, you can specify any other date/time import format specified by a `CSV_FORMAT` clause, which is described below (see page 95).

Supported default import formats for type **`date`** are:

Format	Examples	Remark
YYYY-MM-DD	2010-11-23	
DD.MM.YYYY	23.11.2010	Supported until Version 2.2. Use `CSV_FORMAT 'DD.MM.YYYY'` instead, then.
MM/DD/YYYY	11/23/2010	Supported until Version 2.2. Use `CSV_FORMAT 'MM/DD/YYYY'` instead, then.
NULL	`<NULL>`	
	`\N`	

Supported formats for type **`shortdate`** are:

Format	Examples	Remark
YYYY-MM-DD	`2010-11-23`	
unix timestamp	`1287059484`	
DD.MM.YYYY	`23.11.2010`	*Supported until Version 2.2. Use `CSV_FORMAT 'DD.MM.YYYY'` instead, then.*
MM/DD/YYYY	`11/23/2010`	*Supported until Version 2.2. Use `CSV_FORMAT 'MM/DD/YYYY'` instead, then.*
NULL	`<NULL>`	
	`\N`	

Supported formats for type **`time`** are:

Format	Examples
HH24:MI:SS	`12:30:21`
HH24:MI:SS.MS	`12:30:21.865`
NULL	`<NULL>`
	`\N`

Supported formats for type **`timestamp`** are:

Format	Examples
YYYY-MM-DD HH24:MI:SS	`2010-11-23 12:30:21`
YYYY-MM-DD HH24:MI:SS.MS	`2010-11-23 12:30:21.865`
NULL	`<NULL>`
	`\N`

For all date/time types, as usual, an empty CSV value also is interpreted as `NULL`.

Note that you have to put the value within double quotes if one of the characters "`-`", "`:`", or "`.`" is defined as column separator (see section 10.3.1, page 92).

Supporting other Date and Time Formats with `CSV_FORMAT`

Since Version 2.2., the `CSV_FORMAT` clause inside the `CREATE TABLE` statement (see section 24.2.5, page 289) allows to define how date/time types are interpreted by CSV importer. The format mask can not be used for ETL (see section 10.6, page 104) generated CSV columns.

An example for a CREATE TABLE statement including a column specific format mask could look like the following:

```
CREATE TABLE dateTable
(
  dateColumn DATE ... CSV_FORMAT 'DD.MM.YYYY'
)
...
```

Now, import values such as 24.12.2013 are supported (instead of the default format 2013-12-24).

The following table contains all supported format specifiers which can be used in Cisco ParStream to define a format mask:

Specifier	Range	Length	Def.	Description
`HH`	01-12	1-2	0	Note: Default is 12 hour format
`HH12`	01-12	1-2	0	Hour with 12 hour format
`HH24`	00-23	1-2	0	Hour with 24 hour format
`MI`	00-59	1-2	0	Minute
`SS`	00-59	1-2	0	Second
`MS`	000-999	1-3	0	Millisecond
`AM or PM`	AM, PM, A.M., P.M.	2 or 4	AM	Meridian indicator for HH and HH12 (both `AM` and `PM` allow all values)
`YYYY`	0000-9999 (date) 2000-2187 (shortdate)	1-4	0	Year (4 digits)
`YY`	00-99	1-2	0	Two digit year; Includes years 1930-1999 and 2000-2029.
`MM`	01-12	1-2	1	Month number
`DD`	01-31	1-2	1	Day of month
`EPOCH`	0-253450598399	1-12		Unix/POSIX Time which covers all information (time and date) (e.g. 15148801). Max is 253450598399 which is the 31. Dec. 9999 23:59:59
`EPMS`	0-253450598399999	4-15		Like EPOCH. This format uses milliseconds as smallest unit (instead of Unix timestamp typical seconds).

In addition you can use any other character, which is required then at the specified position and can be used as *separator* to separate different value items. Thus, a format such as `day: DD.` would successfully read in “`day: 23.`” (requiring the characters ’d’, ’a’, ’y’, ’:’, and a space, followed by the day value, followed by a dot). The characters are case sensitive.

Note the following:

- If a format mask allows a variable length of digits, you either need separators to detect the end or the maximum number of digits are processed. For example, for the format mask `'DDMMYYYY'` a value such as `'111970'` is not allowed; you have to pass `'01011970'` instead (thus, leading zeros are required because no separators are given).

Other examples for format mask are:

Example Input	Format Mask	Description
`01.01.99`	`MM.DD.YY`	Using the two digit year with “`.`” separators.
`11:59:59.999 PM`	`HH12:MI:SS.MS AM`	Time with milliseconds and meridian given.
`2007-08-31T16:47`	`YYYY-MM-DDTHH24:MI`	Timestamp using ’T’ as a date-time separator.
`13284555.999`	`EPOC.MS`	Unix timestamp with given milliseconds.
`date 2013-12-31`	`date YYYY-MM-DD`	Date with character literals

Importing Strings

As introduced as the general import format in section 10.3.1, page 89, strings are imported by removing leading and trailing spaces. As for all types, you can encapsulate the value with double quotes ("), which allows to have leading and trailing spaces and the column separator inside the string. With and without enclosing double quotes, you can also escape any character by a backslash, which disables any special meaning.

Note that in ASCII query output the strings are enclosed by double quotes and any special character inside the string (double quote or column separator character) is escaped by a backslash. This ensures a round-trip capability of string: String returned by query results are always valid for imports resulting in the same value.

For example, the CSV input:

```
#no;string
1; hello
2;" \"world tn\" "
3;
;last
```

defines in the second column the input strings (value between > and <) >`hello`<, > `"world tn"` <, the empty string (NULL) and >`last`<. In the first column, the last row defines NULL as value.

A query will output the lines as follows:

```
#no;string
1;"hello"
2;" \"world tn\" "
3;""
<NULL>;"last"
```

Thus, NULL strings are printed as empty strings and for other types the output for NULL values by default is `<NULL>` (which can be modified via the global option `asciiOutputNullRepresentation`; see section 13.2.1, page 122). This output can be used as input again and will result to the same internal values.

Again, note that importing an empty string or the values `\N` and `<NULL>` without additional characters except leading and trailing spaces results in a string being NULL. (see section 10.3.2, page 93).

Note also:

- With `MAPPING_TYPE PROVIDED` you can define the hash values yourself, using the following syntax: “*`hash`*`:`*`value`*” (see section 24.2.6, page 294)

Importing Blobs

Values for blobs can be specified as values for strings (see section 10.4.4, page 97).

Importing MultiValues

Numeric columns can be stored as multivalues by specifying a `MULTI_VALUE` singularity (see section 24.2.4, page 284). These types roughly correspond to arrays and imply certain indexing options.

Note the following about multivalues:

- Multivalue elements are separated by commas in CSV files.
- Multivalue values are `NULL` if they are empty, defined as `<NULL>` or `\N` (see section 10.3.2, page 93).

For example, the following import file provides values for a multivalue as second column:

```
# lineno;multivalue
1;81,82,83,84,85
2;
3;0
4;1
5;1,2
6;3,4,5
7;6,7,8,9,10
8;
```

Note that when using a comma as column separator (or any other character that might be part of the multivalue elements), you can import multivalues only if they are quoted as a whole. For example:

```
# lineno;multivalue
1,"81,82,83,84,85"
2,""
3,"0"
4,"1"
5,"1,2"
6,"3,4,5"
7,"6,7,8,9,10"
8,""
```

Importing Bitvectors

Bitvectors are currently parsed like integers. For example, the input value `3` initializes the bitvector 00000011.

Using the CSV Importer

Starting the CSV Importer

After installing Cisco ParStream and providing a corresponding configuration, the easiest way to start the Cisco ParStream importer is to perform the following steps:

1. Set the variable `PARSTREAM_HOME`:

   ```
   export PARSTREAM_HOME=/opt/cisco/kinetic/parstream-database
   ```

   and add Cisco ParStream libraries to your library search path:

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PARSTREAM_HOME/lib
   ```

2. Go to the directory, where your Cisco ParStream database configuration is located. Usually, this should be a directory containing:
   - a sub-directory `conf` which typically contains the INI file for the basic configuration
   - a import sub-directory or any other `sourcedir` specified in the INI file of the corresponding server.

3. The INI file has to contain the minimum options that are required to run an importer:
   - a global `clusterId`
   - a global `registrationPort`
   - importer-specific `host` and `leaderElectionPort`
   - an importer-specific `rank`
   - an importer-specific `targetdir`

   Note that `clusterId`, `leaderElectionPort` and `rank` are required because any Cisco ParStream server can sooner or later be extended to a multi-node cluster (see chapter 6, page 40).

   You might typically also set:
   - a global `clusterInitTimeout`, which defines the time in seconds to wait for a cluster of multiple nodes to establish. If you know that the database is running with only one server/node you can/should set to just `1`.
   - the importer-specific `sourcedir`

   For example, an INI file for an importer called `imp` might look as follows:

   ```
   # basic cluster settings for single-node clusters:
   clusterId = exampleCluster
   registrationPort = 9040
   clusterInitTimeout = 1

   [import.imp]
   ```

```
host = localhost
leaderElectionPort = 9051

# rank (any number is fine):
rank = 99

# directory that contains the raw data (CSV files) to import:
sourcedir = ./import

# temporary directory that contains read partitions before transferred to the server:
targetdir = ./import-partitions
```

4. Start the Cisco ParStream importer `parstream-import`. The only required argument is the name of the importer to start, which identifies the corresponding section in the INI Files for the other setting (you can pass this name also with `--importername`).

   For example:

   ```
   $PARSTREAM_HOME/bin/parstream-import imp
   ```

If you add $PARSTREAM_HOME/bin to your search path, starting the importer as follows is enough:

```
parstream-import imp
```

By default, the importer reads for each table the files that match the specified subdirectory and file pattern. For example, here for table `MyTable` all files ending with `.csv` are processed:

```
CREATE TABLE MyTable (
  ...
)
IMPORT_DIRECTORY_PATTERN '.*'
IMPORT_FILE_PATTERN '.*\.csv';
```

The default is to look for all files that begin with the name of the table and end with `.csv`. Thus, with

```
CREATE TABLE Hotel (
  ...
);
```

the importer will read all CSV files, where the file name starts with `Hotel`.

By default, the importer runs in an endless loop. Thus, it does not end when all files in the import directory are processed. Instead, it checks again and again whether new files for import exist and processes them.

Note that CSV import files provided by some processes should in general have **different names**. If you just write CSV files using the same name, ensure a file was processed before providing a new file. For this, it is not enough to find the imported data in the database, you also should have the imported file in the backup directory (see section 10.5.2, page 101).

The interval to check for new import files is something to configure as global entry in the configuration file (see section 13.2, page 119):

```
# Time to wait for the importer between searches for new data (in seconds)
reimportInterval = 1
```

When importing data, the data gets parsed and transferred into partitions, which are stored then in a target directory. This can bet set via the option `targetdir`:

```
[import.imp]
targetdir = ./import-partitions
```

For more sophisticated ways to start a Cisco ParStream importer (e.g. logging all output into a file), see the shell script `import.sh` in `$PARSTREAM_HOME/bin`.

Note:

If an importer has to connect to a server, then the port specified in the server configuration and the port above it must be open. If e.g. port 4002 is configured, then both the ports 4002 and 4003 must be available. See section 13.3.1, page 135 for details.

.backup and .rejected folders

Imported CSV files are by default moved into the backup directory of the import directory. For each import file, the backup directory is the sub-directory `.backup`.

You can disable the move of the imported files with the importer option `csvreadonly` (see section 13.4.2, page 146). This, of course, only makes sense, if the importer is not running in a loop. Note that you can't define an alternative directory for the backup folder, because if the backup folder is on a different device, this is a significant performance drawback. Use symbolic links instead.

Because CSV files can contain about everything (including illegal data), it will be filtered by the importer. Every line of the CSV file that cannot be imported into the database will be put into a file in the subdirectory `.rejected`. The name of the file is `<`*`csvfilename`*`>.rejected.txt`, where *csvfilename* is the complete CSV file name including any compression suffixes. This file may contain empty or comment lines in addition to the ones containing illegal data or a wrong number of columns. That file is never compressed, even though the imported CSV file was compressed.

Importer and Schema/Metadata Changes

Normally the CSV importer periodically checks for schema/metadata changes i.e. for a `CREATE TABLE` or `ALTER TABLE` statement issued to the server (see chapter 24, page 277). Importing for a newly created table works automatically, but for an `ALTER TABLE` to work seamlessly you need to do the following steps:

1. Stop creating CSV-files
2. Make sure the CSV importer runs out of CSV files

3. Issue `ALTER TABLE` command(s)
4. Provide CSV-files in new format

CSV Importer Command Line Parameters

The Cisco ParStream CSV importer provides the ability to pass several commandline arguments, to configure and influence the behavior of the importer.

On one hand you can pass several general arguments. For example:

Option	Arg	Description	Default
`--help`		Prints how to start the Cisco ParStream server/importer and exits	
`--version`		Prints the version of the Cisco ParStream server/importer and exits	
`--confdir`	string	Defines the configuration directory, where INI files are searched	./conf
`--configcheck`		Check configuration (INI file and commandline options) and end the program	
`--importername`	string	Defines the name of the importer to start	servername
`--inifile`	string	Constrains to only one configuration (INI) file being used	
`--servername`	string	Defines the name of the server to start	importername
`--sourcedir`	string	overwrites a sourcedir specified in the INI files (see Import-Options)	
`--targetdir`	string	overwrites a targetdir specified in the INI files (see Import-Options)	
`--verbosity`	integer	change verbosity (0: minimum, 5: maximum)	0
`--iterations`	integer	Set maximum number of iterations before ending the importer (0: endless)	0 (endless)
`--finite`		Start the import with exactly one iteration, which should import all CSV files. Sets the default for the number of iterations to 1 and the default for maximum number of CSV files processed to "unlimited".	false

In addition, you can pass some table specific arguments. For example:

Option	Arg	Description
`--table.`*tabname*`.directorypattern`	string	overwrites the `IMPORT_DIRECTORY_PATTERN` for import of table *tabname*.
`--table.`*tabname*`.filepattern`	string	overwrites the `IMPORT_FILE_PATTERN` for import of table *tabname*.
`--table.`*tabname*`.etl`	string	overwrites the `ETL` query for an import of table *tabname*.

See section 13.1, page 116 for a complete list of all commandline arguments.

Finally, all values of the configuration INI file can be overwritten by command line parameters. The prefix of a parameter string depends on the section, in which a specific parameter was defined.

For example, to overwrite the parameter `rank` of the server with the name imp with the value 999, the command line parameter must be formulated as follows:

```
--import.imp.rank=999
```

ETL Import

By default, each column in a CSV file results in exactly one column in the database. However, you can use so called ETL statements (“extract”, “transform”, and “load”) to transform the data you read from CSV files into the format of the database columns. Typical applications of this features are:

- Adding additional columns for better partitioning
- Filtering rows read from CSV files according to a WHERE clause

Note that ETL import transformations don’t apply to INSERT INTO statements (see section 10.7, page 107). Instead to have to provide the corresponding ETL values with an INSERT INTO statement.

To define a ETL import transformation, you need to provide some additional information when defining the table:

- In the corresponding `CREATE TABLE` statement you must define a corresponding ETL statement.
- In addition, columns additionally filled by the ETL statement must have been declared with `CSV_COLUMN ETL`.

Note also that you can specify the ETL import statement via the command line with option `--table.`*tabname*`.etl` (see section 10.5.4, page 102).

Additional ETL Columns

For example, a ETL clause such as

```
ETL ( SELECT id MOD 3 AS etlColumn1
             FROM CSVFETCH(MyTable)
)
```

means that the resulting import will be all columns from table `MyTable` plus a column `etlColumn1` becoming the value from column `id` consisting of values between 0 and 2, computed as modulo on the column `id` in the same input row.

The keyword `CSVFETCH` instructs the importer to use the CSV import file as input. Its argument is the name of the table, the ETL statement applies to.

It’s also possible to specify other column names or wildcards in the SELECT statement, but that’s not necessary or useful because:

- A wildcard representing “all remaining columns” is implicitly added to the ETL statement anyway, having the same effect as:

  ```
  ETL ( SELECT id MOD 3 AS etlColumn1, *
               FROM CSVFETCH(MyTable)
  )
  ```

- Specifying only a fixed list of names would become a problem if later on additional columns are added to the table.

The columns filled by ETL statements should usually should be defined in the `CREATE TABLE` statement with `CSV_COLUMN ETL`:

```
CREATE TABLE MyTable
(
  id ...
  etlColumn1 ... CSV_COLUMN ETL,
)
```

Strictly speaking, `CSV_COLUMN ETL` means that the column does not have a CSV index (as it is the default for `CSV_COLUMN`, see section 24.2.4, page 284), which means that this column doesn't get values from corresponding CSV import files.

Modifying Existing Columns

You can even have ETL import conversions for columns that are not marked with `CSV_COLUMN ETL`. Then, there must be a value in the import file, which is overwritten then during the import according to the ETL statement.

For example, the following statement ensures that all values of a `VARSTRING` column `valString` only have lowercase letters:

```
ETL (SELECT LOWER(valString) AS valString FROM CSVFETCH(MyTable))
```

Filtering CSV Rows

To filter data read out of CSV files, you simple have to provide a WHERE clause in the ETL statement. All rows, where the WHERE clause does not match, are ignored. Note that filtered lines will neither be found in the .rejected folder nor in logfiles.

For example:

```
ETL ( SELECT * FROM CSVFETCH(etlTable)
             WHERE last_access > date '2013-01-31'
)
```

imports all lines from CSV files for table `etlTable` where the date read from the column `last_access` is after 01/31/2013. All rows not matching this definition (this includes rows where `last_access` is `NULL` in this example) will be ignored.

A Complete Example for an ETL Import

The following table definition:

```
CREATE TABLE etlTable
(
  id UINT32 INDEX EQUAL,
  street VARSTRING (100),
  zip UINT32 INDEX EQUAL,
  city VARSTRING (100) COMPRESSION HASH64 INDEX EQUAL,
  -- an artificial column created by the ETL process as "id MOD 3":
```

```
  partitionId UINT32 INDEX EQUAL CSV_COLUMN ETL,
)
PARTITION BY partitionId
ETL ( SELECT id MOD 3 AS partitionId,
             FROM CSVFETCH(etlTable)
             WHERE city = 'San Francisco'
);
```

reads input rows with columns `id`, `street`, `zip`, and `city`, ignoring rows where the city is not “San Francisco,” and adds an additional column `partitionId` to partition the data according to the ID’s read.

For other examples, see the section about table distribution (section 6.3, page 53) or the Separation Aware Execution optimization (section 15.15.1, page 188).

Import Data with INSERT INTO

An existing table can be filled by an INSERT INTO statement. The INSERT INTO statement has the following format (for grammar details see section 27.4, page 360):

```
INSERT INTO <table-name> <select-statement>
```

Note the following:

- You have to provide values for *all* columns of the destination table, except those marked with `SKIP TRUE`. This includes values for ETL columns (see below).
- The columns are identified by their name, not by the order. For this reason you usually need aliases in the SELECT part of INSERT INTO statements (unless you read data from a source table having the same column name(s)).
- For each import you need an available thread. Thus, for each node/server you have to specify with option `maxImportThreads` (see section 13.2.1, page 127) how many imports are possible in parallel (which might also require to set `socketHandlingThreads`, see section 13.3.2, page 138).

Also, note the limitations below.

Example

Consider we have the following two tables:

```
CREATE TABLE Hotels
(
  city VARSTRING (1024) COMPRESSION HASH64 INDEX EQUAL,
  hotel VARSTRING (100) COMPRESSION HASH64 INDEX EQUAL,
  price UINT16 INDEX RANGE,
  "week" UINT8
)
...

CREATE TABLE AveragePrices
(
  city VARSTRING (1024) COMPRESSION HASH64 INDEX EQUAL,
  price UINT16 INDEX RANGE,
  quarter UINT8 INDEX EQUAL
)
...
```

After inserting data into table `Hotels` you can fill the table for the average prices for example as follows:

```
INSERT INTO AveragePrices
            SELECT city, SUM(price)/COUNT(*) AS price, 1 AS quarter
                   FROM Hotels
                   WHERE week BETWEEN 1 AND 13
```

```
                              GROUP BY city;
```

If this command is successful, it returns `INSERT`, a `0` and the number of successfully inserted rows. For example:

```
#OK
INSERT 0 64
```

Note that to insert NULL for specific columns you can't pass NULL for a hashed string column. Instead, you have to pass the empty string. For example, the following is possible (provided `street` is a non-hashed string and `city` is a hashed string):

```
INSERT INTO Address SELECT 777 AS id, NULL AS street, NULL AS zip, '' AS city;
```

Note however, that in general using INSERT INTO to insert single rows is **not** a good idea, because for each row a new partition is created. (This is the reason, why Cisco ParStream provides a INSERT INTO statement for results of queries but not for single rows.)

Limitations

Note the following limitations:

- INSERT INTO statements writes data directly into the target table without any ETL transformation (see section 10.1, page 88). For this reason, **you also have to provide the values for ETL columns.** Of course, you have to ensure that the integrity of ETL columns is not violated by this. That is, either the values for ETL columns already match the ETL statement or you have to apply the same function to them.
- When inserting data, the column types have to match. This especially means that you can't insert a column of hashed strings into a column of non-hashed strings or vice versa.
- Currently, no parentheses are allowed around the inner SELECT statement.
- System tables `ps_info_import` and `ps_info_query_history` are only partly supported yet.

Deleting Data

Cisco ParStream supports deleting data from a table using standard SQL commands. The possibilities and limitations of the delete command are introduced and explained in this chapter.

Delete Statements

Cisco ParStream supports standard SQL `DELETE` statements. The data is removed by evaluating the filter provided in the `DELETE` statement for each record. If a record matches this filter, the records will be removed from the system. Once the records are removed from the table, the statement will return with the number of deleted records.

For example, a query deleting every record with the value '42' in column 'value' of a table 'measurements' would look like this:

```
DELETE FROM measurements WHERE value = 42;
```

If the filter is omitted from the query, all data of the table is deleted:

```
DELETE FROM measurements;
```

Limitations

Please note the following general limitations of `DELETE` statements in Cisco ParStream:

- Filter statements in `WHERE` clauses are only allowed on partitioning columns. The consequence of this limitation is that only complete partitions can be deleted. If a filter contains any non-partitioning columns, an error will be reported.
- In the case of parallel `IMPORT` and `DELETE` operations, where the filter of the `DELETE` statement matches data of the `IMPORT` statement, it is undefined whether the newly imported data is deleted or not. The only guarantee is that the cluster retains a consistent view of the data.
- `DELETE` statements and merges cannot run in parallel. If a merge is already running on the table specified in the DELETE statement, the `DELETE` will wait for the merge to complete. Hence, the DELETE statement can take a long time to complete if a merge is running in parallel.
  Any merge on a table will be postponed until the `DELETE` statement on the table has finished.

Client Applications and Tools

This chapter describes the client applications and tools of Cisco ParStream:

- The socket interfaces `pnc` and `netcat/nc` (see section 12.1, page 110)
- The PSQL Client (see section 12.2, page 115)

Database Clients `pnc` and `netcat`

To connect to a Cisco ParStream server, you can use any client that can serve as socket interface (see chapter 16, page 199). A typical example is `netcat`, but Cisco ParStream provides a more convenient tool called `pnc`, which is described first.

`pnc`

Cisco ParStream provides a python client, called `pnc`, as a more comfortable alternative to working directly with `netcat`. It's features resemble that of a PQSL client but `pnc` connects to Cisco ParStream via the socket interface. After installation, `pnc` is located at `$PARSTREAM_HOME/bin`.

Features of `pnc`

`pnc` provides some advantages about simple socket interfaces such as `netcat`:

- Multi-line queries are supported
  - Queries cover multiple input lines until a semicolon is reached
  - Comment lines, starting with `--` are ignored
- Seeing responses without additional options.
- Measuring execution time.
- Options to execute queries from a file.
- Options to write results to a file.

Note that Cisco ParStream servers currently only accept single-line commands. `pnc` converts multi-line commands in one single-line command, which especially makes it easier to process complex commands such as CREATE TABLE commands read from batch files (SQL files).

Invocation of `pnc`

The invocation corresponds to PSQL.

For example:

```
pnc -h <host> -p <port> -U <loginname> --auto-reconnect --ssl
```

`pnc` has the following options:

Option	Short	Effect
`--host`	`-h`	Hostname to connect to (default: `localhost`)
`--port`	`-p`	Port to connect to (default: port 9042)
`--username`	`-U`	Login name to use as defined by the `CREATE USER` command (see section 9.2.3, page 80). If you use this option, you will be prompted for a pass phrase. Without this option, `pnc` assumes that your server does not use authentication, which by default is not the case.
`--auto-reconnect`	`-r`	With this option `pnc` transparently reconnects to the Cisco ParStream server when it loses the connection.
`--ssl`	`-s`	pnc creates an ssl encrypted connection.
`--help`	`-h`	help

All the command-line arguments are optional.

Performing Queries with `pnc`

By default, `pnc` reads command lines line by line with `Cisco ParStream=>` as prompt. Commands have to end with `;`. Lines, started with `--` are ignored.

Thus after starting `pnc` (here connecting to port 9042):

```
$ pnc -p 9042
```

the tool will establish a connection (if possible) and ask for the commands using its prompt:

```
Connecting to localhost:9042 ...
Connection established.
Encoding:  ASCII
Cisco ParStream=>
```

The you can type any command including standard SQL queries, CREATE TABLE commands, and the additional commands described in section 16.4, page 200.

For example, you can get the list of columns in the database by querying a system table typing at the `pnc` prompt the following:

```
Cisco ParStream=> SELECT table_name, column_name FROM ps_info_column;
```

This results into the following output:

```
#table_name;column_name
"MyTable";"name"
"MyTable";"value"
...
[ 0.241 s]
```

The end of the request usually is the time the request took.

To exit the pnc utility, either press Ctrl-D or use the quit command (see section 16.4.2, page 201):

```
Cisco ParStream=> quit;
```

which results in the following output:

```
Lost connection.
```

Performing Scripts with pnc

You can also use pnc to read commands from standard input. For example, the following command performs a SELECT command passed with echo on a Cisco ParStream server listening on port 9988 of the local host:

```
echo 'SELECT * FROM ps_info_column;' | pnc -p 9988 -U loginname
```

This can also be used to send the contents of full command/batch files, such as SQL files with CREATE TABLE commands:

```
pnc -p 9988 -U loginname < MyTable.sql
```

pnc converts the commands read into single-line commands, while removing comment lines. For example, if the contents of MyTable.sql is the following multi-line command:

```
-- create a first table
CREATE TABLE MyTable
(
  col UINT32
);
```

this command is sent as the following single-line command to Cisco ParStream:

```
CREATE TABLE MyTable ( col UINT32 )
```

This is what you'd have to send to Cisco ParStream servers if a standard command line tool such as netcat is used (see section 12.1.2, page 114).

Note that *inside* a command no comments should be used. You should place them before or after the commands.

Key Combinations and Commands of pnc

pnc allows the following key combinations:

Key (Combination)	Effect
`<ENTER>`	newline
`; <ENTER>`	execute SQL query
`<CTRL>+<C>`	kill

`pnc` provides the following commands (no “`;`” is required):

Command	Effect
`\q <ENTER>`	quit
`\t <ENTER>`	toggle timing (timing is enabled at startup)
`\o output.csv <ENTER>`	write all output to the file `output.csv`
`\d <ENTER>`	describe: List of all tables
`\d mytable <ENTER>`	describe: List columns for table `mytable`

netcat / nc

`netcat` or `nc` is a standard tool you can use to send statements to Cisco ParStream servers.

Note however: If using `netcat` all authorization credentials (login name and pass phrase) should be passed as environment variables to minimize exposure of credentials (login name and pass phrase) to tools like `top`, `ps` and bind the lifetime of these secrets to the session. A common useful rule is to `export HISTCONTROL=ignorespace` and then define these environment variable values in a command line starting with a space, so it will not be written into command line history. It is alternatively possible to use **`pnc`** instead of `netcat`. With `pnc` pass phrase input hides typed pass phrases. In addition, `pnc` provides some additional features. See section 12.1.1, page 110 for details about `pnc`.

Performing Queries with netcat

After connecting with `netcat`, you enter queries followed by return. The results are displayed inline.

The first command usually is the login request, because by default Cisco ParStream requires user authentication (see section 9.2, page 78):

```
LOGIN 'username' 'pass phrase'
```

The server should respond with:

```
#INFO-77060: Authentication successful
```

The login lasts until the end of your connection to the Cisco ParStream server.

Then you can send any SQL command, which is terminated by a newline. For example:

```
SELECT table_name, column_name, column_type, column_size FROM ps_info_column
```

The command might might produce the following output:

```
#table_name;column_name;column_type;column_size
"Address";"year";"numeric";"UINT64";<NULL>
"Address";"month";"numeric";"UINT64";<NULL>
```

```
"Address";"day";"numeric";"UINT64";<NULL>
"Address";"state";"string";"VARSTRING";100
...
```

You can use all SQL commands as well as the additional command described in section 16.4, page 200.

To end the connection to the server, you can send the quit command (see section 16.4.2, page 201):

```
quit
```

Note again that each statement has to end with a newline.

Performing Scripts with netcat

You can pipe input into the standard netcat, which might come from other commands or files. This can be used to script queries.

This way, you can send complete scripts. For example:

```
export HISTCONTROL=ignorespace username='login_name'; pw='pass phrase'
echo -e "LOGIN '${username}' '${pw}'\nSELECT * FROM ps_info_column\nquit\n" |
    nc localhost 9950 -w 10
```

Note the following:

- You have to send commands on lines that end with a newline.
- You should use an option such as `-w` *sec*, which waits the specified number of seconds and then quit, to see responses (on some platforms this option might not be available).

You can also send the content of SQL command files via netcat. For example, you can create create tables that way:

```
cat MyTable.sql | grep -v '^--' | tr '\n' ' ' | tr ';' '\n' | netcat
    localhost 9077
```

Note that you have to

- remove comment lines because they are currently not correctly ignored
- transform newlines into spaces because the input has to be one line
- transform the ending semicolon into a newline because commands have to end with a newline

Again note that usually user authentication is required, which makes the command more complicated or part of the SQL file. By using pnc instead of netcat/nc performing commands from SQL/batch files is a lot easier (see section 12.1.1, page 112).

PSQL client

To use the PSQL Postgres client in conjunction with Cisco ParStream, install package `postgresql` of your Linux distribution. Then enter at the command prompt (a simple example only):

```
psql -h localhost -p 1112 -W "sslmode=disable" username
```

Note that one can use any pass phrase, but it may not be empty.

If the user authentication is disabled (see section 9.2, page 78), you can provide a dummy password to the command. Then enter at the command prompt:

```
psql -h localhost -p 1112 "sslmode=disable password=x"
```

The connect parameters can also be stored in environment variables (necessary for "sslmode" and "password" in older versions of psql):

```
export PGHOST=localhost
export PGPORT=1112
export PGSSLMODE=disable
export PGPASSWORD=x
export PGUSER=username
```

This is the preferred way of submitting credentials as other users may see the issued command line, e.g. via `ps`.

Note that within PSQL, a command must be terminated by a semicolon ";" before it is executed when pressing the return key.

To quit type “`\q`”.

Here are the most important commands:

Command	**Effect**
`\q`	quit
`\r`	reset (clear current query)
`\h` *command*	help
`\timing`	enable timing queries

For further information, see `man psql` or:

http://www.postgresonline.com/special_feature.php?sf_name=postgresql83_psql_cheatsheet

Options Reference

This chapter describes the details of all options you can set for Cisco ParStream servers and importers. It describes the commandline options as well as the options you can set via INI files (or via the commandline).

Commandline Arguments

For both importer and server, you can specify a couple of commandline options.

Pure Commandline Options

On one hand, there are the pure commandline options listed in Table 13.1 that only can be passed via the commandline (in the **For** column, “S” means that the option applies to servers and “I” means that this option applies to importers).

Note the following:

- Instead of `etlMergeHour`, `etlMergeDay`, `etlMergeWeek`, `etlMergeMonth` you can also pass the options `etlMergeLevel1`, `etlMergeLevel2`, `etlMergeLevel3`, `etlMergeLevel4`, respectively.

If you pass a positional parameter that is not an argument of an option, it is used as servername by the server and importername by the importer.

For example, the following three calls are equivalent for starting a server `first`:

```
parstream-server --confdir MyProject/conf first

parstream-server --confdir MyProject/conf --servername=first

parstream-server --confdir MyProject/conf --servername first
```

Passing INI-File Options as Commandline Options

In addition, you can pass all INI file options as commandline arguments. This applies to global options (see section 13.2, page 119) as well as section-specific options. Note that in fact a couple of global options are typically passed as commandline options.

For example, the following command starts the importer `first` with a verbosity level of 3:

```
parstream-import --verbosity=3 first
```

Here, the verbosity argument of the commandline overwrites the global INI file entry for option `verbosity` (if specified):

```
verbosity = 0
```

As another example, the following command starts the server `first` with the port passed via command line:

```
parstream-server first --server.first.port=1234
```

Thus, the port argument of the commandline overwrites the following INI file entry (if specified):

```
[server.first]
port = 2345
```

Importer Commandline Options

For the importer, the following options can be defined as commandline arguments:

Option	Arg.	Description	Default
`--verbosity`	integer	Controls the output verbosity (0: min, 5: max)	false
`--iterations`	integer	Set maximum number of iterations before ending the importer	(0: endless)
`--finite`		Start the import with exactly one iteration, which should import all CSV files. Sets the default for the number of iterations to 1 and the default for maximum number of CSV files processed to "unlimited".	false

It is usually not necessary to specify an option because the option `clusterId` is usually set in the INI file.

The default for `iterations` is 0 (endless).

Option	Arg.	Description	Default	For
`--help`		Prints how to start the Cisco ParStream server/importer and exits		S/I
`--version`		Prints the version of the Cisco ParStream server/importer and exits		S/I
`--confdir`	string	Defines the configuration directory, where INI files are searched	`./conf`	S/I
`--configcheck`		Check configuration (INI file and commandline options) and end the program		S/I
`--inifile`	string	Constrains to only one configuration (INI) file being used		S/I
`--servername`	string	Defines the name of the server to start/use	importername	S/I
`--importername`	string	Defines the name of the importer to start	servername	I
`--sourcedir`	string	overwrites a sourcedir specified in the INI files (see section 13.4, page 146)		I
`--targetdir`	string	overwrites a targetdir specified in the INI files (see section 13.4, page 146)		I
`--table.`*tab*`.directorypattern`	string	overwrites the `IMPORT_DIRECTORY_PATTERN` for imports of table *tab*.		I
`--table.`*tab*`.filepattern`	string	overwrites the `IMPORT_FILE_PATTERN` for imports of table *tab*.		I
`--table.`*tab*`.etl`	string	overwrites the `ETL` query for imports of table *tab*.		I
`--table.`*tab*`.etlMergeMinute`	string	overwrites the `ETLMERGE` clause for an “minute” ETL merge of table *tab*.		S
`--table.`*tab*`.etlMergeHour`	string	overwrites the `ETLMERGE` clause for an “hourly” ETL merge of table *tab*.		S
`--table.`*tab*`.etlMergeDay`	string	overwrites the `ETLMERGE` clause for an “daily” ETL merge of table *tab*.		S
`--table.`*tab*`.etlMergeWeek`	string	overwrites the `ETLMERGE` clause for an “weekly” ETL merge of table *tab*.		S
`--table.`*tab*`.etlMergeMonth`	string	overwrites the `ETLMERGE` clause for an “monthly” ETL merge of table *tab*.		S

Table 13.1: *Pure Commandline Options for Servers (“S”) and Importers (“I”)*

Global Options

Global options are defined at the beginning of the configuration file "parstream.ini". These settings are thus not located within any section. They might apply to the server or the importer or both.

Note that you can also pass these options alternatively via the command line. For example:

```
parstream-server --locale=C first
```

would set (or overwrite) the setting of the global option `locale`.

Global Options in Detail

Cisco ParStream provides many global options so that they are presented in multiple tables:

- Mandatory global options
- Functional global options
- Non-functional global options
- Functional global options for multi-node clusters
- Nonfunctional global options for multi-node clusters

Again, in the **For** column, “S” means that the option applies to servers and “I” means that this option applies to importers. `fileBlockTransferTimeout` also has an effect on partition swaps.

Mandatory Global Options

The following options have to be set for each Cisco ParStream database (either in an INI file or by passing them as command-line arguments).

Option:	**`clusterId`**
Type:	string
Default:	
Effect:	Defines the unique ID, which is used by all servers and importers to identify the cluster.
Affects:	S/I

Option:	**`registrationPort`**
Type:	integer
Default:	
Effect:	TCP port number used for the registration of cluster nodes and exchanging status information with the cluster leader. All servers and importers in a cluster have to use the same registration port.
Affects:	S/I

Functional Global Options

Option:	**`verbosity`**
Type:	integer
Default:	0
Effect:	Controls the output verbosity (0: min, 5: max). See section 9.5.3, page 85 for details.
Affects:	S/I

Option:	**`defaultDebugLevel`**
Type:	integer
Default:	0
Effect:	Debugging level (0: off, 1: min, 5:max). You can change the value for a running server with an `ALTER SYSTEM SET DebugLevel.defaultDebugLevel=`*`value`* command (see section 27.11.1, page 375). See section 9.5.4, page 86 for details.
Affects:	S

Option:	**`debugMessageTimestamp`**
Type:	Boolean
Default:	true
Effect:	Format DEBUG messages with an preceding timestamp similar to PROT, WARN and INFO messages.
Affects:	S/I

Option:	**`reimportInterval`**
Type:	integer
Default:	6
Effect:	Seconds to wait until the importer tries to import again data from the specified sourcedir (`>= 0`).
Affects:	I

Option:	**`limitQueryRuntime`**
Type:	integer
Default:	0
Effect:	Interrupt queries running for longer than this time in milliseconds and return an error, to help save server resources. A value of 0 disables the time limit. You can overwrite this value for each session (see section 27.10.1, page 373).
Affects:	S

Option:	**`numBufferedRows`**
Type:	integer
Default:	32
Effect:	Size of the buffer for output. You can overwrite this value for each session (see section 27.10.1, page 373).
Affects:	S

Option:	**`userAuthentication`**
Type:	Boolean
Default:	true
Effect:	Enables/disables user authentication (see section 9.2, page 78).
Affects:	S/I

Option:	**`locale`**
Type:	string
Default:	C
Effect:	Locale for string sorting. Use the shell command `locale -a` to get a full list of supported locales on your box (see section 8.2, page 74)
Affects:	S

Option:	**`ignoreBrokenPartitions`**
Type:	bool
Default:	false
Effect:	Ignore broken partitions resulting from power outages during start up. If a broken partition is detected, the system will try to restore the partition from another cluster node. If no cluster node holds an intact copy of the partition, the partition will be dropped.
Affects:	S

Option:	**`validatePartitions`**
Type:	bool
Default:	false
Effect:	Detect broken partitions resulting from power outages during start up. This might slow down the node start up.
Affects:	S

Option:	**`iterations`**
Type:	integer
Default:	0 (endless)
Effect:	Set maximum number of iterations before ending (0: endless)
Affects:	I

Option:	**`outputformat`**
Type:	string
Default:	ASCII
Effect:	Set the default output format for queries (`ASCII`, `JSON`, `XML`). You can overwrite this value for each session (see section 27.10.1, page 373). See section 16.3, page 199 for details about the different output formats.
Affects:	S

Option:	**`asciiOutputColumnSeparator`**
Type:	string
Default:	;
Effect:	Set the string that separates columns in the `ASCII` output format. You can overwrite this value for each session (see section 27.10.1, page 373). See section 16.3, page 199 for details about the different output formats.
Affects:	S

Option:	**`asciiOutputMultiValueSeparator`**
Type:	string
Default:	,
Effect:	Set the string that separates multivalue entries in the `ASCII` output format. You can overwrite this value for each session (see section 27.10.1, page 373). See section 16.3, page 199 for details about the different output formats.
Affects:	S

Option:	**`asciiOutputNullRepresentation`**
Type:	string
Default:	<NULL>
Effect:	Set the string that represents NULL values in the `ASCII` output format. You can overwrite this value for each session (see section 27.10.1, page 373). See section 16.3, page 199 for details about the different output formats.
Affects:	S

Option:	**`journaldir`**
Type:	string
Default:	./journals
Effect:	path for journal files (see section 6.2.4, page 49)
Affects:	S/I

Option:	**`udfLibraryPath`**
Type:	string
Default:	`./udf`
Effect:	Basic path for user-defined functionality. May be a relative or absolute path. Currently, it is used: • For external user-defined table operators (xUDTO) in a way that scripts for xUDTO functionality have to be placed here (see section 20.2.2, page 234).
Affects:	S

Option:	**`enableImport`**
Type:	boolean
Default:	true
Effect:	Initially enable or disable imports. See section 16.4.1, page 200 for how to enable and disable imports in a cluster at runtime.
Affects:	S

Option:	**`enableMerge`**
Type:	boolean
Default:	true
Effect:	Initially enable or disable partition merging. The option was formerly called just **`merge`** accepting boolean values and the values `node` and `reimport`, which is still supported to use (See section 14.1.1, page 152 for details). See section 16.4.1, page 201 for how to enable and disable merges in a cluster at runtime.
Affects:	S/I

Option:	**`minutemergeschedule`**
Type:	string
Default:	`0 * * * *` (each full minute)
Effect:	The times at which the merging of seconds- to minute-partitions should be performed. Note that the value is interpreted according to UTC timestamps. See section 14.1.2, page 154. Note also that seconds-partitions are only created for streaming imports.
Affects:	S/I

Option:	**`hourlymergeschedule`**
Type:	string
Default:	`0 0 * * *` (each full hour)
Effect:	The times at which the merging of minute-partitions to hour-partitions should be performed. Note that the value is interpreted according to UTC timestamps. See section 14.1.2, page 154.
Affects:	S/I

Option:	**`dailymergeschedule`**
Type:	string
Default:	`0 0 0 * *` (each midnight UTC)
Effect:	The times at which the merging of hour-partitions to day-partitions should be performed. Note that the value is interpreted according to UTC timestamps. See section 14.1.2, page 154.
Affects:	S/I

Option:	**`weeklymergeschedule`**
Type:	string
Default:	disabled
Effect:	The times at which the merging of day-partitions to week-partitions should be performed. Note that the value is interpreted according to UTC timestamps. See section 14.1.2, page 154. If not explicitly configured, the merge of week-partitions is disabled.
Affects:	S

Option:	**`monthlymergeschedule`**
Type:	string
Default:	disabled
Effect:	The times at which the merging of week-partitions to month-partitions should be performed. Note that the value is interpreted according to UTC timestamps. See section 14.1.2, page 154. If not explicitly configured, the merge of month-partitions is disabled.
Affects:	S

Option:	**`highResolutionLogtime`**
Type:	Boolean
Default:	true
Effect:	Format timestamps of DEBUG, PROT, WARN, and INFO messages with fractional seconds.
Affects:	S/I

Option:	**`blobbuffersize`**
Type:	integer
Default:	1,048,576 (2^{20})
Effect:	Maximum number of bytes a single blob and string value may contain. Rows that contain longer values are not imported. The value must be >= 64. The value is automatically adjusted if this is not the case. If this value is set, it might also make sense to set the import option `inputFileBufferSize` (see section 13.4.3, page 148).
Affects:	S/I

Option:	**`ipVersion`**
Type:	string
Default:	IPv4
Effect:	Specifies the accepted IP protocols for network communication. Possible options are `IPv4`, `IPDualStack`, and `IPv6`. `IPv4` will use IPv4 addresses only. The incoming accept port is bound to all IPv4 interfaces. Hostnames in the cluster configuration will be resolved to IPv4 addresses only. IPv6 addresses are not allowed in this configuration. For outgoing connections from one cluster member or the importer to other cluster members, only IPv4 connections are used. Clients may only connect using IPv4 connections. `IPDualStack` allows for usage of both types of IP addresses, but IPv6 has to be configured on all hosts running a server process. The incoming accept port is bound to all interfaces, allowing IPv4 and IPv6 connections. Hostnames in the cluster configuration will be resolved to IPv4 and IPv6. Both types of IP addresses are allowed in the configuration. For outgoing connections from one cluster member or the importer to other cluster members, IPv4 and IPv6 connections will be used (no preference of the IP version). Clients may connect using IPv4 and IPv6 addresses. `IPv6` will use IPv6 addresses only. The incoming accept port is bound to all IPv6 interfaces. Hostnames in the cluster configuration will be resolved to IPv6 addresses only. IPv4 addresses are not allowed in the configuration. For outgoing connections from one cluster member or the importer to other cluster members, only IPv6 connections are used. Clients may only connect using IPv6 connections.
Affects:	S/I

Option:	**`sslMode`**
Type:	string
Default:	none
Effect:	Enable TLS encryption for network connections. The available options include: - none: no connections will use TLS encryption - client: all database clients will communicate over SSL/TLS encrypted connections - server: all internal cluster connections will communicate over SSL/TLS encrypted connections - all: All connections will communicate over SSL/TLS encrypted connections Note: Only TLS encryption is supported as SSL is deprecated.
Affects:	S/I

Option:	**`sslKeyFile`**
Type:	string
Default:	conf/server.key
Effect:	Private key file of the server used for SSL/TLS encryption.
Affects:	S/I

Option:	**`sslCertFile`**
Type:	string
Default:	conf/server.crt
Effect:	Certificate file of the server used for SSL/TLS encryption.
Affects:	S/I

Option:	**`sslCaFile`**
Type:	string
Default:	
Effect:	Trusted certificate authorities file of the server. The option will be ignored if it is empty.
Affects:	S/I

Option:	**`sslDhFile`**
Type:	string
Default:	
Effect:	Diffie-Hellman parameter file of the server. The option will be ignored if it is empty.
Affects:	S/I

Option:	**`sslCiphers`**
Type:	string
Default:	HIGH:!aNULL
Effect:	List of SSL/TLS ciphers accepted by the server.
Affects:	S/I

Option:	**`sslMinimumTlsVersion`**
Type:	string
Default:	1.2
Effect:	Minimum version of TLS to use for the encryption, e.g., 1.2, 1.1, or 1.0.
Affects:	S/I

Option:	**`overrideProcessRequirements`**
Type:	Boolean
Default:	false
Effect:	Override the requirements of mapped files and number of open file handles and only issue a warning instead of an error.
Affects:	S/I

Option:	**`clientConnectionTimeout`**
Type:	integer
Default:	0
Effect:	Configure the time in seconds after which an inactive connection will be dropped by the server. (0: off)
Affects:	S

Note:

- The format of the ...`schedule` merge options is described in section 14.1.2, page 154. See section 14.1, page 151 for details.

Non-Functional Global Options

Option:	**`partitionMaxRows`**
Type:	integer
Default:	10*1024*1024
Effect:	Maximum number of rows a partition can have. If the number of rows exceeds this limit, a logical partition is split up into multiple physical partitions (see section 5.1, page 29). This option also impacts whether merges are performed (see section 5.1.1, page 31).
Affects:	S/I

Option:	**`partitionMaxRowsForMinuteMerge`**
Type:	integer
Default:	`partitionMaxRows`
Effect:	Maximum number of source rows an minute merge can have (see section 14.1, page 151).
Affects:	S/I

Option:	**`partitionMaxRowsForHourlyMerge`**
Type:	integer
Default:	`partitionMaxRows`
Effect:	Maximum number of source rows an hourly merge can have (see section 14.1, page 151).
Affects:	S/I

Option:	**`partitionMaxRowsForDailyMerge`**
Type:	integer
Default:	`partitionMaxRows`
Effect:	Maximum number of source rows a daily merge can have (see section 14.1, page 151).
Affects:	S/I

Option:	**`partitionMaxRowsForWeeklyMerge`**
Type:	integer
Default:	`partitionMaxRows`
Effect:	Maximum number of source rows a weekly merge can have (see section 14.1, page 151).
Affects:	S/I

Option:	**`partitionMaxRowsForMonthlyMerge`**
Type:	integer
Default:	`partitionMaxRows`
Effect:	Maximum number of source rows a monthly merge can have (see section 14.1, page 151).
Affects:	S/I

Option:	**`maxExecutionThreads`**
Type:	integer
Default:	1.5 times the number of hardware threads rounded up
Effect:	Maximum number of threads to use for execution in total (must be set to a value `>0`, see section 15.1, page 158). Influences the default value of `maxMergeThreads` and `maxImportThreads`.
Affects:	S/I

Option:	**`maxQueryThreads`**
Type:	integer
Default:	0 (same as maxExecutionThreads)
Effect:	Maximum number of threads, out of the total `maxExecutionThreads` available, to use for queries. Value `0` means to use the value of `maxExecutionThreads`.
Affects:	S

Option:	**`maxMergeThreads`**
Type:	integer
Default:	`maxExecutionThreads` / 4, at least 1
Effect:	Maximum number of threads, out of the total `maxExecutionThreads` available, to use for merging partitions (Chapter 14, page 151). Value `0` means to use the value of `maxExecutionThreads`.
Affects:	S/I

Option:	**`maxImportThreads`**
Type:	integer
Default:	`maxExecutionThreads` / 4, at least 1
Effect:	Maximum number of threads, out of the total `maxExecutionThreads` available, to use for server-sided import jobs, such as Java Streaming Imports (Chapter 19, page 216) and `INSERT INTO`'s (section 10.7, page 107). Value `0` means to use the value of `maxExecutionThreads`.
Affects:	S/I

Option:	**`maxExternalProcesses`**
Type:	integer
Default:	`maxExecutionThreads`
Effect:	Maximum number of concurrent external processes for execution of UDT statements, see section 20.2.2, page 234.
Affects:	S/I

Option:	**`defaultQueryPriority`**
Type:	string/integer
Default:	`medium` (same as `4`)
Effect:	Execution priority of query tasks unless another value is specified via the `SET` command (see section 27.10, page 373). The value can be passed as string or as integral value (`low/8`, `medium/4`, `high/2`). In all outputs and queries the integral value is used. Can be overridden by the server section option with the same name (see section 13.3.2, page 136). See section 15.1, page 158 for documentation of the possible values.
Affects:	S

Option:	**`defaultImportPriority`**
Type:	string/integer
Default:	`medium` (same as `4`)
Effect:	Execution priority of query tasks unless another value is specified via the `SET` command (see section 27.10, page 373) for `INSERT INTO` statements or via the Java Streaming Import Interface (see section 19, page 216). The value can be passed as string or as integral value (`low/8`, `medium/4`, `high/2`). In all outputs and queries the integral value is used. Can be overridden by the server and import section option with the same name (see section 13.3.2, page 136 and see section 13.4.3, page 148). See section 15.1, page 158 for documentation of the possible values.
Affects:	S/I

Option:	**`defaultMergePriority`**
Type:	string/integer
Default:	`medium` (same as `4`)
Effect:	Execution priority of merge tasks. The value can be passed as string or as integral value (`low/8`, `medium/4`, `high/2`). In all outputs and queries the integral value is used. Can be overridden by the server and importer section option with the same name (see section 13.3.2, page 136 and see section 13.4.3, page 148). See section 15.1, page 158 for documentation of the possible values.
Affects:	S/I

Option:	**`queryThrottlingInterval`**
Type:	integer
Default:	500
Effect:	Milliseconds of thread time scaled by number of available threads of after which a query is considered “long running” and will get its effective priority reduced (section 15.1, page 158). Value `0` means the effective priority remains stable.
Affects:	S

Option:	**`fileBlockTransferTimeout`**
Type:	float
Default:	10.0
Effect:	Timeout (seconds) of one file data block transfer.
Affects:	I

Option:	**`fileBlockTransferBuffersize`**
Type:	integer
Default:	2*1024*1024
Effect:	Buffersize (bytes) for file data block.
Affects:	I

Option:	**columnStoreSegmentSize**
Type:	integer
Default:	65536
Effect:	Maximum size in bytes of segments when writing sparse or dictionary column stores (> 0). (see section 15.10.1, page 178) .
Affects:	I

Option:	**queryHistoryMaxSeconds**
Type:	integer
Default:	`600` (10 minutes)
Effect:	Maximum duration (in seconds) queries are kept for the query history system table `ps_info_query_history` (See section 26.4, page 321). `0` means that no duration limit exists.
Affects:	S

Option:	**queryHistoryMaxEntries**
Type:	integer
Default:	1000
Effect:	Maximum number of queries kept for the query history system table `ps_info_query_history` (See section 26.4, page 321). `0` means that no quantity limit exists, so that **all** queries are kept.
Affects:	S

Option:	**importHistoryMaxEntries**
Type:	integer
Default:	1000
Effect:	Maximum number of *historic* imports kept for the imports system table `ps_info_import` (See section 26.4, page 322). `0` means that the maximum is really `0`, so that **no** imports are kept at all.
Affects:	S

Option:	**performanceLoggingPath**
Type:	string
Default:	`/var/log/parstream`
Effect:	The directory into which monitoring information is logged (See section 9.5.1, page 84).
Affects:	S/I

Option:	**executionSchedulerMonitoringTimeDetailLevel**
Type:	string
Default:	`summary`
Effect:	Control the detail level of the global execution scheduler event log (See section 9.5.1, page 84). Possible values are `summary` and `intervals`, where the latter causes logging of each execution slice (ca every 20ms) and may therefore generate very large logs. Is overridden by the server option of the same name.
Affects:	S/I

Option:	**`synchronizeFilesystemWrites`**
Type:	Boolean
Default:	false
Effect:	Synchronizes file changes with the storage device, which prevents data loss in case of a power outage for merges and partition transfers. This option should be set to true if the system where the database is running on has no emergency protection like an uninterruptible power supply (UPS) and no clean shutdown of the operating system can be guaranteed. Enabling this option will slow down the operation of the database.
Affects:	S

Global Options for Multi-Node Clusters

By default, Cisco ParStream is running with clustering abilities (see chapter 6, page 40). Several additional options are provided to influence the behavior of such a cluster, which matters if it has more than one server/node.

Again, these options are presented in multiple tables:

- Functional global options for multi-node clusters
- Nonfunctional global options for multi-node clusters

Note that usually all cluster node should have the same cluster option value to ensure that when the leader changes the same policies are used.

Functional Global Options for Multi-Node Clusters

Option:	**`minUpdateRedundancy`**
Type:	integer
Default:	1
Effect:	The leader submits merges to all available nodes of a distribution group only, when the number of participating nodes is not below this value. Imports stop distributing partitions for a distribution group that has less than this number of members available. **Hint:** If you have a cluster where you want to enforce redundancies greater than 1, you have to set this value explicitly (e.g. set this value to `2` to have a redundancy of 2, having 1 backup copy of all the data; see section 6.2.3, page 46)
Affects:	S/I

Note:

- Note that currently import and merge operations are blocked for tables that declare a redundancy value smaller than this global start-time value. Note that Cisco ParStream can't fix the options automatically because the value might be useful if additional nodes will be added (see section 6.2.5, page 50).

Non-Functional Global Options Multi-Node Clusters

These options are mainly options for timers and intervals, which are used to establish the cluster, detect errors, and recover.

Option:	**`maxSchedulerConcurrencyLevel`**
Type:	integer
Default:	5
Effect:	Limits the number of asynchronous tasks a cluster node processes concurrently. The minimum value is 1.
Affects:	S/I

Option:	**`maxSyncConcurrencyLevel`**
Type:	integer
Default:	10
Effect:	Limits the number of asynchronous partition synchronisation tasks the cluster processes concurrently. The minimum value is 1.
Affects:	S/I

Option:	**`maxActivationDeletionConcurrencyLevel`**
Type:	integer
Default:	3
Effect:	Limits the number of asynchronous partition activation or delete tasks the cluster processes concurrently. The minimum value is 1.
Affects:	S/I

Option:	**`clusterInitTimeout`**
Type:	integer
Default:	120
Effect:	Time (seconds) within cluster start-up for finding all nodes in the subnet and electing the leader. Note that you have to start all nodes in the first half of this period. The minimum for `clusterInitTimeout` is 20 seconds. See section 6.2.2, page 42 for details.
Affects:	S/I

Option:	**`clusterReinitTimeout`**
Type:	integer
Default:	clusterInitTimeout
Effect:	Time (seconds) within cluster reinitialization for finding all nodes in the subnet and electing the leader if the cluster already exists. Note that you have to start all nodes in the first half of this period. The minimum for `clusterReinitTimeout` is 10 seconds. See section 6.2.2, page 42 for details.
Affects:	S/I

Option:	**`claimLeadershipMessageInterval`**
Type:	integer
Default:	5
Effect:	Interval (in Seconds) to send out registration messages during the first half of the cluster initialization. The minimum for `claimLeadershipMessageInterval` is 1 second. The maximum for `claimLeadershipMessageInterval` is `clusterInitTimeout / 8` or `clusterReinitTimeout / 8`, respectively, to have at least four claim leadership requests during the node registration period (first half of cluster (re)initialization timeout). See section 6.2.2, page 42 for details.
Affects:	S/I

Option:	**`nodeRegistrationTimeout`**
Type:	integer
Default:	30
Effect:	Timeout (seconds) for the registration and deregistration of a cluster node at the leader.
Affects:	S/I

Option:	**`nodeAliveMessageInterval`**
Type:	integer
Default:	10
Effect:	Interval (seconds) for sending alive notifications from cluster nodes to the leader.
Affects:	S/I

Option:	**`nodeResynchronizationInterval`**
Type:	integer
Default:	60
Effect:	Interval (seconds, min. 30) for checking partition synchronization backlog of active nodes for entries and initiate a resynchronization if needed.
Affects:	S

Option:	**`partitionSearchTimeout`**
Type:	integer
Default:	30
Effect:	Maximum time (seconds) the cluster leader waits for the response of a partition search request.
Affects:	S/I

Option:	**`partitionMergeRequestTimeout`**
Type:	integer
Default:	60
Effect:	Maximum time (seconds) the cluster leader waits for the response of a partition merge request.
Affects:	S/I

Option:	**`synchronizePartitionRequestTimeout`**
Type:	integer
Default:	120
Effect:	Maximum time (seconds) a cluster node waits for the response of a request to synchronize partitions between nodes. The expected processing time depends strongly on the number of columns and the size of the partitions and should be increased appropriately.
Affects:	S/I

Option:	**`requestDefaultTimeout`**
Type:	integer
Default:	60
Effect:	Generic maximum duration (seconds) a cluster node waits for the response of a request with an average expected processing time.
Affects:	S/I

Option:	**`dhsgbConnectionTimeout`**
Type:	integer
Default:	255
Effect:	Timeout (seconds) to establish inter cluster connections for DHSGB (see section 15.15.2, page 189).
Affects:	S

Authentication Options

For authentication (see section 9.2, page 78), the options described in this section are provided under the section `authentication`. For example:

```
[authentication]
pamService = parstream
authenticationWrapperExecutable =
    /opt/cisco/kinetic/parstream_authentication_1/parstream-authentication
```

Option:	**`pamService`**
Type:	string
Default:	`parstream`
Effect:	PAM service configuration used for authentication. Corresponds to a PAM module configuration file with the same name located in `/etc/pam.d`.

Option:	**`authenticationWrapperExecutable`**
Type:	string
Default:	`/opt/cisco/kinetic/parstream_authentication_1/parstream-authentication`
Effect:	External authentication application for PAM authentication including path. If the application cannot be found by the Cisco ParStreamserver, no authentication is possible.

Server-Section Options

In INI files, server-specific options are usually located in the section of the corresponding server. Thus, each server has its own subsection, which can be given any alphanumeric name used as the "servername".

For example:

```
[server.MyServer]
host = localhost
port = 6789
```

As usual, you can pass these options as commandline options, which override the INI file options. For example:

```
parstream-server --server.MyServer.host=localhost --server.MyServer.port=7777
    MyServer
```

There are many server-specific options so that they are presented in multiple tables:

- Functional server-specific options
- Non-functional server-specific options

Note that the behavior of the server is also influenced by the ExecTree option, described in section 13.3.4, page 140.

Functional Server-Section Options

Option:	**`host`**
Type:	string
Default:	localhost
Effect:	Server IP address.

Option:	**`port`**
Type:	integer
Default:	9042
Effect:	Server port number. Note: Cisco ParStream opens **this and consecutive ports** (see section 13.3.1, page 135 for details).

Option:	**`datadir`**
Type:	string
Default:	`./`
Effect:	Directory that contains the data partitions. In a remote server setup the specified path must be an absolute path.

Option:	**`rank`**
Type:	integer
Default:	0
Effect:	The rank (between 0 and 65535) defines the basic order for electing the cluster leader. The node with the lowest rank is the preferred leader. Each node rank has to be unique within the configured cluster to avoid ambiguities. Nodes with an duplicate rank will be excluded from the cluster and terminate. For more details see section 6.2.2, page 42

Ports

Cisco ParStream servers (or query nodes) always open the following public ports:

Port	Service	Description
`port`	parstream-netcat	Basic port as specified in the INI file and used for netcat connections
`port+1`	parstream-postgresql	Port for Postgres connections (i.e. psql and JDBC), and file transfer between Cisco ParStream servers and importers.

For an importer to connect successfully to a remote server, both of these ports must be open.

For managing clustering Cisco ParStream servers also open the following internal ports:

Port	Service	Description
`port+2`	parstream-cluster-messages	A TCP port used for partition synchronization and slave queries
`port+3`	parstream-partition-activation	A TCP port used for partition activation and load
`port+4`	parstream-find-nodes	A TCP port for the cluster leader election (see section 6.2.3, page 47)

In addition, within the following **global ports** might be shared among all nodes of a cluster:

Port	Service Description	
`registrationPort`	parstream-registration-port	A TCP port for the node registration at the leader, exchanging cluster node status information and health checking.

Even importers have to open a port, the `leaderElectionPort`. See section 6.2.3, page 47 for details.

Note that user authentication is used for the first two external ports. See section 9.2, page 78 for details.

Non-Functional Server-Section Options

Option:	**`maxExecutionThreads`**
Type:	integer
Default:	Value of the global option `maxExecutionThreads` (see section 13.2.1, page 127)
Effect:	Maximum number of threads to use for execution in total ($>$ `0`, section 15.1, page 158). Influences the default value of `maxMergeThreads` and `maxImportThreads`.

Option:	**`maxQueryThreads`**
Type:	integer
Default:	Value of the global option `maxQueryThreads` (see section 13.2.1, page 127)
Effect:	Maximum number of threads, out of the total `maxExecutionThreads` available, to use for queries. Value `0` means `maxExecutionThreads`.

Option:	**`maxMergeThreads`**
Type:	integer
Default:	Value of the global option `maxMergeThreads` (see section 13.2.1, page 127)
Effect:	Maximum number of threads, out of the total `maxExecutionThreads` available, to use for merging partitions (Chapter 14, page 151). Value `0` means `maxExecutionThreads`.

Option:	**`maxImportThreads`**
Type:	integer
Default:	Value of the global option `maxImportThreads` (see section 13.2.1, page 127)
Effect:	Maximum number of threads, out of the total `maxExecutionThreads` available, to use for server-sided import jobs, such as Java Streaming Imports (Chapter 19, page 216) and `INSERT INTO`'s (section 10.7, page 107). Value `0` means `maxExecutionThreads`.

Option:	**`defaultQueryPriority`**
Type:	string/integer
Default:	Value of the global option `defaultQueryPriority` (see section 13.2.1, page 128)
Effect:	Execution priority of query tasks unless another value is specified via the `SET` command (see section 27.10, page 373).

Option:	**`defaultImportPriority`**
Type:	string/integer
Default:	Value of the global option `defaultImportPriority` (see section 13.2.1, page 128)
Effect:	Execution priority of query tasks unless another value is specified via the `SET` command (see section 27.10, page 373) for `INSERT INTO` statements or via the Java Streaming Import Interface (see section 19, page 216).

Option:	**`defaultMergePriority`**
Type:	string/integer
Default:	Value of the global option `defaultMergePriority` (see section 13.2.1, page 128)
Effect:	Execution priority of merge tasks.

Option:	**`queryThrottlingInterval`**
Type:	integer
Default:	Value of the global option `queryThrottlingInterval` (see section 13.2.1, page 128)
Effect:	Milliseconds of thread time scaled by number of available threads of after which a query is considered “long running” and will get its effective priority reduced (section 15.1, page 158). Value `0` means the effective priority remains stable.

Option:	**`preloadingthreads`**
Type:	integer
Default:	Value of the global option `maxExecutionThreads` (see section 13.2.1, page 127)
Effect:	Number of preloading threads. Usually we use number of `maxExecutionThreads`. When your I/O subsystem isn’t fast, reduce this number to increase performance

Option:	**`preloadcolumns`**
Type:	string
Default:	nothing
Effect:	How to preload all columns (`complete`, `memoryefficient`, `nothing`). See section 15.9.3, page 177

Option:	**`preloadindices`**
Type:	string
Default:	nothing
Effect:	How to preload all indices (`complete`, `nothing`). See section 15.9.3, page 177

Option:	**`blockqueriesonpreload`**
Type:	boolean
Default:	false
Effect:	If set to true, no further queries are accepted until all configured columns and indices marked for preloading have been loaded.

Option:	**`maxConnectionsJdbc`**
Type:	integer
Default:	250
Effect:	Limits the number of concurrent connections to the server via the jdbc/odbc/postgres interface. The sum of maxConnectionsSocket and maxConnectionsJdbc should be well below the ulimit.

Option:	**`maxConnectionsSocket`**
Type:	integer
Default:	250
Effect:	Limits the number of concurrent connections to the server via the socket interface. The sum of maxConnectionsSocket and maxConnectionsJdbc should be well below the ulimit.

Option:	**`jdbcHandlingThreads`**
Type:	integer
Default:	8
Effect:	Number of threads executing requests over the postgres (jdbc, odbc) connection in parallel. If there are more requests coming in over the pooled connections, they will be queued until the next jdbcHandlingThread is free for executing the query.

Option:	**socketHandlingThreads**
Type:	integer
Default:	8
Effect:	Number of threads executing requests over the socket connection in parallel. If there are more requests coming in over the pooled connections, they will be queued until the next socketHandlingThread is free for executing the query.

Option:	**mappedFilesCheckInterval**
Type:	integer
Default:	10
Effect:	Time in seconds the check for whether the maximum of mapped files is reached, so that unmapping should happen. A value of 0 disables the unmapping mechanism (allows unlimited mapped files). You can change the value for a running server as a whole with an `ALTER SYSTEM SET` command (see section 27.11.1, page 375). See section 15.13, page 184 for details.

Option:	**mappedFilesMax**
Type:	integer
Default:	80,000
Effect:	Maximum number of mapped files before unmapping starts. If the number of memory mapped column and index files exceeds this value and a check for the number of mapped files happens, unmapping of (parts of) these files from memory is triggered. A value of 0 forces unmapping of all unused files with each unmapping check. You can change the value for a running server as a whole with an `ALTER SYSTEM SET` command (see section 27.11.1, page 375). See section 15.13, page 184 for details.

Option:	**mappedFilesAfterUnmapFactor**
Type:	floating-point
Default:	0.8
Effect:	General factor for unmapping mapped files if the limit of maximum number of mapped files is reached. The goal is to have less or equal than `mappedFilesMax * mappedFilesAfterUnmapFactor` mapped files after unmapping. Possible values have to be between 0.0 (all unused files are unmapped with each unmapping check) and 1.0 (after unmapping we should have `mappedFilesMax` mapped files). Because unmapping happens in chunks and mapped files that are used are not unmapped, the factor might not exactly be reached when unmapping happens. You can change the value for a running server as a whole with an `ALTER SYSTEM SET` command (see section 27.11.1, page 375). See section 15.13, page 184 for details.

Option:	**mappedFilesOutdatedInterval**
Type:	integer
Default:	3600
Effect:	If unmapping happens, all mapped files with no access for this amount of seconds are always unmapped. Note that as long as the limit `mappedFilesMax` is not reached, even outdated files are not unmapped. You can change the value for a running server as a whole with an `ALTER SYSTEM SET` command (see section 27.11.1, page 375). See section 15.13, page 184 for details.

Option:	**`mappedFilesMaxCopySize`**
Type:	integer
Default:	16384
Effect:	Maximum size of files in bytes that will be copied completely into heap memory upon first request instead of using a memory mapped approach. The files with sizes smaller than or equal to the value of `mappedFilesMaxCopySize` but will still be managed by the LRU approach in the same way as mapped files. and will therefore still be visible in the `ps_info_mapped_files` system table (see section 26.4, page 320). See section 15.13, page 184 for more details.

Option:	**`fileBlockTransferTimeout`**
Type:	float
Default:	Value of the global option `fileBlockTransferTimeout`
Effect:	Timeout of one file data block transfer in seconds used for this server

Option:	**`maxscanpartitionconcurrencylevel`**
Type:	integer
Default:	Value of the server option `maxExecutionThreads`
Effect:	Degree of parallelism for scanning partitions in the server startup phase (between 0 and `maxExecutionThreads`). Set value 0 to disable parallel partition scanning. The value is automatically limited to `maxExecutionThreads`

Option:	**`logscanpartitionprogressinterval`**
Type:	integer
Default:	10000
Effect:	Interval (number of partitions) for logging the partition scanning progress. Set value 0 to disable partition scanning progress.

Option:	**`mergeConcurrency`**
Type:	integer
Default:	1
Effect:	Number of parallel merges. (see section 14.1.1, page 153)

Option:	**`executionSchedulerMonitoringTimeDetailLevel`**
Type:	string
Default:	`summary`
Effect:	Control the detail level of the global execution scheduler event log (See section 9.5.1, page 84). Possible values are `summary` and `intervals`, where the latter causes logging of each execution slice (ca every 20ms) and may therefore generate very large logs. Overrides the global options of the same name.

Connection Pool Options

For the *connection pool* (see section 15.1, page 162), a couple of options are provided, which can be used by servers and importers.

Option:	**`connectionPool.numConnectionsPerNode`**
Type:	integer
Default:	4
Effect:	Number of connections pre-allocated by a cluster for upcoming queries to each other node. This number should correlate with the typical number of queries that occur in parallel. Note that with distributed hash separations such as DHSGB (see section 15.15.2, page 189) you always need 2 connections per query on the query master, so that the default 4 provides fast query support for 4 incoming parallel queries without DHSGB or 2 incoming parallel queries with DHSGB. The minimum is 1.

Option:	**`connectionPool.nodeErrorRetryInterval`**
Type:	integer
Default:	100
Effect:	Minimal interval in milliseconds to wait before attempting to connect to a server that had a connection error with the last trial.

Option:	**`connectionPool.connectionFetchTimeout`**
Type:	integer
Default:	5000
Effect:	Timeout in milliseconds for a query waiting for a connection, if not enough connections are available.

Option:	**`connectionPool.staleConnectionCheckInterval`**
Type:	integer
Default:	5000
Effect:	Interval in milliseconds to double check the availability of a pre-allocated connection.

Deprecated Server-Section Options

Execution Engine Options

Some parameters that influence the performance of query execution are configurable. The corresponding options are definable in the INI section with the name `[ExecTree]`. Additionally, these options can be changed inside a session of a running server using a SQL SET statement (see section 27.10, page 373). Future releases may expose additional parameters.

Option:	**`BitmapAggregationLimit`**
Type:	integer
Default:	40000
Effect:	The limit for the cardinality of values in aggregations up to which bitmap indices are used for the aggregation. If this parameter is set to 0 then bitmap indices are never used for aggregations. Applies to: bitmap indices

Option:	**`MonitoringMinLifeTime`**
Type:	integer
Default:	0
Effect:	Threshold for writing out monitoring information of queries. Queries with execution times above this threshold (milliseconds in realtime mode) will be written to the monitoring log (see section 9.5.1, page 84). A value of 0 disables this feature. Applies to: server

Option:	**`MonitoringImportMinLifeTime`**
Type:	integer
Default:	0
Effect:	Threshold for writing out monitoring information of imports. Imports with execution times above this threshold (milliseconds in realtime mode) will be written to the monitoring log (see section 9.5.1, page 84). A value of 0 disables this feature. Applies to: server/importer

Option:	**`MonitoringMergeMinLifeTime`**
Type:	integer
Default:	0
Effect:	Threshold for writing out monitoring information of merges. Merges with execution times above this threshold (milliseconds in realtime mode) will be written to the monitoring log (see section 9.5.1, page 84). A value of 0 disables this feature. Applies to: server

Option:	**`QueryMonitoringTimeDetailLevel`**
Type:	string
Default:	summary
Effect:	Detail level of the per-query event log (see section 9.5.1, page 84). Can be set to either `summary` or `intervals` (case insensitive) to record only summaries or each execution slice interval. Take care not to leave this option on all the time as it can produce potentially very large event logs. Applies to: server

Option:	**`MergeMonitoringTimeDetailLevel`**
Type:	string
Default:	summary
Effect:	Detail level of the per-merge event log (see section 9.5.1, page 84). Can be set to either `summary` or `intervals` (case insensitive) to record only summaries or each execution slice interval. Take care not to leave this option on all the time as it can produce potentially very large event logs. Applies to: server

Option:	**`ImportMonitoringTimeDetailLevel`**
Type:	string
Default:	summary
Effect:	Detail level of the per-import event log (see section 9.5.1, page 84). Can be set to either `summary` or `intervals` (case insensitive) to record only summaries or each execution slice interval. Take care not to leave this option on all the time as it can produce potentially very large event logs. Applies to: server/importer

Option:	**`SeparationAwareExecution`**
Type:	boolean
Default:	true
Effect:	In general, enable/disable all separation aware execution optimizations (see section 15.15, page 186 for details).

Option:	**`SeparationEnableDSGB`**
Type:	boolean
Default:	true
Effect:	Enable/disable Data Separated `GROUP BY` (DSGB) (see section 15.15.1, page 186 for details). Note that this optimization is only enabled if also the general option `SeparationAwareExecution` is enabled.

Option:	**`SeparationEnableHSGB`**
Type:	boolean
Default:	true
Effect:	Enable/disable Hash Separated `GROUP BY` (HSGB) (see section 15.15.2, page 189 for details). Note that this optimization is only enabled if also the general option `SeparationAwareExecution` is enabled.

Option:	**`SeparationEnableDSFA`**
Type:	boolean
Default:	true
Effect:	Enable/disable Data Separated Function Aggregations (DSFA) (see section 15.15.3, page 190 for details). Note that this optimization is only enabled if also the general option `SeparationAwareExecution` is enabled.

Option:	**`SeparationEnableDSJ`**
Type:	boolean
Default:	true
Effect:	Enable/disable Data Separated `JOIN` (DSJ) (see section 15.15.4, page 191 for details). Note that this optimization is only enabled if also the general option `SeparationAwareExecution` is enabled.

Option:	**`SeparationEnableHSJ`**
Type:	boolean
Default:	true
Effect:	Enable/disable Hash Separated `JOIN` (HSJ) (see section 15.15.4, page 191 for details). Note that this optimization is only enabled if also the general option `SeparationAwareExecution` is enabled.

Option:	**SeparationEnableDSI**
Type:	boolean
Default:	true
Effect:	Enable/disable Data Separated `IN` (DSI) (see section 15.15.5, page 197 for details). Note that this optimization is only enabled if also the general option `SeparationAwareExecution` is enabled.

Option:	**SeparationEnableDHS**
Type:	boolean
Default:	true
Effect:	Enable/disable all distributed hash separated optimizations, i.e. DHSGB. (see section 15.15.2, page 189 for details). Note that this optimization is only enabled if also the general option `SeparationAwareExecution` and option `SeparationEnableHSGB` are enabled.

Option:	**NumHashSeparatedStreamsPerNode**
Type:	integer
Default:	16
Effect:	Number of hash generated separated streams per participating node on group by columns with many different values. Must be greater than zero.

Option:	**NumAggregationChildren**
Type:	integer
Default:	32
Effect:	Split a long running aggregation node in small parallelized partial aggregations. Applies to: parallelization of aggregation

Option:	**GroupByBitmapLimit**
Type:	integer
Default:	the value of `BitmapAggregationLimit`
Effect:	Number of bitmap operations allowed in group by of bitmap aggregation. Applies to: bitmap indices

Option:	**VectorAggregationEnabled**
Type:	boolean
Default:	true
Effect:	Enable vector aggregation. Applies to: bitmap indices

Option:	**NumValuesVectorPreferred**
Type:	integer
Default:	64
Effect:	The upper bound of values where vector aggregation instead of bitmap aggregation is used. Applies to: bitmap indices

Option:	**MaxRhsValuesForLhsJoinBitmapScan**
Type:	integer
Default:	1024
Effect:	Number of bitmap operations allowed on left side of join operation. Applies to: bitmap indices, join

Option:	**MaxOutputBufferSize**
Type:	integer
Default:	1GB (`1,073,741,824` bytes)
Effect:	Maximum number of bytes which should be written to the output buffer in a single query. This value is processed by the output buffers for the query result of the socket interface (netcat/pnc), for transferring data between slaves and the query master, and for all DHSGB related communication between servers. To remove the limit set the parameter to 0. Applies to: all queries

Option:	**NumAggregationMapBuckets**
Type:	integer
Default:	30000
Effect:	Number of hash buckets used for aggregation data structures.

Option:	**NumDistinctAggregationMapBuckets**
Type:	integer
Default:	the value of `NumAggregationMapBuckets`
Effect:	Number of hash buckets used for distinct aggregation data structures. Should be reduced if distinct aggregations perform slow or use a lot of memory.

Option:	**ReorderGroupByFields**
Type:	boolean
Default:	true
Effect:	If true each ExecBitmapAggregation node tries to reorder the group by fields to speed up the bitmap operations required for the group by calculation. Because this reordering is based on some estimations about the bitmap compressions it may degrade the performance in some rare constellations. Therefore, the option may be switched of if it is better to force the execution engine to perform the group by bitmap operation in the order of the fields given in the `GROUP BY` clause.

Option:	**IterativeGroupByAggregation**
Type:	boolean
Default:	false
Effect:	Future use. Don't set this value without confirmation by Cisco ParStream consultants.

Option:	**MaxIterativeGroupByFields**
Type:	integer
Default:	0 (unlimited)
Effect:	Future use. Don't set this value without confirmation by Cisco ParStream consultants.

Option:	**ExecNodeDataMemoryStatistic**
Type:	string
Default:	CurrentAndMaxStatistic
Effect:	Set data memory statistic type of query processing on ExecNode level. Value "NoStatistic" disables this feature. Note: this option can only be set via the SET command (see section 27.10, page 373).

Option:	**`ExecTreeDataMemoryStatistic`**
Type:	string
Default:	NoStatistic
Effect:	Set data memory statistic type of query processing on ExecTree level. Value "CurrentAndMaxStatistic" enables this feature. Note: Because monitoring memory usage statistic on the ExecTree level needs to be thread safe, this option may slow down query processing significantly. For this reason, this option can only be set via the SET command (see section 27.10, page 373).

Option:	**`QueueActivateReceiver`**
Type:	integer
Default:	16
Effect:	Threshold to force the processing of receiving execution nodes. If at least this many results are in the input queue of a receiving node, the processing of this node is forced to continue. Note that the value always should be less than `QueueDeactivateSender`; otherwise deadlocks will occur.

Option:	**`QueueDeactivateSender`**
Type:	integer
Default:	255
Effect:	Threshold to pause the processing of sending execution nodes. If at least this many results are in the output queue of a sending node, the execution of the node is temporarily stopped. Note that the value always should be greater than `QueueActivateReceiver`; otherwise deadlocks will occur. Note also that the value always should be less than `QueueReactivateSender`.

Option:	**`QueueReactivateSender`**
Type:	integer
Default:	64
Effect:	Threshold to force the re-processing of sending execution nodes. If only up to this many results are in the output queue of a temporarily stopped sending node, the sending node is forced to continue its processing. Note that the value always should be greater than `QueueReactivateSender`.

Note:

- You can query the current value of these options via the system table `ps_info_configuration` (see section 26.3, page 312).

Import-Section Options

Import Definitions

Import-specific definitions are located in the import section of the corresponding server.

Each importer has its own import section, which can be given any alphanumeric name used as the "importername".

For example:

```
[import.MyImporter]
sourcedir = MyProject/import
```

As usual, you can pass these options as commandline options, which override INI file options. For example:

```
parstream-import --import.MyImporter.sourcedir=MyProject/import MyImporter
```

Note that the importer counts as cluster node and each cluster node must have a unique name. Hence its name cannot correspond to any server name.

There are many import-section options so that they are presented in multiple tables:

- Functional import-section options
- Non-functional import-section options

Functional Import-Section Options

Option:	**`sourcedir`**
Type:	string
Default:	*none*
Effect:	Directory with raw data. If this parameter is not set, then no raw data will be imported and instead only partitions will be merged. Can be overwritten in the command line with –sourcedir.

Option:	**`targetdir`**
Type:	string
Default:	
Effect:	Directory path used to temporarily store the partitions before they are transferred to the servers. Not setting this option is an error. Can be overwritten in the command line with –targetdir.

Option:	**`columnseparator`**
Type:	string
Default:	`;`
Effect:	Character used as column separator in CSV files. Might be a special character such as “`\t`”. Characters “`\`”, “`"`”, space, and newline are not allowed. See section 13.4.2, page 146 for details.

Option:	**`csvreadonly`**
Type:	boolean
Default:	false
Effect:	If set, imported CSV files are not moved to the `.backup` folder.

Option:	**`maxnumcsvfiles`**
Type:	integer
Default:	3 (Note, however, that the commandline option `--finite` changes this default value to 0 (unlimited); See section 13.1.3, page 117 for details.)
Effect:	Specifies the maximum number of CSV files that are imported into a single partition in one importer run. 0 means: no limit. Note that if you have many huge CSV files the value 0 may result in a server crash, because all CSV files are processes at once.

Option:	**`rank`**
Type:	integer
Default:	0
Effect:	The rank (between 0 and 65535) defines the basic order for electing the cluster leader. An importer never becomes leader, but for reasons of the cluster functionality the rank is required for importers too and has to be unique within the cluster. For more details see section 6.2.2, page 42

Option:	**`leaderElectionPort`**
Type:	integer
Default:	9046
Effect:	A TCP port for the cluster leader election (see section 6.2.3, page 47) for details.

Non-Functional Import-Section Options

Option:	**`writebuffersize`**
Type:	integer
Default:	12 * 1024
Effect:	Buffer size for serialized writing of column store data. IF the value is > 0, a central “FileWriterNode” writes to the different open column store data files chunky of this size. If `writebuffersize` equals `0`, column stores are written in parallel directly by the import nodes (which can easily exceed the limit of open files).

Option:	**`indexWriteBufferSize`**
Type:	integer
Default:	64 * 1024
Effect:	Buffer size for serialized writing of bitmap index data. If the value is > 0, a central “FileWriterNode” writes to the different open bitmap index files in chunks of this size. If the value is zero, bitmap index files are written in parallel directly by the import nodes (which can easily exceed the limit of open files).

Option:	**`fileBlockTransferBuffersize`**
Type:	integer
Default:	2*1024*1024
Effect:	Buffersize (bytes) for file data block.

Option:	**`numberoffetchnodes`**
Type:	integer
Default:	3
Effect:	Number of Fetch Nodes during import (=CSV files read in parallel)

Option:	**`numberOfWriterNodes`**
Type:	integer
Default:	1
Effect:	Number of column store File Writers during import (=write column stores in parallel)

Option:	**`inputFileBufferSize`**
Type:	integer
Default:	1,048,576 (2^{20})
Effect:	Size of the input file cache. This value should be larger than the longest line in CSV import files. It has to be `>=` 32,768 and `>= blobbuffersize/2` (see section 13.2.1, page 124). The value is automatically adjusted if this is not the case.

Option:	**`maxExecutionThreads`**
Type:	integer
Default:	Value of the global option `maxExecutionThreads` (see section 13.2.1, page 127)
Effect:	Maximum number of threads to use for execution in total (`> 0`, section 15.1, page 158). Influences the default value of `maxMergeThreads` and `maxImportThreads`.

Option:	**`maxMergeThreads`**
Type:	integer
Default:	Value of the global option `maxMergeThreads` (see section 13.2.1, page 127)
Effect:	Maximum number of threads, out of the total `maxExecutionThreads` available, to use for merging partitions (Chapter 14, page 151). Value `0` means `maxExecutionThreads`.

Option:	**`maxImportThreads`**
Type:	integer
Default:	Value of the global option `maxImportThreads` (see section 13.2.1, page 127)
Effect:	Maximum number of threads, out of the total `maxExecutionThreads` available, to use for server-sided import jobs, such as Java Streaming Imports (Chapter 19, page 216) and `INSERT INTO`'s (section 10.7, page 107). Value `0` means `maxExecutionThreads`.

Option:	**`defaultImportPriority`**
Type:	string/integer
Default:	Value of the global option `defaultImportPriority` (see section 13.2.1, page 128)
Effect:	Execution priority of query tasks unless another value is specified via the `SET` command (see section 27.10, page 373) for `INSERT INTO` statements or via the Java Streaming Import Interface (see section 19, page 216).

Option:	**`defaultMergePriority`**
Type:	string/integer
Default:	Value of the global option `defaultMergePriority` (see section 13.2.1, page 128)
Effect:	Execution priority of merge tasks.

Option:	**preloadingthreads**
Type:	integer
Default:	Value of the global option `maxExecutionThreads` (see section 13.2.1, page 127)
Effect:	Number of preloading threads. Usually we use number of `maxExecutionThreads`. When your I/O subsystem isn't fast, reduce this number to increase performance.

In addition, you can use option `ExecTree.MonitoringImportMinLifeTime` to log imports than run longer than a passed duration in milliseconds (see section 13.3.4, page 141).

Connection Pool Options

For the *connection pool* (see section 15.1, page 162), a couple of options are provided, which also can be used by importers. See section 13.3.2, page 139 for details.

Optimization Options

Cisco ParStream provides query rewrite optimizations (see section 15.8, page 172), which can be enabled and disabled with INI file settings (as described here) and via `SET` commands on a per-session basis (see section 21.3.1, page 260).

Theses options belong to the INI file section `[optimization]` and are designed in a way that you can enable or disable them as a whole or individually.

For example, the following setting enables all query rewrite optimizations:

```
[optimization]
rewrite.all = enabled
```

As another example, the following setting enables the "joinElimination" optimization only:

```
[optimization]
rewrite.all = individual
rewrite.joinElimination = enabled
```

The optimization options are in detail as follows:

Option:	**rewrite.all**
Type:	`enabled` or `disabled` or `individual`
Default:	`individual`
Effect:	If this option is `enabled`, all optimizations are enabled. If the parameter is `disabled`, all optimizations are disabled. If the parameter is `individual`, optimizations can be switched on and off individually (see below). The value of this option can be changed at runtime (see section 27.10.1, page 374).

Option:	**`rewrite.joinElimination`**
Type:	`enabled` or `disabled`
Default:	`disabled`
Effect:	If the optimization option `rewrite.all` is set to `individual`, the setting of this parameter switches the join elimination optimization (see section 15.8.1, page 173) on or off. The value of this option can be changed at runtime (see section 27.10.1, page 374).

Option:	**`rewrite.mergeJoinOptimization`**
Type:	`enabled` or `disabled`
Default:	`enabled`
Effect:	If the optimization option `rewrite.all` is set to `individual`, the setting of this parameter switches the merge join optimization (see section 15.8.3, page 174) on or off. The value of this option can be changed at runtime (see section 27.10.1, page 374).

Option:	**`rewrite.hashJoinOptimization`**
Type:	`enabled` or `disabled`
Default:	`enabled`
Effect:	If the optimization option `rewrite.all` is set to `individual`, the setting of this parameter switches the hash join optimization (see section 15.8.2, page 174) on or off. The value of this option can be changed at runtime (see section 27.10.1, page 374).

Option:	**`rewrite.sortElimination`**
Type:	`enabled` or `disabled`
Default:	`enabled`
Effect:	If the optimization option `rewrite.all` is set to `individual`, the setting of this parameter switches the elimination of sort nodes on or off. If this option is set to on, sort nodes will be optimized away if the input is already sorted correctly. The value of this option can be changed at runtime (see section 27.10.1, page 374).

Option:	**`rewrite.validationNodeOptimization`**
Type:	`enabled` or `disabled`
Default:	`enabled`
Effect:	If the optimization option `rewrite.all` is set to `individual`, the setting of this parameter switches the validation node optimization on or off. If this option is set to on, validation nodes will be optimized away if no validation is necessary. Currently the validation node checks if not-null fields are correctly filled and it checks the valid length of length restricted string fields. The value of this option can be changed at runtime (see section 27.10.1, page 374).

Merging Partitions

This chapter explains the features of Cisco ParStream to merge partitions.

Merging Partitions

As introduced in section 5.1.3, page 32, Cisco ParStream imports data in “minute” partitions (or “seconds” partitions for streaming import),

Servers (leaders) can then initiate the merge of imported partitions (“minute” partitions with suffix `_PM`, , or “seconds” partitions with suffix `_PS`) into aggregated partitions of a higher level (“minute” partitions with suffix `_PM` “hour” partitions with suffix `_PH`, “day” partitions with suffix `_PD`, “week” partitions with suffix `_PW`, or “month”/“final” partitions with suffix `_PF`).

Note again that the name “seconds”, “minute”, “hour” etc. are just pure abstractions for initial and further levels of merges. You can define when merges from one level to the next apply and therefore indirectly define your understanding of an “hour” or “day”.

This is controlled by several options (see chapter 13, page 116):

- *Whether* to perform merges
- *When* to perform merges
- And you can specify transformations for your data during merges (so that you can for example purge your data to have only one row with a sum of the old rows). See section 14.2, page 154 for details.

Note that you can temporarily disable scheduled server merges by sending the command “`ALTER SYSTEM CLUSTER DISABLE MERGE`” to a cluster node (see section 16.4.1, page 201).

In addition, note the following:

- To avoid getting too large partitions, merges also respect the general option to limit the maximum number of rows of a partition, `partitionMaxRows` (see section 13.2.1, page 126). For a merge, the criteria for `partitionMaxRows` is the number of rows in the source partitions to ensure that the merge result is never split up into multiple partitions as a consequence of exceeding the `partitionMaxRows` limit. For this reason, merges are usually partially performed or even skipped, if this would result into partitions that might become too large (i.e. if the sum of rows in a a particular merge would exceed `partitionMaxRows`). Thus, after a merge “minute to hour”, for instance, there might still be “minute” partitions.
- To have a fine-grained control for the size of merged partitions, you can also set the global options `partitionMaxRowsForMinuteMerge`, `partitionMaxRowsForHourlyMerge`, `partitionMaxRowsForDailyMerge`, `partitionMaxRowsForWeeklyMerge`, and `partitionMaxRowsForMonthlyMerge` (see section 13.2.1, page 126).
- However, if during a merge the data is transformed or purged using **ETL merge** statements (see section 14.2, page 154), merges are never skipped and might result in one source partition being replaced by a transformed partition. The transformed partition might even be smaller then, if the ETL merge purges the data via a `GROUP BY` clause (see section 14.2, page 154 for an example). Thus, after a merge still multiple partitions might be smaller than `partitionMaxRows`.

Continuous Partition Merge

The “Continuous Partition Merge” ensures partition merging, even when some of the cluster nodes are offline. Merging is controlled by the cluster leader. The leader ensures that all nodes merge the same source partitions to guarantee consistent partitions over all nodes. Merging will happen as long as at least one node (or a configured greater number of nodes) of a distribution group is active (online and synchronized). The resulting partitions of missed merges are synchronized in the same way as imported partitions, when a node gets online.

Merging Options

Several options allow to influence the merge behavior.

Merge Policy

The most important option is the global option `enableMerge`, initially specifying whether merges are enabled. This option can be set for an importer and for a server (see section 13.2.1, page 122). The default value is `true`.

For backward compatibility, also the global option `merge` is supported, which can have the following values:

Merge Policy	Effect
`true`	merges will be performed (default)
`reimport`	merges will be performed
`false`	no merges will happen
`none`	same

As written, you can enable or disable merges at runtime by sending the command “`ALTER SYSTEM CLUSTER DISABLE MERGE`” to a cluster node (see section 16.4.1, page 201).

Basic Merge Options

Merging is orchestrated by the cluster leader node (see section 6.2.1, page 41). Merging is possible even if the cluster is in a degraded state (node failures) (see section 14.1, page 152).

These merges can be controlled by the following global options, which should be set for *all* servers (query nodes) in the cluster because any query node can become a leader:

Option	Effect	Default
`merge`	defines the merge policy (see section 14.1.1, page 152)	`true`
`minutemergeschedule`	The times (UTC) at which the merging of seconds-partitions to minute-partitions should be performed. See section 14.1.2, page 154.	`0 * * * *`
`hourlymergeschedule`	The times (UTC) at which the merging of minute-partitions to hour-partitions should be performed. See section 14.1.2, page 154.	`0 0 * * *`
`dailymergeschedule`	The times (UTC) at which the merging of hour-partitions to day-partitions should be performed. See section 14.1.2, page 154.	`0 0 0 * *`
`weeklymergeschedule`	The times (UTC) at which the merging of day-partitions to week-partitions should be performed. See section 14.1.2, page 154. If not explicitly configured, the merge of week-partitions is disabled.	disabled
`monthlymergeschedule`	The times (UTC) at which the merging of week-partitions to month-partitions should be performed. See section 14.1.2, page 154. If not explicitly configured, the merge of month-partitions is disabled.	disabled

The format of the ...`schedule` options is described in section 14.1.2, page 154. The defaults are that the minute merge is performed every minute, the hourly merges are performed at every full hours and that the daily merge is performed at midnight UTC. Other merges are disabled by default.

Other Merge Options

There are other options, that influence merges:

- Option **`maxMergeThreads`** (see section 13.2.1, page 127) allows to define how many threads may be used by merges.
- Option **`partitionSearchConcurrencyLevel`** limits the number of merges that are composed in parallel by the leader. This limits the load in the leader and the nodes caused by searching for mergeable partitions and composing the triggered merges.

  The recommended value depends on the number of cluster nodes and the partitioning values and should be not lower than the number of cluster node. For example:

  ```
  partitionSearchConcurrencyLevel=4
  ```

- Option **`mergeConcurrency`** (see section 13.3.2, page 139) sets how many merges may take place in parallel. More merges in parallel might result in faster merge operations, but will also use more resources and can lead to e.g. poorer query performance.

There are further options to adjust the merge behavior. See section 13.2, page 119 for a complete overview.

Cron-Like Schedule Options Syntax

The ...`schedule` merge options for cluster leaders use a cron-like syntax.

The general syntax is as follows:

<seconds> <minutes> <hours> <days_of_month> <days_of_week>

For these five values, you can specify:

- A comma-separated list of absolute values (such as `17` or `13,15,19`) to have your job executed **exactly at** second, minute, ... Note that no spaces are allowed inside the comma-separated list.
- A `*` as a wildcard to have your job executed **every** second, minute, ...
- For seconds, minutes, hours: An expression `*/`n, such as `*/6`, to have your job executed **every n-th** second, minute, ...

  Note that n is newly processed for each minute, hour, day. That is, if for minutes `*/17` is specified, this is equivalent to specifying `0,17,34,51` (at the end of the hour, the last interval is 9 minutes).

The day of week must be specified numerically, the week starts with `0` for Sunday. That is, `1` stands for Monday, `2` for Tuesday, `3` for Wednesday, `4` for Thursday, `5` for Friday, and `6` for Saturday.

For example:

Value	Meaning
`*/6 * * * *`	Execute a job every six seconds (in every minute, every hour, every day)
`13 12 * * *`	Execute a job hourly at 12 minutes and 13 seconds after the full hour
`0 0 0 * 1`	Execute a job weekly every Monday at 00:00 AM
`0 */30 * 1 *`	Execute a job every 30 minutes on the first day of each month
`0 0 9,13,18 * *`	Execute a job each day at 9am, 1pm, and 6pm

ETL Merge

With the ETL merge feature, an ETL statement (“extract”, “transform”, and “load”) can be specified to define data transformations for every merge level (Seconds to minute, minute to hour, hour to day, day to week, and week to month).

Every supported select SQL statement can be used to define how to merge rows as long as the column types remain. The new contents after the merge is the result of the passed select statement.

The typical application of this feature is:

- **Purging data** (combining multiple entries of counters for a short interval into one entry for a longer interval). This is done via aggregations and `GROUP BY` clauses (see section 14.2.1, page 155 for an example).

Note that the merge policy `reimport` is required:

```
# if the cluster performs the ETL merge:
merge=reimport
```

Limitations of ETL Merge

Using an ETL merge, it is not allowed to change the partitioning schema or partitioning values. If some aspect of the partitioning is changed, the database will behave erroneously, because these violations will not be detected automatically.

Furthermore, an ETL merge SQL select statement has to return all values that are required by the schema. All values have to have the correct identifier and type. Compatible types can be coerced automatically. Overflows, type and other schema violations will be reported as errors and no merge will take place.

Note also that filtering during ETL merges is currently not possible. Using a WHERE clause in an ETL merge statement will result in an error.

Example for a Purging ETL Merge

A typical example for an ETL merge would be a merge that combines counters such as the number of hits of a web site.

Guess, we have the following table:

```
CREATE TABLE Hits
(
  url VARSTRING COMPRESSION HASH64 INDEX EQUAL,
  host VARSTRING(100),
  hits UINT64,
)
PARTITION BY url
DISTRIBUTE EVERYWHERE;
```

The column `hits` contains the number of hits in a given time period. Multiple rows in one or multiple import files will have entries here. Now if we have multiple import files and a merge is triggered, we can specify that all rows for a specific url and host are merged to one row containing the sum of all hits. The corresponding `SELECT` statement for such a request:

```
SELECT url, host, SUM(hits)
       FROM Hits
       GROUP BY url, host;
```

has to become a `ETLMERGE` statement in the table specification:

```
CREATE TABLE Hits
(
  url VARSTRING COMPRESSION HASH64 INDEX EQUAL,
  host VARSTRING(100),
  hits UINT64,
)
PARTITION BY url
DISTRIBUTE EVERYWHERE
ETLMERGE HOUR (
  SELECT url, host, SUM(hits) as hits
         FROM PARTITIONFETCH(Hits)
         GROUP BY url, host
)
;
```

Here, we specify a ETL merge statement for the merge to hour partitions (first, respectively second, merge level). Note that inside the ETL merge statement

- we read the data from table `Hits` with **`PARTITIONFETCH()`**
- the sum of `hits` (`SUM(Hits)`) becomes the new entry for column `hits` (`AS hits`)

To enable these kinds of merges for all levels, you have to specify:

```
CREATE TABLE Hits
(
  url VARSTRING COMPRESSION HASH64 INDEX EQUAL,
  host VARSTRING(100),
  hits UINT64,
)
PARTITION BY url
DISTRIBUTE EVERYWHERE
ETLMERGE MINUTE (
  SELECT url, host, SUM(hits) as hits
         FROM PARTITIONFETCH(Hits)
         GROUP BY url, host
)
ETLMERGE HOUR (
  SELECT url, host, SUM(hits) as hits
         FROM PARTITIONFETCH(Hits)
         GROUP BY url, host
)
ETLMERGE DAY (
  SELECT url, host, SUM(hits) as hits
         FROM PARTITIONFETCH(Hits)
         GROUP BY url, host
)
ETLMERGE WEEK (
  SELECT url, host, SUM(hits) as hits
         FROM PARTITIONFETCH(Hits)
         GROUP BY url, host
)
ETLMERGE MONTH (
  SELECT url, host, SUM(hits) as hits
         FROM PARTITIONFETCH(Hits)
         GROUP BY url, host
)
;
```

Note:

- You can pass the ETL merge strategy also via the command line. For example:

  ```
  parstream-server first --table.Hits.etlMergeHour="SELECT url, host,
      SUM(hits) as hits FROM PARTITIONFETCH(Hits) GROUP BY url, host"
  ```

  Before Version 2.2 you had to specify (which is still supported):

  ```
  parstream-server first --table.Hits.meta.etlMergeLevel1="SELECT url, host,
      SUM(hits) as hits FROM PARTITIONFETCH(Hits) GROUP BY url, host"
  ```

  See section 13.1.1, page 116 for details.

Performance Optimizations

This chapter describes optimizations that can improve query speed or reduce space requirements.

Note that while in some special situations one can influence how queries are processed internally so that certain queries can execute faster, these manual tunings can lead to worse performance in other situations. For this reason the trade-offs need to be examined carefully.

by Cisco ParStream. For these situations the queries execute faster, but others might be a little bit slower. Hence, it is a trade-off which kind of configuration fits best for your situation.

> **Note:**
>
> Be careful! A speedup in one kind of query may result in a performance penalty in another!

Execution Control

Servers offering 12, 24, 48, and even more parallel executing threads on the hardware layer are available off the shelf for prices well suited even for the mid-level server market. To directly transform this native low-level independent parallel execution capability into a direct advantage for Cisco ParStream users, Cisco ParStream manages all compute and I/O intensive *execution tasks* – such as queries, imports, and merge tasks – within a pool of parallel running independent threads. These threads are assigned execution time on the available compute cores. By this, the task scheduling directly translates the natural parallelism offered by today's affordable computer hardware into measurable high-level advantages for database users like scale-out and flexible concurrency under even heterogeneous query loads.

The strategy that Cisco ParStream implements strikes a balance between executing each and every individual query as early and fast as possible (using the first-in-first-out (FIFO) principle), while optimally balancing parallel competing tasks. The selected strategy together with optional parametrization by the DB administrator and by users allows for fair concurrency, by preventing short-running queries from being blocked by long running ones.

The principle works roughly as follows:

- You can specify the number of threads the Cisco ParStream database uses in total for task execution.
- These threads are then used by queries, imports, and merges.
- You can specify limits for each execution type so that imports or merges cannot block queries.
- You can specify the priority queries, imports, and merges initially get, which impacts the initial minimal number of threads the task gets.
- You can specify the amount of thread execution time after which the priority of long-running tasks decreases, which is done by reducing the minimal number of threads the task gets (it will always be at least 1).
- Priority influences how many threads a new tasks gets to execute the request. Priority is not used to starts tasks in different order. Thus, the tasks scheduler works on a strict FIFO manner.

Execution Control in Detail

In detail, there is first the configuration of the total number of threads and for which execution tasks these threads can be used:

- The **total number of threads** available for execution is set via the global/server option `maxExecutionThreads`. The default is 1.5 times the number of hardware threads rounded up. The default value should be suitable to absorb I/O related latencies, but if even under heavy query load the cpu remains idle increasing this value could help.
- To **avoid mutual blocking of query, merge, and import tasks** the total number of execution threads can be partitioned via the global/server options `maxQueryThreads`, `maxMergeThreads`, and `maxImportThreads`, which limit the number of threads which can be allocated to query, merge, and import task, respectively. The defaults are:

Option	Value	Meaning
`maxQueryThreads`	0	all threads can be used for queries
`maxMergeThreads`	maxExecutionThreads/4	at most a quarter of all threads can be used for merges
`maxImportThreads`	maxExecutionThreads/4	at most a quarter of all threads can be used for imports

Then, among these threads, you can influence via priorities the fraction of the available threads a specific task should be assigned:

- For queries, merges, and imports you can specify the following initial priorities:

Numeric Value	Convenience Value	Meaning
`2`	`high`	highest priority (try to use at least half of all available threads)
`4`	`medium`	default priority (try to use at least $\frac{1}{4}$ of all available threads)
`8`	`low`	lowest priority (try to use at least $\frac{1}{8}$ of all available threads)

  The numeric values are provided to be able to easily sort or filter according to priorities when inspecting relevant system tables. They are used in all system tables and `INSPECT` statements (see below). The convenience values are provided to be able to set priorities in a more readable way.
  The default is `4` or `medium`.
- You can set these initial priorities as global or server-specific options `defaultQueryPriority`, `defaultImportPriority`, or `defaultMergePriority` (see section 13.2, page 119).
- You can also set the default query or import priority per session via the SQL `SET` command (see section 27.10, page 373).
- The effect of these priorities is to set the minimal number of threads that initially are assign to the task. In fact, the numeric value is used as divisor to process the internal task attribute `min_num_threads`, which can be requested for running queries by the system table `ps_info_running_query` (see section 26.4, page 321) or the SQL `INSPECT` command (see section 27.6, page 362).
  For example, if `maxQueryThreads` is `16`, a query with `medium` priority (value `4`) will initially request `4` as minimum number of threads (`16` divided by `4`).

- Over time the "effective priority" of a task can shrink by using/setting the option `queryThrottlingInterval` globally or per server. Setting `queryThrottlingInterval` to a value greater than 0, instructs the scheduler to reduce the individual minimum number of threads for a given task after it has consumed `queryThrottlingInterval * maxQueryThreads` milliseconds of the total thread time during the execution time of the query. Total thread time is the sum of all threads a tasks uses. This means, throttling will only count when a query actually uses some CPU resources and throttling will happen earlier the more threads work on it concurrently. The implemented algorithm halves the minimal number of threads before inspection in every iteration down to a minimal value of 1. This introduces the concept of niceness and also guarantees completion.
- Based on the resulting minimum number of threads for each task at a certain time, the execution scheduler then works as follows:
  - First, all tasks get all their minimum number of threads, beginning with the oldest task.
  - If after assigning all the requested minimum number of threads there are still unassigned threads (leftovers) all these threads will be assigned to the oldest task (to finish oldest task first according to the FIFO principle).
  - If there are not enough thread to fulfill the minimum requests of all tasks, the youngest tasks are blocked.
  - If for the next tasks there are less threads available than the requested minimum number, the task will still start but only with the remaining number of available threads.
- In principle, tasks might temporarily not use all their requested threads. If Cisco ParStream detects this, the threads might be used by other tasks. This for example might happen during a streaming import, when Cisco ParStream is waiting for data from the client to arrive.

Thus, prioritization is not done by changing the order of tasks. Instead, if threads are available for an execution type, the tasks *start* strictly in the order of the statements (FIFO strategy). Priorities only influence the requested minimum number of threads, so that tasks with higher priority can finish faster/earlier.

Note that the FIFO strategy is employed cluster-wide, meaning the timestamp used to sort tasks is taken on the issuing cluster node.

Consequences of Execution Control by Example

The effect of this algorithm can be demonstrated by an example:

- Assume the Cisco ParStream database uses the default values for the number of threads (16 in total, at most 4 for imports, and at most 4 for merges).
- A first query running alone will get all 16 threads (4 as wished minimum, which is the result of 16 divided by 4, plus 12 not used by any other task). Thus we have the following situation:
  - Q1 has 16 threads (initial minimum 4 plus 12 leftovers)
- Assume we `SET` the query priority to `high` or `2` and we start a second query, the situation will be:
  - Q1 has 8 threads (initial minimum 4 plus 4 leftovers)
  - Q2 has 8 threads (initial minimum 8 (16 divided by 2)
- With the query priority `SET` to `low` or `8`, a third query task will have the following effect:

  - Q1 has 6 threads (4 as minimum plus 2 leftovers)
  - Q2 has 8 threads (initial minimum 8)
  - Q3 has 2 threads (initial minimum 2 (16 divided by 8)
- Starting an import with the default priority (`medium` or `4`) will have the following effect:
  - Q1 has 5 threads (4 as minimum plus 1 leftover)
  - Q2 has 8 threads (initial minimum 8)
  - Q3 has 2 threads (initial minimum 2)
  - I1 has 1 thread (initial minimum 1 (4 divided by 4)
- After setting query priority back to the default priority (`medium` or `4`) starting another query with default priority will have the following effect:
  - Q1 has 4 threads (4 as minimum)
  - Q2 has 8 threads (initial minimum 8)
  - Q3 has 2 threads (initial minimum 2)
  - I1 has 1 thread (initial minimum 1)
  - Q4 has 1 thread (initial minimum 4 (16 divided by 4) but only 1 thread left)
- Trying to process anything else is block until one of these tasks finished, is killed, times out, or is throttled. Thus:
  - Q1 has 4 threads (4 as minimum)
  - Q2 has 8 threads (initial minimum 8)
  - Q3 has 2 threads (initial minimum 2)
  - I1 has 1 thread (initial minimum 1)
  - Q4 has 1 thread (initial minimum 4 but only 1 thread left)
  - Q5 has 0 thread (initial minimum 4 (16 divided by 4) but no thread left)
- With throttling for the high-priority query Q2 (which might happen first, because throttling depends on total thread time consumption), the situation will be as follows:
  - Q1 has 4 threads (4 as minimum)
  - Q2 has 4 threads (initial minimum 8 halved)
  - Q3 has 2 threads (initial minimum 2)
  - I1 has 1 thread (initial minimum 1)
  - Q4 has 4 thread (initial minimum 4)
  - Q5 has 1 thread (initial minimum 4 but only 1 thread left)

Thus:

- By default half of the threads are reserved for queries only (a quarter might be used by imports or queries and a quarter might be used for merges and queries).
- The highest settable priority (`high` or `2`) will lead to only up to halve the number of available threads to be assigned to the first scheduled task, meaning that one cannot reserve the whole threadpool for a single task unless no other task is running.
- By default, *one* import or merge task only gets $\frac{1}{16}$ of all threads if there are too much other tasks so that there are no unassigned threads after fulfilling all requested minimums.

- To configure Cisco ParStream to reserve dedicated threads for each task type, you can set `maxExecutionThreads` to `maxQueryThreads+maxMergeThreads+maxImportThreads`.
- Even high priority queries might get blocked by many low priority queries. If this is a problem and you cannot wait for timeouts (option `limitQueryRuntime`, see section 13.2.1, page 120) or throttling, you can kill these queries using the `INSPECT` and `KILL` commands described below.

Thread Inspection and KILL Command

Additional inspection and control commands are available to help remedy critical corner cases of query processing:

- **Threadpool inspection**
  The `INSPECT THREADPOOL` command (see section 27.6, page 362) offers a quick insight in the execution engine.
  The command is provided to be able to query the state of the running tasks and their associated threads even if normal queries would be blocked, because it does not use the normal mechanisms of the execution engine.
  The result from this call has the same format as table data returned from query results and system tables showing the resource consumption of running tasks, identified by their `execution_id`.
- **Abort running queries**
  Using the `ALTER SYSTEM KILL` command (see section 27.11, page 375) you can terminate a running task by passing its `execution_id`.

Dealing with the Connection Pool

For an efficient and reliable communication between connected nodes a *connection pool* is provided for each node, which can be configured via server options (see section 13.3.2, page 139).

The connection pool roughly operates as follows:

- Based on a list of the connected servers each node always has the goal to have a specific number of connections available to each connected server for immediate use. If one or more of these available connections is used for a concrete query, new connections are asynchronously requested to re-establish the desired number of available connections. The number of available connections can be controlled by option `connectionPool.numConnectionsPerNode` (see section 13.3.2, page 139). The default is 4.
- If a trial to establish a new connection fails, the mechanism to request additional connections is paused for a certain period of time defined by the server option `connectionPool.nodeErrorRetryInterval` (see section 13.3.2, page 139). However, the connections already established still can be used by new queries.
- If a trial to establish new connections fails, this has no direct impact on the existing connections. Thus, Cisco ParStream does not assume in general that all connections are broken if there is a problem to establish a new one. If indeed all connections to a node are broken (e.g. because the node is no longer available), this will be part of the usual error handling when the existing connections are used again.

- In case a new query needs a connection, but the pool is exhausted, the query will be blocked until a new connection is available or `connectionPool.connectionFetchTimeout` (see section 13.3.2, page 139) is reached. This also implies that a query will stall for `remoteNodeConnectionPool.connectionFetchTimeout` milliseconds in case the connection to the remote node cannot be established.
  - In clusters with multiple nodes the query might fail or failover to the next redundant node after the timeout.
- When a query using a connection has finished (i.e. all data from a slave have been transmitted) the connection is given back into the pool. The pool will never drop such re-used connections, thus usually after some queries there will be more connections in each pool than `connectionPool.numConnectionsPerNode` (see section 13.3.2, page 139). Cisco ParStream also tries to re-use connections even if an error occurred during processing of sub-selects and should succeed doing so as long as there is no network failure outside of the Cisco ParStream server process. Note that currently such error-resilience is not implemented for insert slaves, such that the connections used to distribute data among the cluster during an INSERT INTO request or a streaming import will likely be dropped if the request fails or is rolled back. You can observe the number of connections currently in use as well as the number of re-use successes and failures via the system table `ps_info_remote_node` (see section 26.4, page 317).
- Cisco ParStream periodically checks whether available connections still can be used. This can be controlled by option `connectionPool.staleConnectionCheckInterval` (see section 13.3.2, page 139).
- You can check the current state of the connection pool using the system table `ps_info_remote_node` (see section 26.4, page 317).

  Note that the connection pool internally gets established with the first distributed query. For this reason, the initial state you can query with this system table might not be valid until the first distributed query was processed.

Careful Partitioning

Cisco ParStream is a database that is specialized to analyze huge amounts of data. For this, the data has to be partitioned so that typical queries are able to skip most of the data. Thus, appropriate partitioning is a key for good performance. Note that you can partition according to both data and functions over data. See section 5.1, page 29 for details.

Partition Exclusion

A key element for the performance of Cisco ParStream is a technique called partition exclusion. In order to minimize the amount of data that needs to be accessed by a query, Cisco ParStream uses different techniques to exclude partitions from the running executions.

A query is always mapped to an execution tree in which fetching leaf nodes read data from different partitions. The goal is to exclude reads when Cisco ParStream knows that there can't be data for the running query. Thus, partition exclusion minimizes the number of leaf nodes (breadth) of the tree by analyzing the given query with respect to the knowledge of the partitioning as well as knowledge about partition-wise value distribution based on bitmaps

This leads to the consequence that partitioning in Cisco ParStream should be designed in a way that ideally only the necessary data is read. However, too many partitions can also have drawbacks. Thus, the goal is to find the right design and granularity for partitioning to benefit from partition exclusion.

So, let's explain explained the technique by example.

Please note that there is a similar mechanism in place to exclude full cluster nodes from a query (see section 6.3.1, page 54).

Partition Exclusion by Partition Value

When a query is received, the `WHERE` condition of the statement is evaluated with respect to conditions on partitioning columns. In case of multilevel partitioning this is done in an iterative way. For this first level of exclusion no access to any partition data is necessary, this is purely done based on metadata.

Assuming we have the following table:

```
CREATE TABLE MyTable
(
  ts TIMESTAMP INDEX EQUAL,
  userId INT64 INDEX EQUAL,
  platform VARSTRING,
  etlDay DATE INDEX EQUAL CSV_COLUMN ETL,
  userIdGroup UINT64 INDEX EQUAL CSV_COLUMN ETL
)
PARTITION BY etlDay, userIdGroup
DISTRIBUTE EVERYWHERE
ETL (SELECT CAST(ts AS DATE) AS etlDay,
            userId MOD 12 AS userIdGroup
            FROM CSVFETCH(MyTable))
;
```

This yields a partitioning structure where we get a first partition level partitioning by the day the given `timestamp` belongs to (`etlDay`) and a second partition level of 12 partitions based on the value of `userId`. Thus, in total we get a partition hierarchy partitioning by the user ID per day.

Receiving a query such as

```
SELECT COUNT(*) FROM MyTable WHERE etlDay = date'2013-04-12';
```

will directly eliminate all partition subtrees for any `etlDay` not equal to `'2013-04-12'`. Thus, all partitions except the 12 partition in the subtree of `'2013-04-12'` are completely ignored.

This works on all levels, thus

```
SELECT COUNT(*) FROM MyTable WHERE userIdGroup = 5;
```

cannot prune full subtrees, but will exclude all partitions not having value 5 as `userIdGroup` (i.e. where the value of `userId MOD 12` yields 5).

Partition Exclusion via Bitmaps Indexes

Following the metadata based exclusion of partitions, the exclusion is also possible based on details available via existing bitmap indexes. The general principle is that, if it is possible to conclude that a partitions cannot contain any rows that are relevant for the query, the whole partition is skipped. For that, we only need the meta data stored into bitmap files; we doesn't have to perform the usual bitmap index processing. This optimization kicks in for all bitmap backed filter operations regardless of them being a partitioning attribute or not.

Following the example above, exclusion on partitioning relevant attribute

```
SELECT COUNT(*) FROM MyTable WHERE userId = 12345;
```

will exclude all partitions that do not contain a `userId` with value 12345 based on the existing bitmap dictionary. With partition exclusion by query analysis described above, we have to access at most one partition per day (exactly those with a value of 9 for `userIdGroup` because `12345 MOD 12` yields 9). Now by using bitmaps we can also skip full days, if the `userId` with value 12345 (or the corresponding value 9) is not present in all of them.

Consider another example:

```
SELECT COUNT(*) FROM MyTable WHERE platform <> 5;
```

Again partitions on the leaf level are skipped if we can tell by examining the partition level bitmap dictionary that it contains no relevant data. In this case we simple check if there is a bitmap for value 5 or not.

Following the exclusion of whole subtrees and partitions we further reduce the amount of data being accessed by utilizing the available indexes to exclude data from column stores or completely skipping the access to column stores if the required operation can be executed on the bitmap itself (for example aggregates like SUM / COUNT / AVG ...).

ORDER BY Bitmap Index Optimization

As introduced in section 5.3, page 35, Cisco ParStream supports different bitmap indices. It can be an advantage if the physical data is internally sorted according to the index value (especially for range indices). For this reason, Cisco ParStream provides the ability to force a sorting of the imported column data with a `ORDER BY` clause (for backward compatibility you can also use `SORTED BY` instead of `ORDER BY`).

That means, `ORDER BY` can be used to specify one or multiple columns to control the physical sorting of the data records in each partition during an import or merge.

At first, this has an effect in the size of the bitmaps, because a bitmap on the first order criteria has minimal size. In addition you can improve the the bitmap operation performance. As in most natural data of real world scenarios the column are related in some way sorting most times improves operation speed on the other column, too.

As usual, there are trade-offs. For this reason, you should double check the effect of this option with a small amount of typical data. However, good candidates for this optimization are columns with many different values (such as timestamps) if you mainly have queries that use WHERE clauses with with BETWEEN conditions for it. Then using a range index with the data sorted according to the data of this column, can become a big improvement.

In general, you should try to sort according to columns with biggest indices first. They should become smaller but you have to find out whether other columns grow.

Optimizing the Partition Access Tree

For a database partitioned according to a specific definition, it can be helpful to optimize partition access according to other columns. In fact, if we have a WHERE clause, using a column that is not specified as a partition, it can be helpful to create a partition access tree with this column.

For example, if we have partitioned according to columns `A` and `B`:

```
CREATE TABLE MyTable (
  A ...,
  B ...,
  C
)
PARTITION BY A, B
...
```

and typical queries use a `WHERE` clause for column `C`:

```
SELECT * FROM MyTable WHERE C = ...;
```

then the following optimization using `PARTITION ACCESS` might help:

```
CREATE TABLE MyTable (
  A ...,
  B ...,
  C
)
PARTITION BY A, B
PARTITION ACCESS A, C
...
```

Note that all columns listed for partition access have to have an index.

By specifying a `LIMIT`, you can enable/disable this feature for access tree subnodes that would have more subnodes than the specified limit. For example:

```
CREATE TABLE MyTable (
  A ...,
  B ...,
  C
)
PARTITION BY A, B
PARTITION ACCESS A, C LIMIT 5
```

```
...
```

The default partition access limit is `1`. Thus:

```
CREATE TABLE MyTable (
  ...
)
PARTITION BY A, B
PARTITION ACCESS A, C
...
```

is equivalent to

```
CREATE TABLE MyTable (
  ...
)
PARTITION BY A, B
PARTITION ACCESS A, C LIMIT 1
...
```

Smart Query Distribution

Since Version 2.2, Cisco ParStream evaluates there `WHERE` condition attached to a query to determine the set of known distribution values that can actually match the condition, and given this which nodes of the cluster it needs to actually involve in the evaluation of the query.

For example, given an initial distribution like the following:

```
CREATE TABLE MyTable (
  A ...,
  ...
)
PARTITION BY A, ...
DISTRIBUTE OVER A WITH INITIAL DISTRIBUTION (
  (1 TO node1 node2),
  (2 TO node2 node3),
  (3 TO node3 node1),
)
...
```

A query with a `WHERE` condition such as `... WHERE A=1` is processed knowing that `node1` is enough to evaluate it. This leads to a reduction in network traffic as well as robustness against node failures. So the example query will be evaluated, even if `node2` and `node3` are down, while an unconstrained query on the same table would not. This also works with internally generated distributions and more complex queries such as `... WHERE B>0 AND A=1`, which would again only involve `node1`.

Note, however, that subqueries with column aliases will break this mechanism. For example the query

```
SELECT * FROM (SELECT A AS B FROM MyTable) WHERE B = 1;
```

will currently be sent to all cluster nodes, and fail if `node2` **and** `node3` are down.

JOIN Optimizations

Cisco ParStream provides a couple of optimizations for `JOIN` (see section 27.3.1, page 330).

Note that Cisco ParStream executes a join with either a “Hash Join” or a “Nested Loop Join” algorithm depending on whether the join condition contains an equal predicate (`A.a = B.b`) or not. The joins are processed in order of appearance in the SQL statement. That is, Cisco ParStream generates a left deep join tree.

For example, the following query will produce a join between table `tabA` and table `tabB`, and join the result with table `tabC`:

```
SELECT * FROM tabA
         INNER JOIN tabB ON tabA.id = tabB.id
         INNER JOIN tabC ON tabB.id2 = tabC.id;
```

The following sections show the optimizations that can be applied for joins with predicates containing equality conditions.

Ordering Of Tables

Join queries result in a left deep tree. The query given as introductory example in section 15.7, page 168 results in the tree shown in figure 15.1.

Due to the tree’s structure the query is processed from the query’s rightmost to its leftmost table. **To achieve an optimal performance you should always make sure, that the table with the least data being fetched is the query’s rightmost table.** The table with the second least data should be the query’s second rightmost table and so on. This eventually results in the table with the most data fetched for the query is the leftmost table.

The data fetched from a table can be limited by a condition pushdown (see section 15.7.2, page 168). Hence, the table having the most data is not necessarily the table, which provides the most data for a given query.

Condition Pushdowns

The following subsections show the different strategies that are applied to limit the amount of data being fetched from a given table. Limiting the data fetched from tables eventually limits the amount of data that must be processed by the join node to calculate the join results.

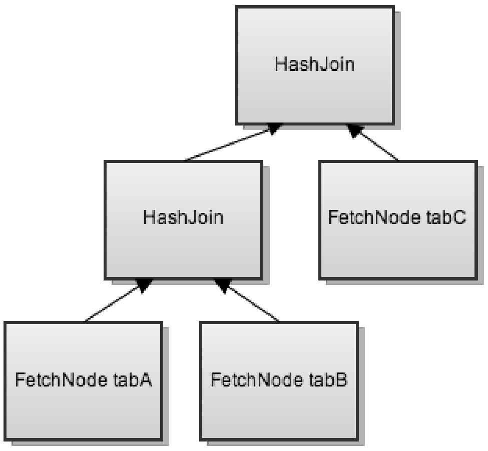


Figure 15.1: *Example of a left deep join tree*

WHERE Condition Pushdown

`WHERE` conditions and `JOIN` conditions are analyzed for terms that may be pushed down to the appropriate fetch nodes. This has different advantages:

- The data that must be fetched for a table can be reduced significantly.
- The filter, that must be processed by the join node in order to calculate the join's result tuples, is simplified.

However, there is a restriction for LEFT OUTER, RIGHT OUTER and FULL OUTER JOINs: Cisco ParStream may only push down predicates to the non-preserved table.

To push down terms of a WHERE condition, the WHERE condition is analyzed for sub terms that can be pushed down. Terms that are pushed down end up as filter in the appropriate FetchNode. Terms that cannot be pushed down are preserved as post condition filter. The join node will evaluate the post condition filter. **Therefore, an optimal performance is achieved if all terms can be pushed down to the appropriate FetchNodes, leaving an empty post filter.**

You can verify if sub terms of a WHERE condition are pushed down by using the sqlprint command (see section 16.4.2, page 202). For example, executing the following query:

```
sqlprint SELECT tabA.id, tabB.id FROM tabA INNER JOIN tabB ON tabA.id =
    tabB.id WHERE tabB.id = 2;
```

results in the following Preprocessed Tree:[1]

```
Preprocessed tree:

OutputNode requiredOutputRows: none  fields: (tabA.id, tabB.id) uniqueNodeId:
    4 limit: none offset: 0 output-format: default
```

[1] The Description Tree and Parametrized Tree are not of interest in order to analyze the WHERE condition pushdown.

```
JoinNodeSingle requiredOutputRows: none [parallelizable] fields: (tabA.id,
  tabB.id) uniqueNodeId: 5 condition:  join info: join condition: tabA.id =
  tabB.id, join type: INNER

 FetchNode requiredOutputRows: none [parallelizable] fields: (tabA.id)
  uniqueNodeId: 1 condition:  table: tabA fetchtype: eForceBitmapFetch

 FetchNode requiredOutputRows: none [parallelizable] fields: (tabB.id)
  uniqueNodeId: 2 condition: tabB.id = 2 table: tabB fetchtype: eNoPreset
```

Each of the nodes shown in the Preprocessed Tree above have an condition entry holding all filters that may be applied by the individual node on the data it processes. In the given example the WHERE filter is pushed down to the FetchNode for table `tabB`, limiting the amount of data being fetched from this table. Because there is no term for `tabA` in the WHERE filter, nothing is pushed down to `tabA`'s FetchNode.

However, even if there is no WHERE condition for `tabA` the amount of data being fetched for this table can be significantly limited by a Runtime Condition (See section 15.7.2, page 172).

There are three rules that are applied to calculate the WHERE condition pushdown. The three rules are explained in the following. For each rule an example is given, showing which condition is pushed down to the left, which condition is pushed down to the right, and which remaining conditions serve as postfilter.

- **Rule 1:** If the node is an AND instruction, each sub-tree may be pushed down if it can be answered with the subset of the input.
  For example:

  ```
  SELECT A.*, B.* FROM tableA A
         INNER JOIN tableB B ON A.id = B.id
         WHERE (A.x = 12) AND (B.y = 13);
  ```

  results in the following:
  - Pushdown left:

    ```
    (A.x = 12)
    ```

  - Pushdown right:

    ```
    (B.y = 13)
    ```

  - Postfilter:

    ```
    nothing
    ```

- **Rule 2:** If the node is an OR instruction, the whole tree may be pushed down if it can be answered with the subset of the input.
  For example:

```
SELECT A.*, B.* FROM tableA A
       INNER JOIN tableB ON A.id = B.id
       WHERE (A.x = 12 AND A.y = 3) OR (A.k > 10);
```

results in the following:

- Pushdown left:

  ```
  (A.x = 12 AND A.y = 3) OR (A.k > 10)
  ```

- Pushdown right:

  ```
  nothing
  ```

- Postfilter:

  ```
  nothing
  ```

- **Rule 3:** Additionally, Cisco ParStream can push down several terms of OR conditions, which must remain in the post filter to ensure correct results.
  Consider, for example:

  ```
  SELECT A.*, B.* FROM tableA A
         INNER JOIN tableB ON A.id = B.id
         WHERE (A.x = 12 AND A.y = 3) OR (A.k > 10);
  ```

  This pushdown is easier to understand if Cisco ParStream first builds a Conjunctive Normal Form (CNF) and then applies the two rules above:

  ```
  SELECT A.*, B.* FROM tableA A
         INNER JOIN tableB B ON A.id = B.id
         WHERE (A.x > 7 OR A.y < 3) AND (A.x > 7 OR B.k > 4) AND
               (B.i = 3 OR A.y < 3) AND (B.i = 3 OR B.k > 4);
  ```

  Thus, this results in the following:

  - Pushdown left:

    ```
    (A.x > 7 OR A.y < 3)
    ```

  - Pushdown right:

    ```
    (B.i = 3 OR B.k > 4)
    ```

  - Postfilter:

    ```
    (A.x > 7 OR B.k > 4) AND (B.i = 3 OR A.y < 3)
    ```

Runtime Condition Pushdown

The order in which the different tables are joined has a huge influence on the performance. Cisco ParStream employs a feature called Runtime Condition Pushdown, which performs a semi-join of the qualifying join keys of the right relation with the left relation. To generate this list of possible join keys, Cisco ParStream executes the right relation first. Hence, the sub-trees in the join tree are executed from the rightmost to the leftmost.

This allows us to reduce the amount of data that is fetched if the following conditions are met:

- The join column of the left relation has a bitmap index. If there is no bitmap index the Runtime Condition Pushdown is disabled for this column.
- The number of distinct values of the join key is smaller than MaxRhsValuesForLhsJoinBitmapScan (see section 13.3.4, page 140).
- It is an inner join or right outer join.

Therefore, the order of appearance of tables in a SQL statement should be from highest number of rows to lowest number of rows.

Given the following query:

```
SELECT tabA.id, tabB.id FROM tabA
       INNER JOIN tabB ON tabA.id = tabB.id
       WHERE tabB.id = 1;
```

This query will only fetch rows, where the condition `tabB.id=1` is met (See section 15.7.2, page 169). After these rows have been fetched, Cisco ParStream gets the Runtime Condition `tabA.id IN (1)`, which limits the amount of data being fetched from `tabA` significantly.

In a nutshell JOIN condition pushdown works as follows:

- Before execution of the hash join nodes all the right hand side fetches are executed. Currently there is a known limitation for the right hand side fetches that limit the amount of data per FetchNode on the right hand side to one million rows.
- In this case it means that Cisco ParStream fetches all ids from `tabB` where the condition `id = 1` is fulfilled.
- These values are cached and, because Cisco ParStream knows by the equi condition `tabA.id = tabB.id` the mapping to the respective field in `tabA`, the matching values are then pushed to the respective fetch nodes of `tabA`.

Query Rewrite Optimizations

Cisco ParStream can rewrite `JOIN` queries to simpler and therefore faster queries if certain edge conditions hold.

These optimizations are disabled by default. They can be enabled globally or individually using either INI file options (see section 13.5, page 149) and/or, on a per-session basis, using `SET` commands (see section 21.3.1, page 260).

Join Elimination

With the optimizer option `rewrite.joinElimination` (see section 13.5, page 149) Cisco ParStream can eliminate an `INNER JOIN` completely.

For example, the following query:

```
SELECT f.a FROM facts f INNER JOIN dimension d ON d.id = f.d_id
```

would internally be rewritten to:

```
SELECT f.a FROM facts f WHERE f.d_id IS NOT NULL
```

If enabled, this optimization will happen if the following conditions hold:

- **Rule 1:** It is an inner join. This optimization is not applicable to other join types.
- **Rule 2:** The join condition consists of a single equal predicate only.
- **Rule 3:** Let the join condition be `LHS_TABLE.X = RHS_TABLE.Y`. Then, `LHS_TABLE.X` must have been declared as referencing `RHS_TABLE.Y`. The column definition of `LHS_TABLE.X` must look like:

  ```
        CREATE TABLE LHS_TABLE {
                ...
                X ... REFERENCES RHS_TABLE.Y
                ...
        ) ...
  ```

  and `RHS_TABLE.Y` is declared to be `NOT NULL` and `UNIQUE`, or `PRIMARY KEY` which implies `NOT NULL` and `UNIQUE`. The definition of `RHS_TABLE.Y` looks like:

  ```
        CREATE TABLE RHS_TABLE {
                ...
                Y ... NOT NULL UNIQUE ...
                ...
        ) ...
  ```

  or

  ```
        CREATE TABLE RHS_TABLE {
                ...
                Y ... PRIMARY KEY ...
                ...
        ) ...
  ```

- **Rule 4:** Neither the select-list, nor `WHERE`, `ORDER BY`, `GROUP BY`, `HAVING` or anything else may reference columns of the right-hand side table.

The REFERENCES declaration above is a foreign key relation: every non-null value in LHS_TABLE.X must be contained in RHS_TABLE.Y. Please note that Cisco ParStream does neither check this, nor the UNIQUE declaration of RHS_TABLE.Y. It is the responsibility of the user to ensure that these properties hold. Wrong query results are the consequence if the properties do not hold.

Hash Join Optimization

With the optimizer option rewrite.hashJoinOptimization (see section 13.5, page 150) Cisco ParStream can speed up joins by converting nested loop joins to hash joins.

If enabled, this optimization will happen if the following condition holds:

- **Rule 1:** The join condition contains at least one equi-condition.

Merge Join Optimization

With the optimizer option rewrite.mergeJoinOptimization (see section 13.5, page 150) Cisco ParStream can use a merge-join strategy to reduce memory consumption and speed up joins.

If enabled, this optimization will happen if the following conditions hold:

- **Rule 1:** The join is an inner, left-outer, right-outer or full-outer join.
- **Rule 2:** The join condition contains at least one equi-condition. Example:

```
    SELECT * FROM tabA INNER JOIN tabB ON tabA.a = tabB.c AND tabA.b =
tabB.d
```

- **Rule 3:** The tables being joined have a common sorting, considering equi-conditions. In the example of rule 2, the equi-conditions are tabA.a = tabB.c and tabA.b = tabB.d. Now consider a table definition like this:

```
CREATE TABLE tabA (
    ...
)
...
SORTED BY a ASC, b DESC
...
```

```
CREATE TABLE tabB (
    ...
)
...
SORTED BY c ASC, d ASC
...
```

In this case, there is a common sorting containing one column pair tabA.a and tabB.c.

- **Rule 4:** One of the following holds true:
  - The join can use data separation (see section 15.15.4, page 191 for details).
  - The option `NumHashSeparatedStreamsPerNode` has the value 1 (see section 13.3.4, page 143).

Sort Elimination

With the optimizer option `rewrite.sortElimination` (see section 13.5, page 150) Cisco ParStream can speed up queries by removing unnecessary sorting operations.

If enabled, this optimization will happen if the following condition holds:

- **Rule 1:** The data being queried is sorted by an `ORDER BY` statement that is identical to or a subset of an `ORDER BY` statement in the `CREATE TABLE` statement.

Query Rewrite Analysis with ANALYZE REWRITE

The effect that the rewrite optimizer performs can be inspected with the `ANALYZE REWRITE` command that can be issued over the netcat interface.

The syntax of the command is:

```
<analyze statement> ::=
   "ANALYZE" "REWRITE" [ <analysis verbosity ] <SQL statement to analyze>

<analysis verbosity> ::=
   "ALL"
   | "NEARMATCHES"
   | "MATCHES"

<SQL statement to analyze> ::=
   <dynamic select statement>
   | <insert statement>
```

The verbosity determines how much output is generated. `ALL` is the most verbose output, `MATCHES` is the least verbose output. If no verbosity is specified, it defaults to `ALL`.

Small Optimizations

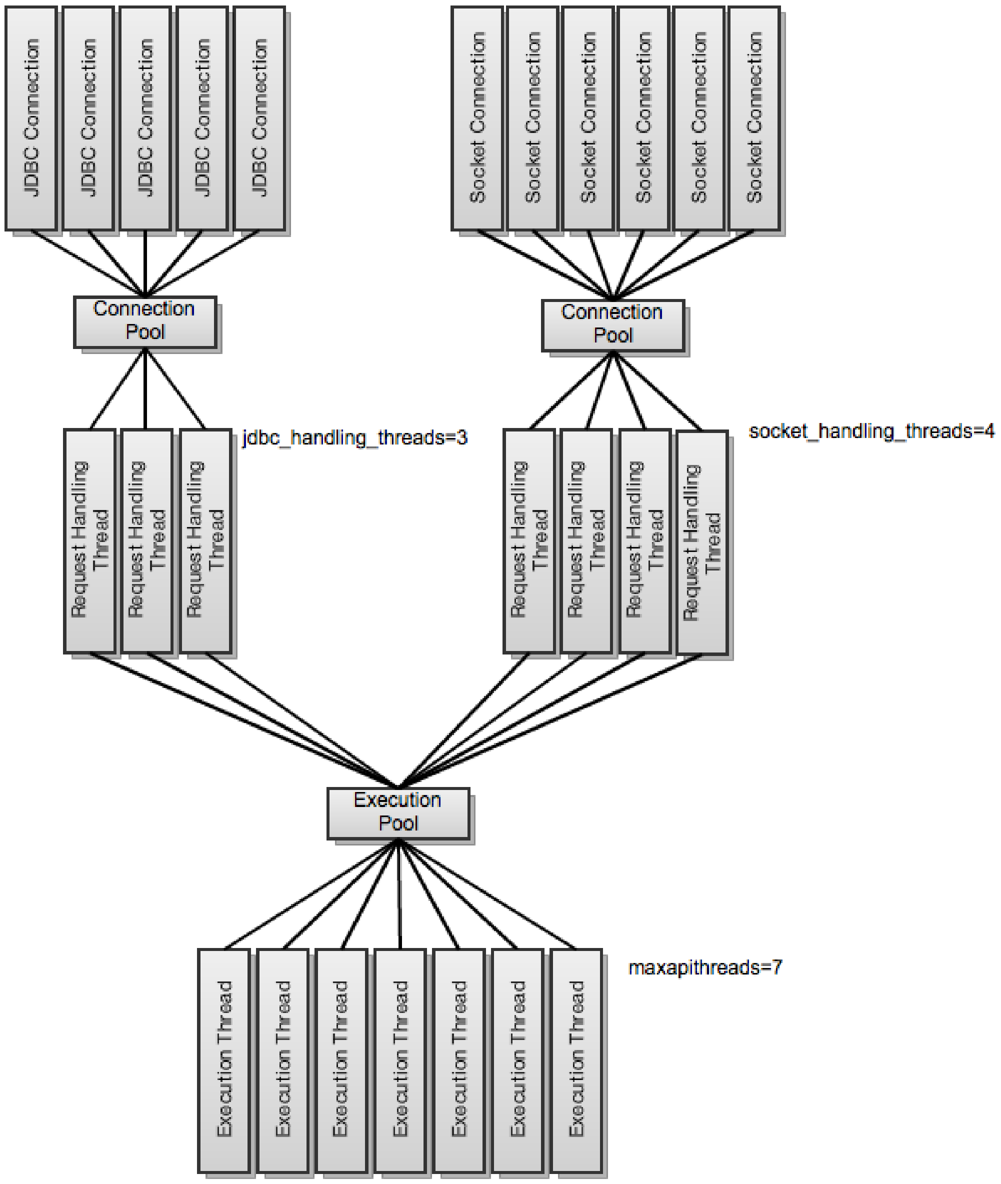
Several other options allow you to optimize the behavior of the Cisco ParStream database.

ExecTree Options

Note that the performance of the Cisco ParStream server is also influenced by the ExecTree option, described in section 13.3.4, page 140.

Threadpool Settings

Cisco ParStream uses several threads organized in thread pools, for which you can specify the size. For example:



The incoming threads execute one request from all connections. There are a maximum number of `jdbc_handling_threads + socket_handling_threads` running and handling queries at the same time. The SQL parsing and creation of the execution plan is done in this Request Handling Thread. After the execution plan is created, the query executes in parallel (and also in parallel with other queries) in the Execution Thread Pool. The number of execution threads can be set with the option `maxExecutionThreads` and `maxQueryThreads` (see section 13.2.1, page 127).

Preloading

A couple of options allow to preload data so that queries for this data gets accelerated. The following server-specific options (see section 13.3, page 134) allow to preload data:

Option	Effect	Default
`preloadcolumns` *val*	Whether and how to preload columns. Possible values are: `complete`: preload all columns and map files (for strings and blobs) `memoryefficient`: preload all columns, but when column is a string or blob only the map-files are preloaded `nothing`: preload nothing	`nothing`
`preloadindices` *val*	Whether and how to preload all indices. Possible values are: `complete`: preload all indices `nothing`: preload no indices	`nothing`
`preloadingthreads` *num*	Number of preloading threads. Usually Cisco ParStream uses the number of the option `maxExecutionThreads` (see section 13.2.1, page 127), but when your I/O subsystem isn't fast, reduce these number to increase performance	`maxExecutionThreads`
`blockqueriesonpreload` *bool*	If set to true, no further queries are accepted until all configured columns and indices marked for preloading have been loaded.	false

For example:

```
[server.srv1]
preloadcolumns = memoryefficient
preloadindices = complete
```

In addition, you can specify for individual columns whether to preload column data and/or indices by using the `PRELOAD_COLUMN` and/or `PRELOAD_INDICES` clause (see section 24.2.4, page 284).

Note that you can query which data is loaded via the system table `ps_info_mapped_file` (see section 26.4, page 320).

Column Store Compressions

Column store compression is a technique that can reduce the size of column stores in the file system and/or in memory.

Cisco ParStream currently supports the following forms of column store compression:

- A *Sparse Compression* approach, which optimizes column stores when there is a high number of *one* typical value (see section 15.10.1, page 178).

- A *Dictionary Compression* approach, which optimizes column stores when there is a hit number of *multiple* typical value (see section 15.10.2, page 179).
- A low-level compression that compresses column stores using the `LZ4` algorithm (see section 15.10.3, page 181).

Sparse Column Store Compression

Sparse column store compression is effective on columns with a high number of equal values.

Note that, if there are *multiple* typical values, then dictionary column compression usually is a better approach (see section 15.10.2, page 179).

How Sparse Compression Works

The idea of the sparse column store compression is to store one value, the most frequent one in the column store, once in the column store, as so-called default value.

The values different from the default value are stored together with their row id in segments of the column store, which are regions of the column store file. How many segments are written to a sparse column store during import depends, among other things, on a maximum segment size that can be configured with the global options `columnStoreSegmentSize` (see section 13.2.1, page 126).

The effectiveness of the sparse compression depends on how often the default value occurs in the column store. The higher the number of occurrences of the default value compared to the number of non-default values is, the better will the column store be compressed.

Sparse-compressed column stores are not only smaller in the file system, they also save room when they are loaded into memory, and due to their smaller size also save IO time.

Sparse column store compression is activated for a column by specifying a `COMPRESSION SPARSE` with an optional default value in the `CREATE TABLE` statement. If no default value is specified, `NULL` is used because that is a very frequent case.

Note that only fixed-width column types can use the sparse column compression. These are all integer and floating-point column types, all datetime types, `BITVECTOR8`, and hashed strings and blobs of singularity `SINGLE_VALUE`.

Note also that one cannot combine `SPARSE` compression with `LZ4` compression.

Examples

A simple example of creating a sparse column store:

```
CREATE TABLE SomeTable (
  col1 INT16 COMPRESSION SPARSE
)
DISTRIBUTE EVERYWHERE;
```

Another example, using VAT with a default value of 19 percent (as it is in Germany, for instance):

```
CREATE TABLE Expenses (
  vat UINT8 COMPRESSION SPARSE SPARSE_DEFAULT 19,
  ...
)
DISTRIBUTE EVERYWHERE;
```

You can also use a DEFAULT clause to specify the default values used for sparse compression:

```
CREATE TABLE Expenses (
  vat UINT8 DEFAULT 19 COMPRESSION SPARSE,
  ...
)
DISTRIBUTE EVERYWHERE;
```

But note that currently such a DEFAULT clause is no general default value in case no value is give during imports.

A more complex example of creating sparse columns showing all types for which sparse compression is available:

```
CREATE TABLE three_segments
(
  bitvector8_col BITVECTOR8 COMPRESSION SPARSE,
  uint8_col UINT8 COMPRESSION SPARSE,
  uint16_col UINT16 COMPRESSION SPARSE,
  uint32_col UINT32 COMPRESSION SPARSE,
  uint64_col UINT64 COMPRESSION SPARSE,
  int8_col INT8 COMPRESSION SPARSE,
  int16_col INT16 COMPRESSION SPARSE,
  int32_col INT32 COMPRESSION SPARSE,
  int64_col INT64 COMPRESSION SPARSE,
  float_col FLOAT COMPRESSION SPARSE,
  double_col DOUBLE COMPRESSION SPARSE,
  shortdate_col SHORTDATE COMPRESSION SPARSE,
  date_col DATE COMPRESSION SPARSE,
  time_col TIME COMPRESSION SPARSE,
  timestamp_col TIMESTAMP COMPRESSION SPARSE,
  varstring_col VARSTRING COMPRESSION HASH64, SPARSE,
  blob_col BLOB COMPRESSION HASH64, SPARSE,
)
DISTRIBUTE EVERYWHERE;
```

See also section 24.2.4, page 285 for details.

Dictionary Column Store Compression

Dictionary column store compression is most effective on columns with small numbers of distinct values.

Note that, if there is only *one* typical value, then sparse column compression usually is a better approach (see section 15.10.1, page 178).

How Dictionary Compression Works

A dictionary-compressed column stores contains a dictionary of all the distinct values in the column store. Each distinct value has an index in that dictionary.

For each row in the column store, the index of the row's value in the dictionary is stored in the column store instead of the value itself. Because indices can be stored as bit-fields that take much less room than the value itself, this can save a significant amount of memory.

Dictionary-compressed column stores are not only smaller in the file system, they also save room when they are loaded into memory, and due to their smaller size also save I/O time.

Dictionary column store compression is activated for a column by specifying a `COMPRESSION DICTIONARY` in the `CREATE TABLE` statement (see section 24.2.4, page 284).

Note that only fixed-width column types can use the dictionary column compression. These are all integer and floating-point column types, all date/time types, `BITVECTOR8`, hashed strings, and blobs of singularity `SINGLE_VALUE`.

Note also that one cannot combine `DICTIONARY` compression with `LZ4` compression.

Examples

A simple example of creating a dictionary column store:

```
CREATE TABLE SomeTable (
  col1 INT16 COMPRESSION DICTIONARY
)
...
;
```

A more complex example of creating dictionary columns showing all types for which dictionary compression is available.

```
CREATE TABLE SomeOtherTable
(
  bitvector8_col BITVECTOR8 COMPRESSION DICTIONARY,
  uint8_col UINT8 COMPRESSION DICTIONARY,
  uint16_col UINT16 COMPRESSION DICTIONARY,
  uint32_col UINT32 COMPRESSION DICTIONARY,
  uint64_col UINT64 COMPRESSION DICTIONARY,
  int8_col INT8 COMPRESSION DICTIONARY,
  int16_col INT16 COMPRESSION DICTIONARY,
  int32_col INT32 COMPRESSION DICTIONARY,
  int64_col INT64 COMPRESSION DICTIONARY,
  float_col FLOAT COMPRESSION DICTIONARY,
  double_col DOUBLE COMPRESSION DICTIONARY,
  shortdate_col SHORTDATE COMPRESSION DICTIONARY,
  date_col DATE COMPRESSION DICTIONARY,
  time_col TIME COMPRESSION DICTIONARY,
  timestamp_col TIMESTAMP COMPRESSION DICTIONARY,
  varstring_col VARSTRING COMPRESSION HASH64, DICTIONARY,
  blob_col BLOB COMPRESSION HASH64, DICTIONARY,
)
...
;
```

See also section 24.2.4, page 285 for details.

`LZ4` Compression

This column store compression technique uses `LZ4` to reduce the size of column store files in the file system.

Caution: LZ4 compression has its benefits but can also have significant drawbacks:

- LZ4 shortens import speed if disk I/O is significantly slower than the time to compress the data.
- However, when reading data from the column store to process queries, both more memory and time is needed to uncompress the data. In fact, a column store is completely mapped into memory to uncompress it.

That is, **never turn LZ4 compression on without evaluating the approach and comparing the effect both with and without LZ4 compression**.

For details how to enable `LZ4` compression, see section 24.2.4, page 285.

LIMIT optimization

The `LIMIT` optimization is an optimization which comes with no user configurations required. The `LIMIT` in the OutputNode will be pushed down the expression tree as far as possible, as illustrated the following examples:

```
SELECT * FROM ... [WHERE ...] LIMIT N
```

Rules for this kind of command:

- no node transmits more than `N` results
- no buffers of more than `N` rows are kept
- on each node the query stops once `N` results have been collected
- due to internal parallelism this does not guarantee that more than N results are ever produced but from each partition only up to N values are extracted

```
SELECT * FROM ... [WHERE ...] ORDER BY ... LIMIT N
```

Rules for this kind of command:

- no node transmits more than `N` results
- all data matching `WHERE` will still be fetched
- sort will be massively parallelized via initial sort/sort merge strategy and sort on slave nodes as well as query master

- internal buffers, the number of which is usually on the order of number of selected partitions, are limited to N rows

```
SELECT sum(...) FROM ... [WHERE ...] GROUP BY ... [HAVING ...] LIMIT N
```

Such queries will not benefit greatly from the optimization done here, because Cisco ParStream does not optimize inside the aggregations themselves, and the aggregations will only produce results after all data has been processed.

```
SELECT sum(...) FROM ... [WHERE ...] GROUP BY ... [HAVING ...] ORDER BY ...
    LIMIT N
```

As above, but keep in mind that the optimization only applies after the `GROUP BY` has been executed.

```
SELECT sum(...) FROM ... [WHERE ...] LIMIT N
```

Any limit > 0 will not result in any change in query behavior, because only one row can ever be expected.

`LIMIT 0` will not trigger any actual data processing, but only return the resulting field-list and can thus be safely used to extract column lists and perform quick validity checks of queries.

Parallel Sort

Sort is parallelized across cluster nodes and partitions. Each partition will get its own SortNode assigned, which will be generated automatically by the server. Additionally SortMergeNodes will combine the results of the separate child nodes, which can be SortNodes, other SortMergeNodes or TransportNodes for distributed queries, as illustrated for a simple example in figure 15.2.

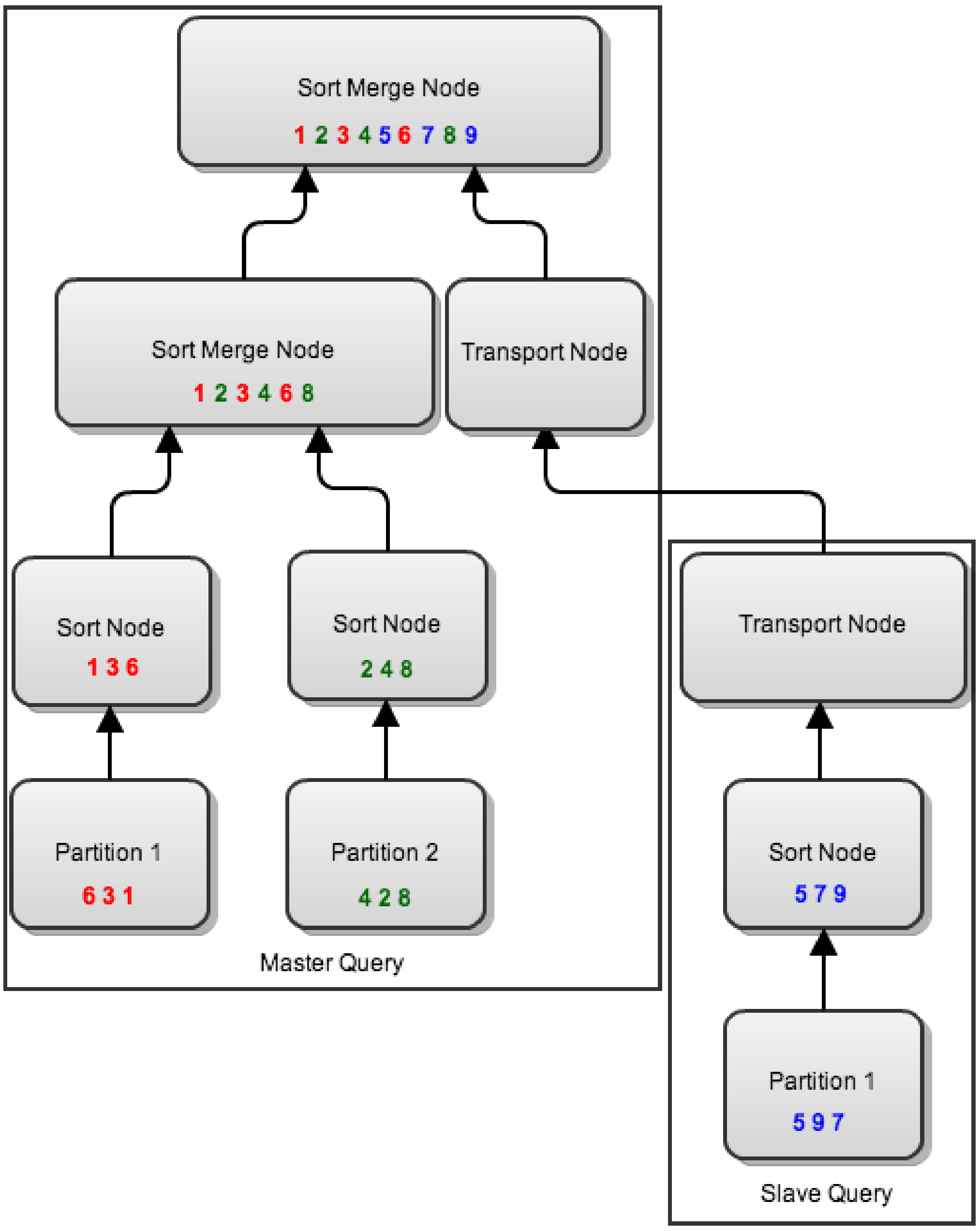


Figure 15.2: *Query processing of the parallel sort.*

Controlling the Number of Mapped Files

Cisco ParStream tries to hold a useful amount of data in memory to provide the best possible performance. However, keeping too much data in memory is counter-productive because the system might start to swap. For this reason, you have the ability to control how much and how long data is kept in memory.

A mapped file collector is provided to unmap old data if certain conditions are met. The conditions roughly are:

- A specified limit of maximum mapped files is reached
- The last access to the data files is older than a specified amount of time

The following server options control this behavior in detail:

Option	Effect	Default
`mappedFilesMax`	maximum number of mapped files before unmapping happens	80,000
`mappedFilesCheckInterval`	Interval in seconds that limit of maximum number of mapped files is checked	10
`mappedFilesOutdatedInterval`	If unmapping happens, all files with an access older than this amount of Seconds are unmapped	3600 (1 hour)
`mappedFilesAfterUnmapFactor`	Factor for `mappedFilesMax` to compute the resulting number of mapped files after unmapping.	0.8
`mappedFilesMaxCopySize`	Maximum size of files in bytes that will be copied completely into heap memory instead of using a memory mapped approach.	16384

Note:

- In principle the algorithm for unmapping files follows a LRU (least recently used) approach, which means that if too much files are mapped into memory, the files with the oldest last access are unmapped. Note however that to optimize this algorithm, individual files are not unmapped strictly according to their access timestamps.
- Note that as long as the limit `mappedFilesMax` is not reached, even outdated files are not unmapped.
- A value of 0 for `mappedFilesMax` or `mappedFilesCheckInterval` disables the whole unmapping algorithm.
- See section 26.4, page 320 for system table `ps_info_mapped_file`, listing which files are currently memory mapped.
- The `mappedFilesAfterUnmapFactor` is used to specify the target number of mapped files when unmapping applies. With the default 80,000 for `mappedFilesMax` the default `0.8` for `mappedFilesAfterUnmapFactor` means that the goal is to unmap files so that less or equal 64,000 mapped files exist afterwards. Thus, if you had 90.000 mapped files when unmapping happens, the goal is to unmap at least 26,000 files. A factor of 0 means that all files shall get unmapped.

  Note however, that this factor is only a rough goal. Mapped files are unmapped in chunks so that more files might get unmapped. It might also happen that less files are unmapped because the files are still/again in use or new mapped files were added while unmapping happens.

- The mappedFilesMaxCopySize option deals with the fact that loading files as a whole might be faster than using an memory mapped approach for small files. Usually, the system call mmap() internally uses pages of 4096 bytes size to manage the file access. Every mmap-segment will add an additional entry into the processes page table. If the number of entries grows, it may exceed a threshold, which noticeably slows down every subsequent memory access addressing prior unused pages (page fault). Therefore using mmap for files smaller than a certain size makes no sense, because the performance penalty for the page faults is very high and the page only consist of a small number of pages. For files smaller than one page (4096 bytes) the effect is even worse, because they will be anyway mapped completely from disk storage into memory. The files with sizes smaller than or equal to the value of mappedFilesMaxCopySize will therefore be copied from disk to memory in full, but will be managed by the LRU approach in the same way as mapped files and will therefore be visible in the ps_info_mapped_files system table (but because they are not really memory mapped anymore, they will not be visible in the processes /proc/self/maps system file).
- You can set the values at runtime with a ALTER SYSTEM SET command (see section 27.11.1, page 375). Note that it might take up to mappedFilesCheckInterval seconds until the new values are processed. If unmapping was disabled, it might take additional 5 seconds until the first check is performed.
- You can query the values of these options using the system table ps_info_configuration (see section 26.3, page 312).

Disable Tracking of Access Times in File System

The file system keeps track of multiple statistics about its managed files. Among others, the last file access time is updated every time a file is being accessed. Cisco ParStream does not make use of this information, hence, we can improve file system performance by disabling the tracking of the access times. You can disable the tracking by adding noatime to the mount options in /etc/fstab.
An example would look like this:

```
/dev/hda1    /                        ext4    defaults,noatime         1 1
```

Separation Aware Execution

Cisco ParStream provides a couple of optimizations based on the awareness of locally or distributed separated data.

In general, these separation aware execution optimizations are enabled by the ExecTree option `SeparationAwareExecution` (see section 13.3.4, page 142).

If enabled in general, you can disable specific separation aware execution optimizations in case they are counter-productive with the following options in the `[ExecTree]` section:

Option	Effect	Page
`SeparationEnableDSGB`	Enable/disable Data Separated `GROUP BY` (DSGB)	186
`SeparationEnableHSGB`	Enable/disable Hash Separated `GROUP BY` (HSGB)	189
`SeparationEnableDSFA`	Enable/disable Data Separated Function Aggregations (DSFA)	190
`SeparationEnableDSJ`	Enable/disable Data Separated `JOIN` (DSJ)	191
`SeparationEnableHSJ`	Enable/disable Hash Separated `JOIN` (HSJ)	196
`SeparationEnableDSI`	Enable/disable Data Separated `IN` (DSI)	197

Several of these optimizations have distributed variants. In fact:

- For Data Separated `GROUP BY` (DSGB) you can also have Distributed Data Separated GROUP BY (DDSGB) (see section 15.15.1, page 189).
- For Hash Separated `GROUP BY` (HSGB) you can also have Distributed Hash Separated `GROUP BY` (DHSGB). (see section 15.15.2, page 189).
- For Data Separated JOIN (DSJ) you can also have Distributed Data Separated JOIN (DDSJ) (see section 15.15.4, page 194).

With the ExecTree option `SeparationEnableDHS` you can enable/disable all distributed hash separated optimizations, i.e. DHSGB.

Data Separated GROUP BY (DSGB and DDSGB)

Data Separated GROUP BY (DSGB) is an approach to optimize `GROUP BY` queries. The basic idea is to distribute the data of a distinct column into disjoint sets (partitions). When queries address this/these disjoint column(s) in a `GROUP BY` clause, you will gain a significant query speed up.

Assume we have a query like `SELECT ... GROUP BY` *`userid`*. Internally, we get an execution tree like in Figure 15.3. It is easy to see, that the upper aggregation stage does not scale.

When we organize the data in disjoint sets (see Figure 15.4), we are able to build a simpler execution tree without the bottleneck in the upper stage. In the upper stage, we only concatenate the results of the stages stated below.

The aim is to distribute the work of an aggregate node into independent nodes. Each node can accomplish its work in parallel to the other nodes. The result can easily be concatenated by an output node. Depending on the amount of data the parameter to determine the disjoint portions should be chosen reasonably. To achieve this you have to tell the import process how to build the disjoint data sets.

For example, guess we have a table with different user IDs:

```
CREATE TABLE MyTable (
```

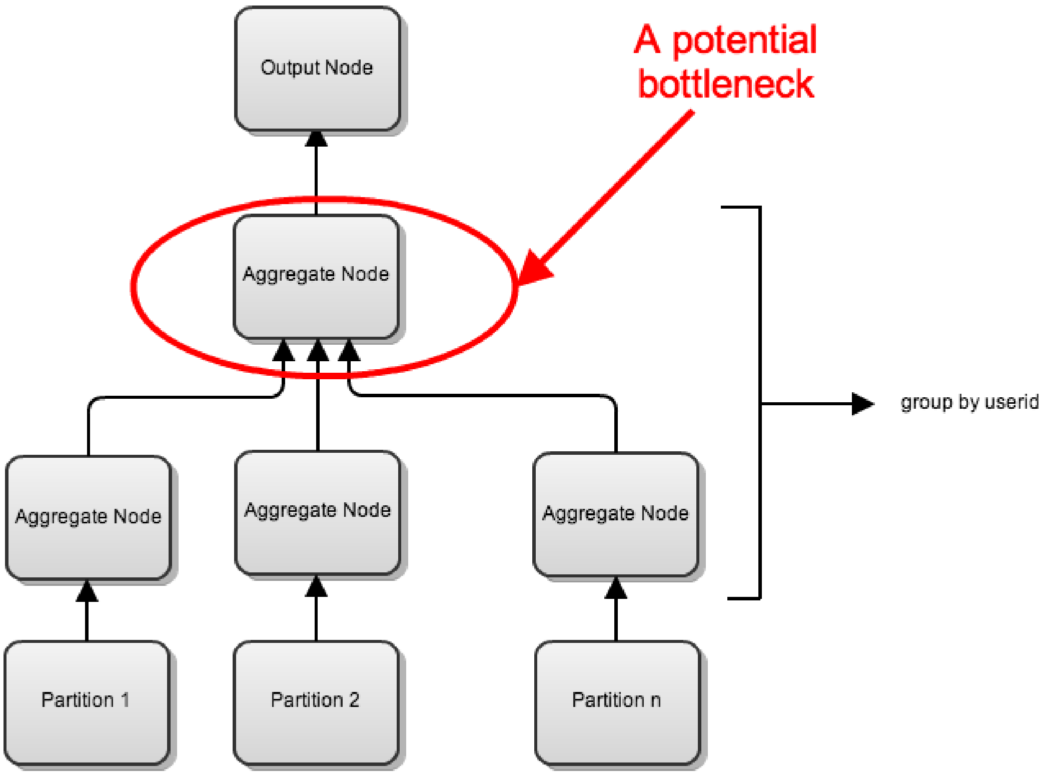


Figure 15.3: *Common aggregation execution*

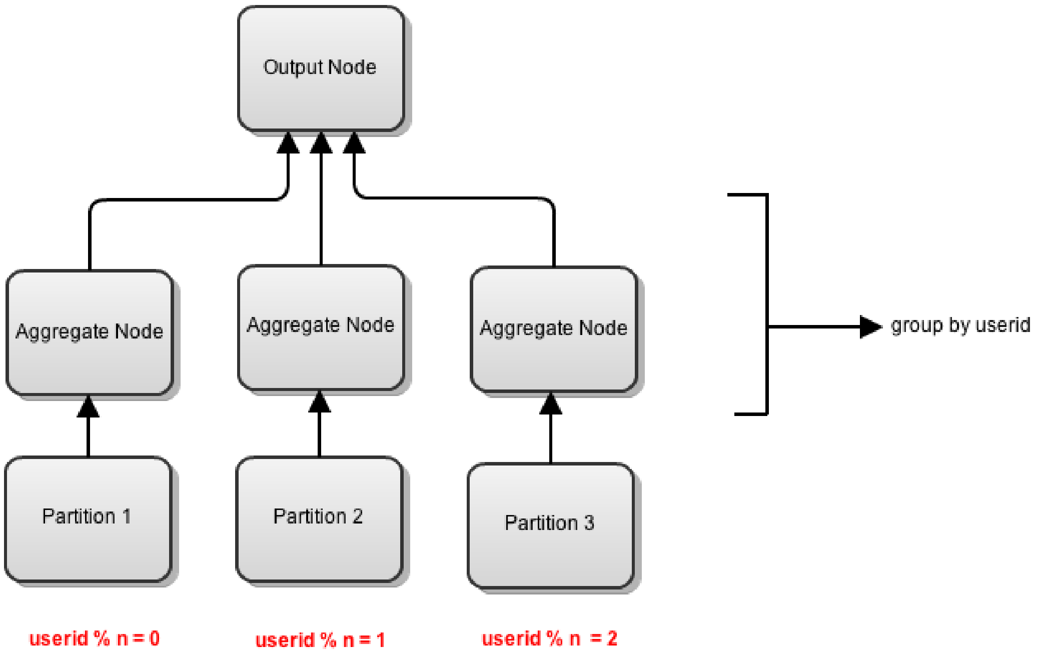


Figure 15.4: *Data separated execution*

```
  userid UINT32 ...
  ...
)
...
```

To build a disjoint limited set of data, we have to specify an ETL select statement distributing the `userid`s over 20 `usergroup`s: For example:

```
ETL ( SELECT userid MOD 20 AS usergroup
      FROM CSVFETCH(MyTable)
)
```

For this new ETL column we need a corresponding column, which we use to partition the data:

```
CREATE TABLE MyTable (
  userid UINT32 ...
  ...
  usergroup UINT16 INDEX EQUAL CSV_COLUMN ETL
)
PARTITION BY ... usergroup ...
```

The important point is to tell the database optimizer that queries to column `userid` can be optimized because they are distributed (separated) by the newly introduced column:

```
CREATE TABLE MyTable (
  userid UINT32 ... SEPARATE BY usergroup
  ...
  usergroup UINT16 INDEX EQUAL CSV_COLUMN ETL
)
PARTITION BY ... usergroup ...
```

Thus, to bring all together in the `CREATE TABLE` statement we need the following:

```
CREATE TABLE MyTable (
  userid UINT32 ... SEPARATE BY usergroup
  ...
  usergroup UINT16 INDEX EQUAL CSV_COLUMN ETL
)
PARTITION BY ... usergroup ...
...
ETL ( SELECT userid MOD 20 AS usergroup
      FROM CSVFETCH(MyTable)
);
```

Finally, we have to enable this optimization in general with the corresponding INI options:

```
[ExecTree]
SeparationAwareExecution = true
SeparationEnableDSGB = true
```

Distributed Data Separated GROUP BY (DDSGB)

For Data Separated GROUP BY there is also a variant provided, called *Distributed Data Separated GROUP BY* (DDSGB). Several of these optimizations have distributed variants. For DDSGB (Distributed Data Separated GROUP BY) you also have to distribute the data according to the `usergroup` column as described in section 6.3, page 53.

Hash Separated GROUP BY (HSGB and DHSGB)

Hash Separated GROUP BY (HSGB) is another approach to optimize `GROUP BY` queries. A query with a group by clause results internally in several aggregation stages. You can speed up these aggregations by building an internal mesh of parallelization. There is also a distributed variant of this optimization called *Distributed Hash Separated GROUP BY* (DHSGB).

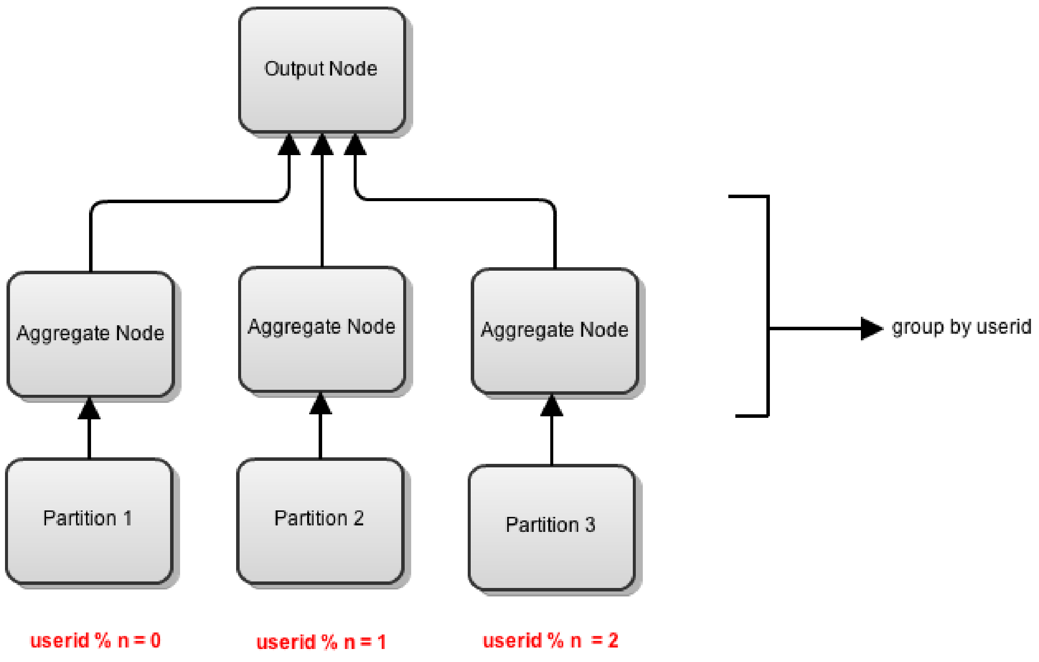


Figure 15.5: *Distributed Hash Separated GROUP BY (DHSGB)*

To enable the Hash Separated GROUP BY, the following INI parameters have to be set in the `[ExecTree]` section:

```
[ExecTree]
SeparationAwareExecution = true
SeparationEnableHSGB = true
```

This feature works on a cluster as well. To enable the support for DHSGB, the following INI file parameter must be set additionally:

```
[ExecTree]
SeparationEnableDHS = true
```

You can control the degree of parallelism per node used by (D)HSGB via another option in the [ExecTree] section:

```
[ExecTree]
NumHashSeparatedStreamsPerNode = 16
```

For example, the value 16 means to have 16-way-parallelism of inner aggregation stages on each participating cluster-node.
Finding the right value for NumHashSeparatedStreamsPerNode is not easy. Bear in mind that the speed up depends on many criteria, such as number of CPUs, number of cores in the CPUs, amount of memory, and so on. Setting this value to zero or below will yield an error.

With the global option dhsgbConnectionTimeout (see section 13.2.2, page 133) you can define the timeout when establishing inter cluster connections for DHSGB.

Data Separated Function Aggregation (DSFA)

A query with a DISTINCT clause or a DISTVALUES function can be parallelized even without a GROUP BY clause. Cisco ParStream leverages the information given by the data separation on a column to reduce the number of comparisons necessary to get a distinct set of values. The only prerequisites to enable this feature are that the column in the DISTINCT or DISTVALUES is either a partitioning column or separated by a partitioning column, and that DSFA is enabled in the INI configuration:

```
[ExecTree]
SeparationAwareExecution = true
SeparationEnableDSFA = true
```

Then, with

```
SELECT DISTINCT city FROM tabA;
```

we assume that city is separated by a partitioning column. Hence, each partition contains a set of cities that is disjoint to the sets in the other partitions. Therefore, Cisco ParStream calculates all distinct values per partition and then concatenates the results of all partitions to get a final set of distinct values.
And with

```
SELECT COUNT(DISTINCT city) FROM tabA;
```

the procedure is identical to the example above. The sole exception is that instead of concatenating the sets of disjoint values, we sum up the different number of distinct values.

This feature can be extended to work in a cluster if the partitioning column is also the distribution column. Then, Cisco ParStream can calculate the distinct values on each server independently and concatenate the result as a final step.

JOIN Parallelization

Cisco ParStream applies different parallelization strategies for JOIN queries. These strategies enable the optimizer to scale join nodes horizontally to achieve an optimal query performance. The strategies applied are:

- Data Separated JOIN (DSJ, see page 191)
- Distributed Data Separated JOIN (DDSJ, see page 194)
- Hash Separated JOIN (HSJ, see page 196)

The optimizer always tries to leverage DSJs if possible and falls back to HSJs if DSJ is not applicable. In a multi-stage join both of these strategies may appear in combination, because the optimizer will use DSJs as long as possible and fall back to HSJs if DSJ can't be applied anymore.

Figure 15.6 shows how the optimizer decides which parallelization strategy to use.

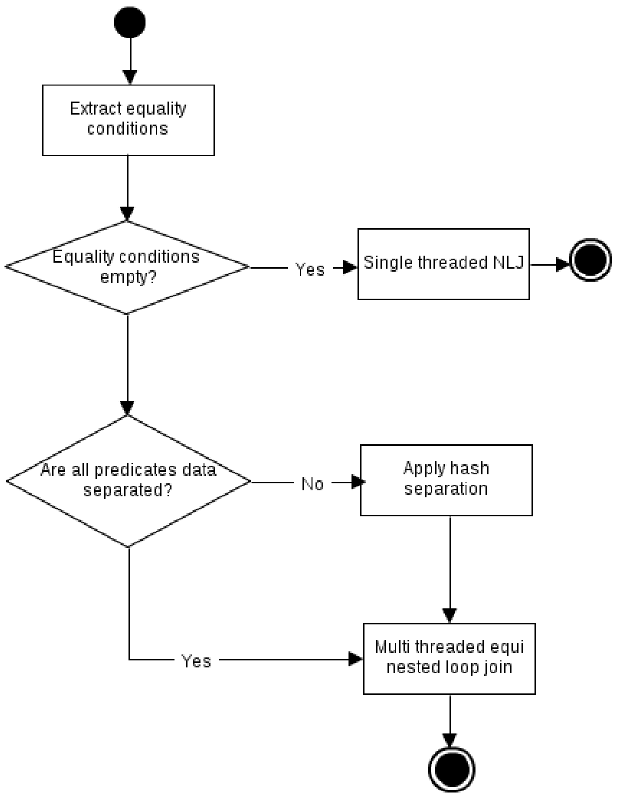

Extract equality conditions
Equality conditions empty?
Yes
Single threaded NLJ
Are all predicates data separated?
No
Apply hash separation
Yes
Multi threaded equi nested loop join

Figure 15.6: *Decision flow for parallelization strategies*

Data Separated JOIN (DSJ)

A JOIN can exploit Data Separation. This enables the optimizer to scale the `JOIN` horizontally and join complete partitions pairwise. This feature can be applied when joining:

- two partitioning columns, or
- two columns that are separated by another column, which splits these columns virtually into disjoint parts (see section 15.15.4, page 192 for details).

To enable Data Separated JOINs the ExecTree options SeparationAwareExecution (see section 13.3.4, page 142) and SeparationEnableDSJ must be set in the [ExecTree] section:

```
[ExecTree]
SeparationAwareExecution = true
SeparationEnableDSJ = true
```

Tables are implicitly separated by their respective partitioning column. Assume we have two tables tabA and tabB, both tables are partitioned by the respective id column. Executing the following query will result in the execution tree shown in figure 15.7 (we assume that both tables have the values 1 and 2 for their individual id column):

```
SELECT * FROM tabA
       INNER JOIN tabB ON tabA.id = tabB.id;
```

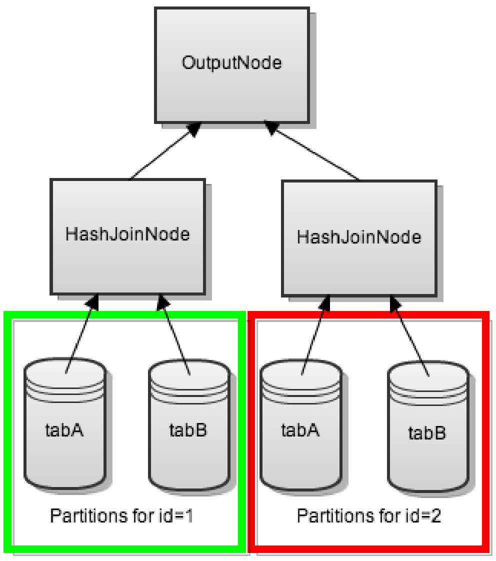


Figure 15.7: *Example of a Separated Join with partitions for column id (in both tables column id has the values 1 and 2, respectively)*

To exploit Data Separation for columns that are separated by another column (see section 15.15.1, page 188 for further details about columns separated by another column) you have to specify a hint for the optimizer. For this, Cisco ParStream introduces a new column property called “**REFERENCES**” (see section 24.2.4, page 284), which allows to exploit a defined Data Separated JOINs similarly to DSGB (see section 15.15.1, page 186). The new property allows to define that a column in one table is separated by the same function as another column in another table.

For example: Assume you have two tables tabA and tabB containing user ids. Due to the high cardinality of your user ids assume further, that you want to group user ids by a column user_group which is created by an ETL statement during import. The function you may use to create the column user_group is user_id % 20 to generate 20 groups. These 20 groups contain disjunct sets of user_ids. The function user_id % 20 is utilized by tabA and tabB in order to create their respecting user_group column. Now assume you issue the following query:

```
SELECT * FROM tabA INNER JOIN tabB ON tabA.user_id = tabB.user_id;
```

If the optimizer knew that both user_id columns fall into 20 disjunct groups, a DSJ could be executed directly on the user_group column.

We give an example why here a DSJ is applicable: Assume there is a user_group with the value 1 for tabA which has no counterpart in tabB. Due to the fact that the column user_id falls into disjunct subsets we know that there exist no matching user_ids in this subset for tabA and tabB. This allows the optimizer to exclude complete partitions without the need to match the individual user_ids within the partitions. In order to exploit this knowledge the optimizer needs a hint. This hint is given by the "REFERENCES" keyword telling the optimizer that both user_id columns are separated by the same function. It is, due to the rule of transitivity, sufficient if you give this hint in one table definition. For the sake of this example we'll show the relevant parts of tabA's table definition:

```
CREATE TABLE tabA
(
  user_id UINT64 SINGLE_VALUE SEPARATE BY user_group REFERENCES tabB.user_id
    INDEX EQUAL,
  ...
  user_group UINT64 SINGLE_VALUE INDEX EQUAL CSV_COLUMN ETL
)
PARTITION BY user_group,
...
ETL (SELECT user_id MOD 20 AS user_group FROM CSVFETCH(tabA));
```

Hence, Cisco ParStream can use the existing data separation to parallelize the join locally into multiple hash joins. Otherwise, Cisco ParStream would have to separate the values manually by a pre-hashing phase.

Please note:

- Referenced column have to be defined as NOT NULL and UNIQUE.
- You can also reference a table as a whole (i.e. REFERENCES tabB). In that case, the referenced column is the column marked as PRIMARY KEY in the referenced table (see section 24.2.4, page 284).
- If tabA.col1 REFERENCES tabB.col2, every value of tabA.col1 has to occur somewhere in tabB.col2 unless it is NULL. This constraint is not validated by Cisco ParStream, valid data has to be guaranteed by the customer.
- REFERENCES is a transitive property and cycles will lead to an error.
- A wrong REFERENCES configuration may lead to wrong query results.
- Separation aware execution has to be enabled for this feature to work (see section 13.3.4, page 140).

Distributed Data Separated JOIN (DDSJ)

Cisco ParStream currently only supports Distributed Data Separated Joins. Locally on each cluster member, Cisco ParStream uses a Data Separated Join (DSJ) to compute partial solutions of the query. Due to data separation, Cisco ParStream can simply concatenate the results, locally computed, on the query master to create the final result.

If Cisco ParStream cannot exploit data separation to parallelize a join across cluster members, Cisco ParStream reverts back to a join on the query master. That means, Cisco ParStream ships all data to the query master and computes the result locally. In this case, Cisco ParStream may still apply a hash separation to use a hash join, but using data separation and computing the join on all cluster members in parallel will yield better performance.

To use a Distributed Data Separated Join, a few conditions have to be fulfilled:

- The join columns are either distribution columns, or they are separated by one of the distribution columns (see section 6.3, page 53) using the same function (see section 15.15.4, page 192).
- One of the tables is distributed across the cluster. The other tables are either replicated on every node using distribution everywhere (see section 6.3.1, page 58), or they are collocated with the distributed table (see section 6.3.1, page 54).
- All requirements from Data Separated Join (DSJ). See section 15.15.4, page 191.

Figure 15.8 shows how the optimizer decides how to distribute a join query.

By the rules above, Cisco ParStream can conduct a join within a cluster in three different ways:

- **Rule 1:** Assume columns `tabA.a` and `tabB.b` are collocated. Then, for example, the following query:

  ```
  SELECT * FROM tabA
           INNER JOIN tabB ON tabA.a = tabB.b;
  ```

  would result in a distributed join, which is executed on each node individually. This is shown in figure 15.9.
- **Rule 2:** Assume columns a and b are collocated and a third column c is either collocated to column a or b, or fully distributed.
  For example, the following query:

  ```
  SELECT * FROM tabA
           INNER JOIN tabB ON tabA.a = tabB.b
           INNER JOIN tabC ON tabB.b = tabC.c;
  ```

  would result in a distributed join, which is executed on each node individually: This is shown in figure 15.10.
- **Rule 3:** If the data sets are neither collocated nor at least one of the tables is fully distributed, concurrent execution of the join across a cluster is not possible. Hence, all work must be done by the query master after collecting all relevant data.
  In this case, for example, the following query:

  ```
  SELECT * FROM tabA
           INNER JOIN tabB ON tabA.a = tabB.b;
  ```

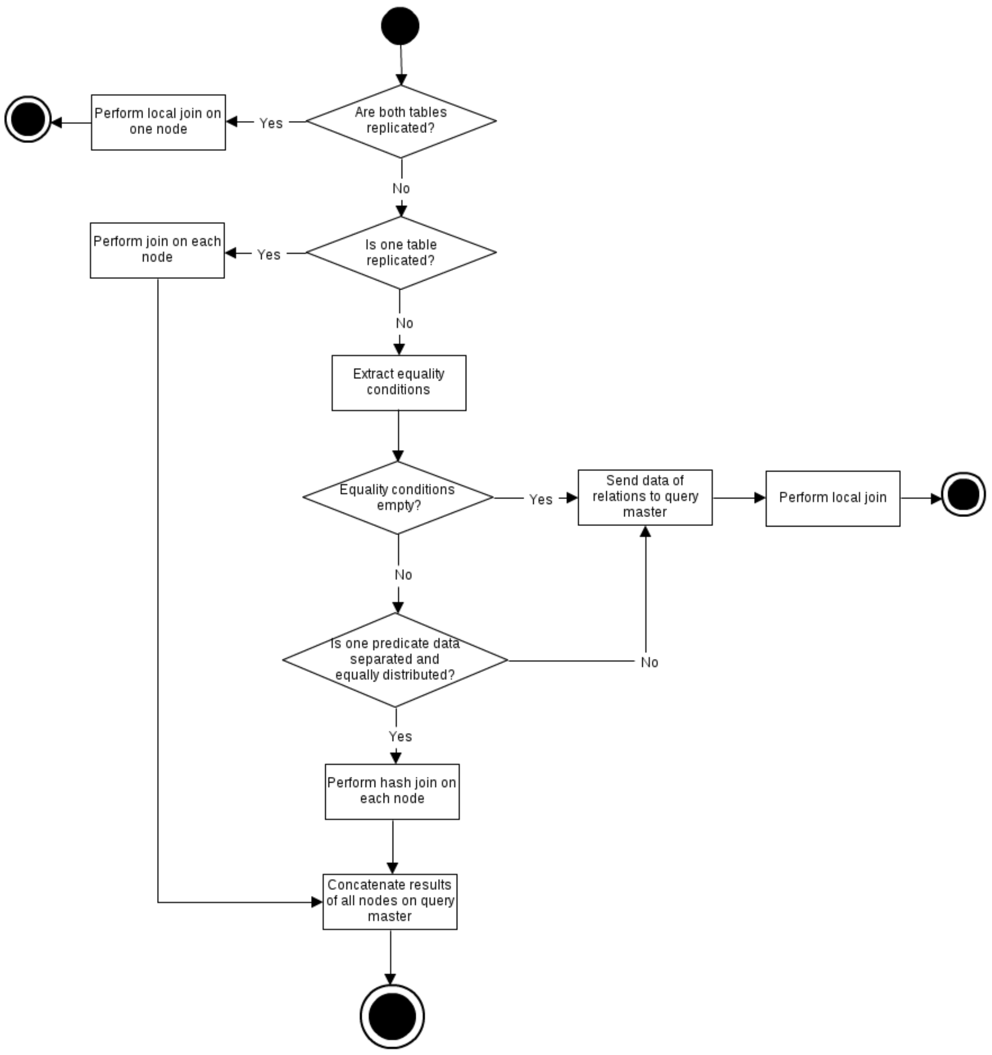

Are both tables replicated?
Yes
Perform local join on one node
No
Is one table replicated?
Yes
Perform join on each node
No
Extract equality conditions
Equality conditions empty?
Yes
Send data of relations to query master
Perform local join
No
Is one predicate data separated and equally distributed?
No
Yes
Perform hash join on each node
Concatenate results of all nodes on query master

Figure 15.8: *The optimizer's activities to decide the distribution strategy*

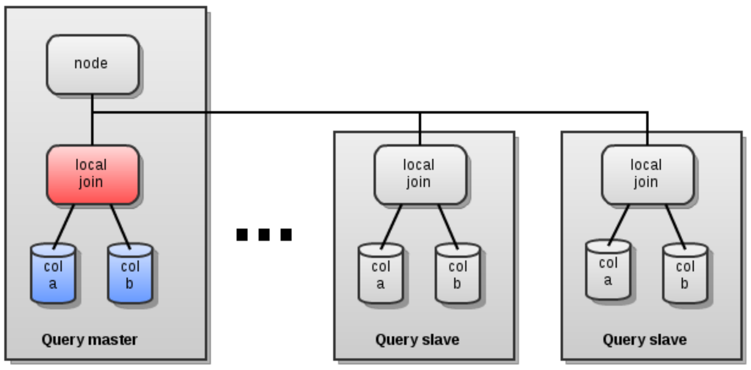


Figure 15.9: *Columns a and b are collocated which leads to a distributed execution of the join on each cluster member.*

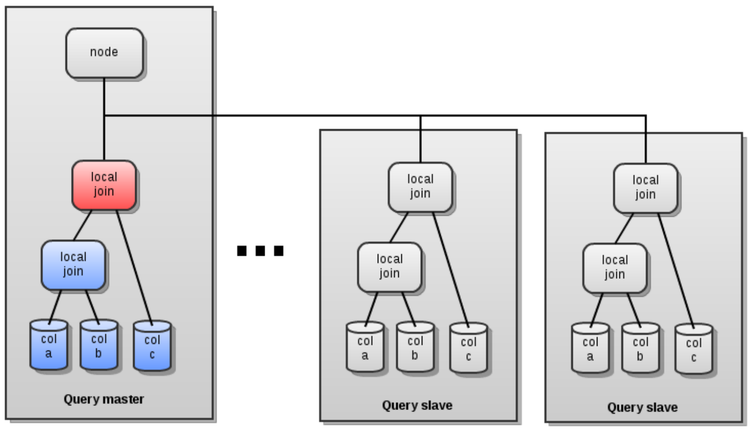


Figure 15.10: *Columns a and b are collocated, column c is either collocated to column a or b, or fully distributed, which leads to a distributed execution of the join on each cluster member.*

will result into a situation as shows in figure 15.11.

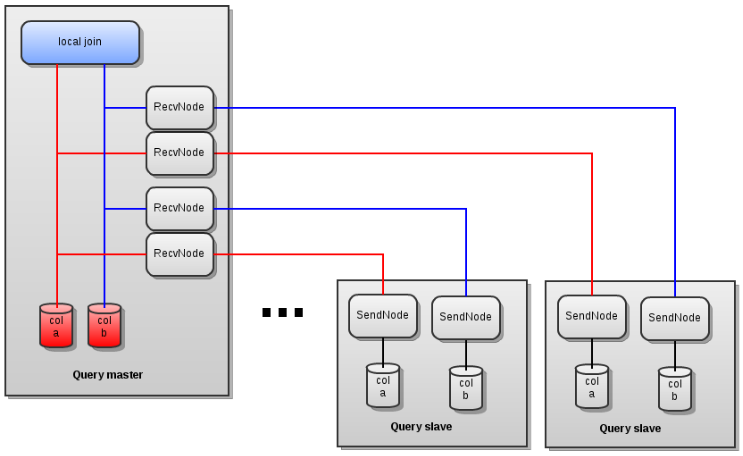


Figure 15.11: *Columns a and b are not collocated, neither is one of the tables fully distributed. This leads to a local join on the query master.*

Hash Separated Join (HSJ)

If a Data Separated `JOIN` (see section 15.15.4, page 191) is not applicable, the optimizer will fall back to a Hash Separated `JOIN` (HSJ). HSJ is applied if the columns within the join condition are not separated by the same logical attribute. If this is the case the optimizer does not know how the data is distributed. Still the optimizer is capable to scale the `JOIN` horizontally by adding HashSeparatingNodes. This layer distributes data, based on its hash value, to the layer of join nodes above. This eventually allows the join to be scaled. The support for HSJ can be enabled by the following options:

```
[ExecTree]
SeparationAwareExecution = true
```

```
SeparationEnableHSJ = true
```

The number of parent nodes of HashSeparatingNodes is determined by the `[ExecTree]` option `NumHashSeparatedStreamsPerNode` (see section 13.3.4, page 143):

```
[ExecTree]
NumHashSeparatedStreamsPerNode = 16
```

By this configuration entry, 16 HashJoinNodes will be created above 32 HashSeparatingNodes.[2]

Given the following query:

```
SELECT * FROM tabA INNER JOIN tabB ON tabA.id = tabB.id;
```

which joins over two columns not separated by the same logical attribute, and `NumHashSeparatedStreamsPerNode` set to 3, will result in the execution tree shown in figure 15.12.

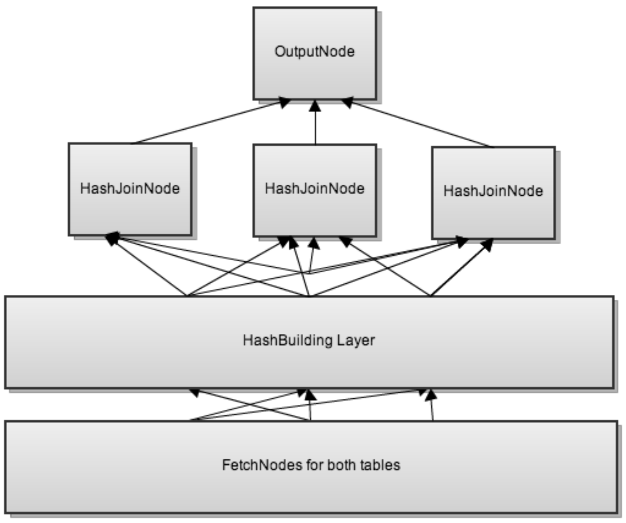


Figure 15.12: *Simplified example of a JOIN query where the optimizer decided to use hash separation.*

Data Separated IN (DSI)

We can leverage data separation to calculate a Data Separated `IN` (DSI) for queries like in the following example:

```
SELECT * FROM tabA WHERE tabA.userId IN (SELECT id FROM tabB);
```

The prerequisite to perform a Data Separated `IN` (DSI) is that the columns used in the IN, in the example “tabA.userId” and “tabB.id”, are:

- two partitioning columns, or

[2] For each JoinNode two HashSeparatingNodes are created: One for the join’s LHS, and one for the join’s RHS children.

- two columns that are separated by another column, which splits these columns virtually into disjoint parts. See section 15.15.4, page 192 for a detailed explanation.

In this case, Cisco ParStream calculates all relevant keys on a partition level to reduce the size of unnecessary filter statements and to retrieve only relevant data.

You can enable the support for Data Separated `IN` (DSI) by using the following ini configuration:

```
[ExecTree]
SeparationAwareExecution = true
SeparationEnableDSI = true
```

If you operate a cluster with multiple nodes, the Data Separated `IN` can only be applied if the two columns involved in the `IN` are co-located, i.e., they are either columns used to distribute the table over the cluster or they are separated by distribution columns.

If the Data Separated `IN` is disabled, the only option to use an `IN` statement involving two tables is to have one table replicated on every cluster node. Otherwise, the `IN` query will not be accepted and return an error message.

Socket Client Interface

Client applications can connect to a Cisco ParStream database via a plain character-based socket interface. Various utilities and tools can utilize this interface.

The socket interface is realized over a TCP/IP connection. All data is transmitted as UTF-8 encoded text, there is no binary transmission of data. Simultaneous connections are processed within a server in parallel, using threads. Multiple requests within the same connection are processed sequentially.

Each request consists of one line and is terminated either with a zero-byte or a client's ("\n"). Every line of a result set is terminated with a LF ("\n"). The entire result set is terminated with an empty line followed by a zero-byte.

Security

For a user to authenticate with a Cisco ParStream server via the netcat interface, he needs to issue a login command using valid login credentials after connecting to Cisco ParStream:

```
login '<username>' '<pass phrase>'
```

Tooling

Users can interact with Cisco ParStream database over the socket interface using these tools:

1. **nc** (netcat): a standard socket interface client (see section 12.1.2, page 113).

2. **pnc**: an improved socket interface client provided by Cisco ParStream (see section 12.1.1, page 110).

In addition to SQL conforming commands, Cisco ParStream also accepts additional control commands sent through the netcat interface that read or alter system parameters. See section 16.4, page 200 for details.

SQL commands can also sent to Cisco ParStream with the **psql** console client using the PostgreSQL compatible protocol. See section 12.2, page 115 for details.

Output Format

Cisco ParStream can have different interface formats. The default format, **`ASCII`**, returns the result of queries as plain ASCII data. In addition, you can use

- **`JSON`**: The output format is JSON (See section 16.7, page 207 for details).
- **`XML`**: The output format is XML (See section 16.6, page 205 for details).

The output format can be set in the configuration file as global option `outputformat` (see section 13.2.1, page 121):

```
# global
outputformat = JSON
```

or by sending a `SET` command to the server (see section 27.10.1, page 373):

```
SET OUTPUTFORMAT = 'JSON';
```

or unquoted

```
SET OUTPUTFORMAT = JSON;
```

See section 23.5.1, page 273 for the effect of the output format on `NULL`/empty strings.

See section 16.6, page 205 for details of the XML format.
See section 16.7, page 207 for details of the JSON format.

Control Commands

Beside the usual SQL statements, the socket interface provides a number of commands to control a Cisco ParStream database installation. Some of them are just provided for internal use and might only be helpful to fix something on a low-level base.

Cluster Control Commands

To control the cluster as a whole, a number of commands are provided, which you can issue by using the `ALTER SYSTEM` command (see section 27.11, page 375):

`ALTER SYSTEM` **`CLUSTER SHUTDOWN`**
`ALTER SYSTEM` **`NODE SHUTDOWN`**
`ALTER SYSTEM` **`NODE [`*`nodename`*`] SHUTDOWN`**

> Shuts down all a specific node/server or all nodes or of a cluster (including importer nodes). Active synchronization or import tasks are allowed to finish, but no new such tasks will be allowed to start. If the optional parameter *nodename* is not passed, the command will shut down the node/server the query is issued on. The “`ALTER SYSTEM NODE SHUTDOWN`” command can be used to shut down any server/node (independent from whether it is part of a cluster).

`ALTER SYSTEM` **`CLUSTER DISABLE IMPORT`**
`ALTER SYSTEM` **`CLUSTER ENABLE IMPORT`**

> Enable or disable data import operations for the cluster. When data imports are disabled the cluster will not accept new streaming import connections or import with the Cisco ParStream importer. Active data imports will be allowed to finish.
>
> The import status can be queried via the system table `ps_info_cluster_node` (see section 26.4, page 317).

`ALTER SYSTEM` **`CLUSTER DISABLE MERGE`**
`ALTER SYSTEM` **`CLUSTER ENABLE MERGE`**

Enable or disable the execution of scheduled merges. If merges are being disabled, currently running merges will run until completed, but no new merges will be executed.

The merge status can be queried via the system table `ps_info_cluster_node` (see section 26.4, page 317).

`ALTER SYSTEM` **`CLUSTER EXECUTE MERGE`** `LEVEL (HOUR | DAY | WEEK | MONTH)`

Force execution of the specified merge. The merge will be executed immediately, even if merges are currently disabled.

Additional Internal Commands

The following command have effect only on a single-node cluster. Note that in clusters with multiple servers/nodes corresponding cluster commands should be preferred if provided.

`help` [*request*]

This requests returns a description of the available requests. If the optional parameter is set to the name of a request, then a detailed description of that request is returned.

`INSPECT` *subject*

- Outputs tabular information about the given subject without using the execution engine (i.e. unlike `SELECT` based system table queries)
- See section 27.6, page 362 for details.

`LOGIN` *’username’ ’pass phrase’*

Tries to authenticate with the provided user name and pass phrase. If user authentication is enabled (see section 9.2, page 78), the user can issue queries after a successful authentication with this command.

Note that the pass phrase is currently sent without encryption, so that it’s up to the system or network administrator to ensure a secure transport over the network if this is required.

`quit`

Closes the connection to the server and frees all resources held by that connection. This request is used mostly for automated tests; to terminate a script that contains various requests in a clean way.

`SET` *command*

- Sets session-specific values.
- See section 27.10, page 373 for details.

`showexectree` *command*

- The command `showexectree` can be used by the socket interface to display the execution tree into which SQL statements are translated.
- For example: A request such as follows:

  ```
  showexectree SELECT * FROM Address;
  ```

  might output:

  ```
  parstream::ExecOutputNode(0x7f31a400b300)
  input
      Address.__street_value(offset=0,size=16,varstring,single_value,hashed
      value,column=Address.street), ...
   queue(of 0x7f31a400b300, size 0)
   parstream::ExecTransNodeRecvTCP(0x7f31a4006d30)
   output
      Address.__street_value(offset=0,size=16,varstring,single_value,hashed
      value,column=Address.street), ...
   parstream::ExecTransNodeRecvTCP(0x7f31a4007fd0)
   output
      Address.__street_value(offset=0,size=16,varstring,single_value,hashed
      value,column=Address.street), ...
   parstream::ExecFetchNodeColumnData(0x7f31a400ad30)
   output
      Address.__street_value(offset=0,size=16,varstring,single_value,hashed
      value,column=Address.street), ...
  output Address.id(offset=64,size=4,int32,single_value,not
      hashed,column=Address.id), ...
  ```

`ALTER SYSTEM NODE SHUTDOWN`

Tells the local server to stop accepting requests, and to shutdown the next time a connection to the server is opened — after all pending queries and imports have been processed. In multi-node clusters the corresponding cluster control command should be used (see section 16.4.1, page 200).

`sqlprint` *command*

- The command `sqlprint` can be used by the socket interface to display the description tree in different processing stages (after parsing, after meta data enrichment, after optimizations) into which SQL statements are translated.
- For example: A request such as follows:

```
sqlprint SELECT AVG(value) AS val FROM testtable;
```

might output:

```
Description tree:
OutputNode requiredOutputRows: none  fields: (val) uniqueNodeId: 3 limit:
    none offset: 0
 AggregateNode requiredOutputRows: none  fields: (val) uniqueNodeId: 2
    aggregate fields: val:=AVG(value) group by:  level: 0/2
  FetchNode requiredOutputRows: none [parallelizable] fields: (value)
    uniqueNodeId: 1 condition:  table: testtable

Preprocessed tree:
OutputNode requiredOutputRows: none  fields: (val) uniqueNodeId: 3 limit:
    none offset: 0
 AggregateNode requiredOutputRows: none  fields: (val) uniqueNodeId: 6
    aggregate fields: val:=AVGCOUNTSUM(val) group by:  level: 0/2
  AggregateNode requiredOutputRows: none  [optional] fields: (val,
    val__CNT_INTERN) uniqueNodeId: 5 aggregate fields:
    val__CNT_INTERN:=SUM(val__CNT_INTERN), val:=SUM(val) group by:  level:
    1/2 [cascadable]
   AggregateNode requiredOutputRows: none [parallelizable] fields: (val,
    val__CNT_INTERN) uniqueNodeId: 4 aggregate fields:
    val__CNT_INTERN:=COUNT(value), val:=SUM(value) group by:  level: 2/2
    FetchNode requiredOutputRows: none [parallelizable] fields: (value)
    uniqueNodeId: 1 condition:  table: testtable

Parametrized tree:
|->[val(offset=0,size=8,double,single_value,not hashed,nocolumn)]:rowSize=0
ParametrizedNode parstream::OutputNode NoRemoteProcessing no separation
    possible no sorting partition-overlapping
 |->[val(offset=0,size=8,double,single_value,not
    hashed,nocolumn)]:rowSize=8
 ParametrizedNode parstream::AggregateNode NoRemoteProcessing no
    separation possible no sorting partition-overlapping
  |->[val(offset=0,size=8,uint64,single_value,not hashed,nocolumn),
    val__CNT_INTERN(offset=8,size=8,uint64,single_value,not
    hashed,nocolumn)]:rowSize=16
  ParametrizedNode parstream::AggregateNode NoRemoteProcessing no
    separation possible no sorting partition-overlapping
   |->[val(offset=0,size=8,uint64,single_value,not hashed,nocolumn),
    val__CNT_INTERN(offset=8,size=8,uint64,single_value,not
    hashed,nocolumn)]:rowSize=16
   ParametrizedNode parstream::AggregateNode NoRemoteProcessing no
    separation possible no sorting partition-local
    |->[value(offset=0,size=8,uint64,single_value,not
    hashed,column=testtable.value)]:rowSize=8
    ParametrizedNode parstream::FetchNode NoRemoteProcessing no separation
    possible no sorting partition-local

Optimized tree:
|->[val(offset=0,size=8,double,single_value,not hashed,nocolumn)]:rowSize=0
ParametrizedNode parstream::OutputNode NoRemoteProcessing no separation
    possible no sorting partition-overlapping
  |->[val(offset=0,size=8,double,single_value,not
     hashed,nocolumn)]:rowSize=8
 ParametrizedNode parstream::AggregateNode NoRemoteProcessing no
     separation possible no sorting partition-overlapping
   |->[val(offset=0,size=8,uint64,single_value,not hashed,nocolumn),
     val__CNT_INTERN(offset=8,size=8,uint64,single_value,not
     hashed,nocolumn)]:rowSize=16
...
```

- Note that the output format is not standardized and might change with any new Cisco ParStream version.

`unload` *partition tablename*

Marks the specified partition as disabled. (partition status “disabled-by-unload”, see section 5.1.3, page 34). The partition will subsequently be ignored at server startup and by load requests.

- *partition* is the path of the partition’s directory under the partition directory (usually `datadir`).
- *tablename* is the name of the table the partition belongs to.

The return value is the number of unloaded partitions which should be 1 if the partition was unloaded and 0 if unloading failed.

For example:

```
    unload 3875/20100120_Z_2010-08-17T09:03:39_first_PM MyTable
unload "14/2010-11-14 14:00:14Z_2013-09-17T20:23:24_first_PM"
MyTable
```

Note:
Unloading a partition is not reversible and will physically delete the data of the partition!

`load` *directory tablename*

Scans the specified directory (must be a sub-directory of the server’s `datadir`) for partitions and loads them. The partitions have to be created with a standalone import. Loading partitions that are not activated is rejected. The return value is the number of loaded partitions.

For example, if inside the partition directory of the server is a partition for the value 835:

```
    load 835 MyTable
```

ASCII Interface

In the `ASCII` setting the output by default is output in a csv like syntax:

Input:

```
SELECT * FROM Wages WHERE wage_id = 47;
```

Output:

```
#wage_id;education;address;sex;experience;union;wage;age;race;occupation;sector;marr
47;10;5;0;13;0;4.85;29;3;6;0;0
```

The column separator, separator for entries in multi-values, and the NULL representation can be customized with the options `asciiOutputColumnSeparator` (see section 13.2.1, page 121), `asciiOutputMultiValueSeparator` (see section 13.2.1, page 122), and `asciiOutputNullRepresentation` (see section 13.2.1, page 122), respectively.

Error Handling

In the case of an error an error code and a human-readable error message is returned. The line that contains the answer for a request starts with "#ERROR", followed by '-', and an error number, if an error has occurred. The error code is followed by the error message.

For example:

```
#ERROR-000: Internal Error

#ERROR-100: Invalid Query

#ERROR-110: Unknown Request

#ERROR-120: Unknown Column

#ERROR-130: Insufficient Number of Parameters

#ERROR-200: No Data Found
```

XML Interface

The data can also be transmitted packed inside XML.

Input:

```
SELECT * FROM Wages WHERE wage_id = 47;
```

Output:

```
<output>
  <dataset>
    <wage_id>47</wage_id>
    <education>10</education>
    <address>5</address>
    <sex>0</sex>
    <experience>13</experience>
    <union>0</union>
    <wage>4.85</wage>
    <age>29</age>
    <race>3</race>
    <occupation>6</occupation>
    <sector>0</sector>
    <marr>0</marr>
  </dataset>
</output>
```

Input:

```
SELECT wage_id FROM Wages WHERE wage_id > 47 LIMIT 5;
```

Output:

```
<output>
  <dataset>
    <wage_id>48</wage_id>
  </dataset>
  <dataset>
    <wage_id>51</wage_id>
  </dataset>
  <dataset>
    <wage_id>58</wage_id>
  </dataset>
  <dataset>
    <wage_id>61</wage_id>
  </dataset>
  <dataset>
    <wage_id>64</wage_id>
  </dataset>
</output>
```

This can be checked with:

```
SELECT count(*) FROM Wages WHERE wage_id > 47;
```

Output:

```
<output>
  <dataset>
    <auto_alias_1__>486</auto_alias_1__>
  </dataset>
</output>
```

Here you don't get a real data row, but the internal count of Cisco ParStream back.

Error Handling

Error messages in the XML output format are textually the same as for the ASCII output format, but formatted as XML with a single `error` element:

```
<error>ERROR-000: Internal Error</error>

<error>ERROR-100: Invalid Query</error>

<error>ERROR-110: Unknown Request</error>

<error>ERROR-120: Unknown Column</error>

<error>ERROR-130: Insufficient Number of Parameters</error>

<error>ERROR-200: No Data Found</error>
```

JSON Interface

The data can also be transmitted packed inside JSON.

Input:

```
SELECT * FROM Wages WHERE wage_id = 47;
```

Output:

```
{"rows":
  [
    {
      "wage_id":47,
      "education":10,
      "sex":0,
      "experience":13,
      "union":0,
      "wage":4.85,
      "age":29,
      "race":3,
      "occupation":6,
      "sector":0,
      "marr":0
    }
  ]
}
```

Input:

```
SELECT wage_id FROM Wages WHERE wage_id > 47 LIMIT 5;
```

Output:

```
{"rows":
  [
    {"wage_id":48},
    {"wage_id":51},
    {"wage_id":58},
    {"wage_id":61},
    {"wage_id":64}
  ]
}
```

This can be checked with:

```
SELECT count(*) FROM Wages WHERE wage_id > 47;
```

Output:

```
{"rows":[{"auto_alias_1__":486}]}
```

Here you don't get a real data row, but the internal count of Cisco ParStream back.

Error Handling

Error messages in the JSON output format are textually the same as for the ASCII output format, but formatted as JSON dictionary with a single entry `error`:

```
{"error" : "ERROR-000: Internal Error"}

{"error" : "ERROR-100: Invalid Query"}

{"error" : "ERROR-110: Unknown Request"}

{"error" : "ERROR-120: Unknown Column"}

{"error" : "ERROR-130: Insufficient Number of Parameters"}

{"error" : "ERROR-200: No Data Found"}
```

ODBC Client Interface

Cisco ParStream relies on PostgreSQL ODBC driver psqlODBC (v11.00.0000) to provide ODBC client connectivity to a Cisco ParStream database.

The Cisco ParStream server port for client ODBC connections is 1 higher than the basic port of the corresponding Cisco ParStream server. For example, if `port=9042`, the ODBC connection port is `9043`. See section 13.3.1, page 135 for details.

ODBC Configuration Brief

The content of this section is general ODBC information provided as a courtesy. Cisco ParStream specific material is limited to the content of the Cisco ParStream DSN.

ODBC Connection String

A standard ODBC connection string is a semicolon separated list of connection attribute/value pairs, for example:

```
"DSN=demo;SERVERNAME=10.10.20.99;PORT=9043;DATABASE=demo;
   Username=parstream;Password=n-a;"
```

Note:

Per ODBC specification, connection attribute names are *case insensitive*.

Data Source Name (DSN)

DSN is a named tag in a *ODBC.INI* file that lists a set of connection attributes. Here is an example of a Cisco ParStream DSN "Demo":

```
[demo]
Driver     = /usr/lib64/psqlodbc.so
Servername = 10.10.20.99
Port       = 9043
Database   = demo
Username   = parstream
Password   = n-a
```

If a connection string includes DSN=*DSN_NAME* attribute, the ODBC driver (or Driver Manager) builds the actual connection string by combining all attributes listed under [*DSN_NAME*].

A DSN listed in the *System ODBC.INI* file would be a **System DSN**, accessible by all users of the host OS.

A DSN listed in the *User ODBC.INI* file would be a **User DSN**, accessible only by the given user of the host OS.

A given database may have any number of DSNs pointing to it.

It is legitimate to have the same *System* and *User DSN* at the same time. Users should use caution - in this case *User DSN* may "cast shade" on *System DSN* which may lead to inadvertent confusions. In case of a conflict (same connection attribute defined with different values) the order of precedence is as follows:

- System DSN
- User DSN
- Connection String

I.e. *User DSN* settings overwrites *System DSN* settings, and attributes explicitly specified in the connection string overwrite either *User* or *System DSN*.

Note:

DSN is not required to make a database connection. A client can specify all connection attributes on-the-fly in the connection string. However a DSN provides convenience and connection attributes consistency.

Installing ODBC Driver on Linux

The installation instructions below are for RHEL/CentOS/Oracle Linux. Adjust the instructions accordingly to install the driver on other Linux OS variants.

As user *root*, install the psqlODBC driver (`postgresql-odbc` rpm) and unixODBC Driver Manager (`unixODBC` rpm). Note that `postgresql-odbc` RPM package pulls in `unixODBC` package as dependency.

```
$ yum install postgresql-odbc
```

If you are planning to write custom C/C++ code using ODBC interface, you'll need to install an optional `unixODBC-devel` RPM package that includes ODBC API include files (e.g. `sql.h`, `sqlext.h`, `sqltypes.h`, etc.):

```
$ yum install unixODBC-devel
```

Verify unixODBC default configuration:

```
$ odbcinst -v -j
unixODBC x.y.z
DRIVERS............: /etc/odbcinst.ini
SYSTEM DATA SOURCES: /etc/odbc.ini
FILE DATA SOURCES..: /etc/ODBCDataSources
USER DATA SOURCES..: /home/parstream/.odbc.ini
...
```

Confirm that the SYSTEM DSN is `/etc/odbc.ini` and USER DSN is `$HOME/.odbc.ini`.

Configuring Cisco ParStream ODBC Connection on Linux

psqlODBC driver relies on a ODBC Driver Manager to read *System* or *User DSN*. With unixODBC the default locations of the ODBC.INI files are as follows:

- *System DSN* file is `/etc/odbc.ini`
- *User DSN* file is `$HOME/.odbc.ini`

The default locations may be overwritten with environment variables `$ODBCSYSINI` and `$ODBCINI`. Please refer to the unixODBC documentation for additional information.

System ODBC.INI file is generally used for production deployments.

User ODBC.INI file is often used in multi-user development environments

A minimalistic user setup would include only one file, either `/etc/odbc.ini` or `$HOME/.odbc.ini`. In a typical use case the ODBCINST.INI file (e.g. `/etc/odbcinst.ini`) is not required.

Edit the *System* or *User DSN* file and add a DSN section using a DSN "demo" example included earlier in the chapter as a template. You should only need to adjust *Servername* and *Port* settings.

You can now test a DSN "demo" connection with unixODBC command-line tool `isql`:

```
$ isql -v demo
+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| quit                                  |
|                                       |
+---------------------------------------+
SQL>
```

Installing ODBC Driver on Windows

32-bit and 64-bit versions of psqlODBC drivers are available for download from PostgreSQL (`https://www.postgresql.org/ftp/odbc/versions/msi/`). Each driver is packaged as a standard Windows MSI file installer.

As a Windows Administrator, start the MSI installer and follow the on-screen instructions to complete the driver installation.

Configuring Cisco ParStream ODBC Connection on Windows

As of Windows NT, the System and User ODBC.INI files have been replaced with respective registry entries:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI]
[HKEY_CURRENT_USER\SOFTWARE\ODBC\ODBC.INI]
```

On Windows, ODBC DSN is configured with a Control Panel ODBC Applet.

Note:

32bit psqlODBC driver installed on a 64bit Windows platform operates within a WoW64 subsystem.

The System DSN registry entry for WoW64 subsystem is

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\ODBC\ODBC.INI]
```

To configure a DSN for a 32bit start psqlODBC driver installed on a 64bit Windows, start the WoW64 Control Panel ODBC Applet:

```
C:\Windows\SysWOW64\odbcad32.exe
```

psqlODBC driver installation includes ANSI and Unicode driver variants. ANSI driver is preferred for performance sensitive applications over the Unicode (UTF-16) version that uses 2 bytes to represent each character.

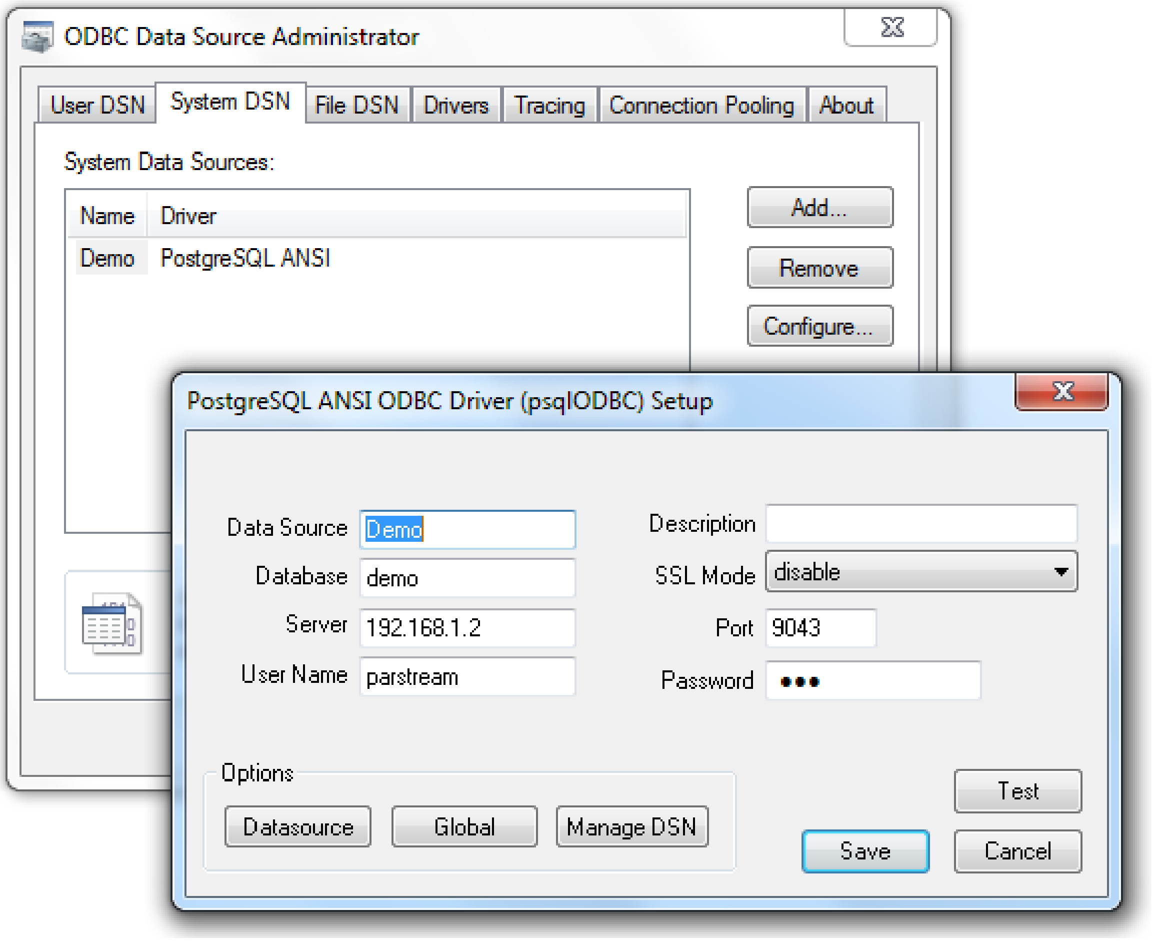
To configure a DSN, start the Control Panel ODBC Applet (for example, Start > Control Panel > Administrative Tools > Data Sources (ODBC)).

Click Add > PostgreSQL ANSI

Fill out the fields in the DSN Setup dialog as follows (see figure 17.1):

- "**Data Source**" and "**Database**" can be anything of your choice.
- "**Description**" can be empty.
- "**SSL Mode**" must be set to "**disable**."
- "**Server**" and "**Port**" have to be set to the values of the corresponding Cisco ParStream server.
- "**User Name**" and "**Pass phrase**" have to be set to the corresponding values if authentication is enabled (see section 9.2, page 78) or some arbitrary values otherwise (these field are not allowed to be empty).
- "**User Name**" and "**Pass phrase**" have to be set to some arbitrary values (the fields are not allowed to be empty)

Use the "Test" button to verify the database connectivity.



System Data Sources:

Name	Driver
Demo	PostgreSQL ANSI

Data Source	Demo
Database	demo
Server	192.168.1.2
User Name	parstream
Description	
SSL Mode	disable
Port	9043
Password	●●●

Figure 17.1: *Setting up DSN with ODBC Control Panel Applet*

JDBC Client Interface

The Cisco ParStream JDBC Driver uses PostgreSQL client protocol and extends the capabilities of the PostgreSQL JDBC Driver by adding additional data types, such as UINT, supported by Cisco ParStream. It is possible to use the stock PostgreSQL JDBC Driver if Cisco ParStream specific features are not used by the application.

The Cisco ParStream JDBC driver is a Type IV (pure) JDBC driver.

The Cisco ParStream JDBC driver requires Java 8 JRE or JDK.

The Cisco ParStream JDBC driver Class Name is "com.parstream.ParstreamDriver".

The Cisco ParStream server port for client JDBC connections is 1 higher than the basic port of the corresponding Cisco ParStream server. For example, if `port=9042`, the JDBC connection port is `9043`. See section 13.3.1, page 135 for details.

Cisco ParStream JDBC URL Specification

In JDBC, a database is represented by a URL. Cisco ParStream URL has the following format:

`jdbc:parstream://[host][:port]/[database][?connectionAttribute1=value1][&connectionAttribute2=value2]...`

where
host - host name or IP address of the server
port - server listening port for JDBC connections described above
database - database name, required, but currently a no-op for Cisco ParStream server
Additional connection attributes are: *user* - user name used for login (see section 9.2, page 78) *password* - password for login (see section 9.2, page 78) *ssl* - can be set to "true", to enable encrypted communication
Below are examples of valid Cisco ParStream URLs:

```
DriverManager.getConnection(
  "jdbc:parstream://localhost:9043/noop?user=johndoe&password=mysecret&ssl=true");
```

or:

```
Properties prop = new Properties();
prop.setProperty("user", "johndoe");
prop.setProperty("password", "mysecret");
prop.setProperty("ssl", "true");
DriverManager.getConnection("jdbc:parstream://localhost:9043/noop?&loglevel=0", prop);
```

or without authentication:

```
DriverManager.getConnection("jdbc:parstream://localhost:9043");
```

Installing JDBC Driver

The Cisco ParStream JDBC driver includes the following:

File	Description
`cisco-parstream-jdbc-<VERSION>-javadoc.jar`	javadoc generated documentation
`cisco-parstream-jdbc-<VERSION>.jar`	JAR file for Java 8

A pure Type IV driver does not need an "installation". A Cisco ParStream JDBC driver JAR file can be placed into any directory accessible by client Java applications.

On Linux platforms Cisco ParStream recommends copying the driver JAR file to `$PARSTREAM_HOME/jdbc`.

On Windows platforms the driver JAR may be stored, for example, in `C:\ParStream\jdbc`.

Configuring JDBC Connections

To create a JDBC connection, you need:

1. Set the Java CLASSPATH to include the full path to the Cisco ParStream JDBC driver JAR file. Java CLASSPATH can be set using multiple alternative methods.
   For example, in Linux Bash:

   ```
   $ export CLASSPATH=.:<full path to
       JAR>/cisco-parstream-jdbc-<VERSION>.jar:$CLASSPATH
   ```

   or on Windows, if setting in the Command Prompt:

   ```
   C:\>set CLASSPATH=.;%CLASSPATH%
   C:\>set
       CLASSPATH=C:\ParStream\jdbc\cisco-parstream-jdbc-<VERSION>.jar;%CLASSPATH%
   ```

2. In your Java application, load the Cisco ParStream driver:

   ```
   Class.forName("com.parstream.ParstreamDriver");
   ```

3. Then make the connection:

   ```
   String URL = "jdbc:parstream://localhost:9043";
   String dbUser = "parstream";
   String dbPwd  = "n-a";

   Connection conn = DriverManager.getConnection(URL, dbUser, dbPwd);
   ```

Java Streaming Import Interface (JSII)

This chapter covers the *Java Streaming Import Interface* provided by Cisco ParStream. It provides the ability to import data into the database from a Java application.

Introduction

The Java Streaming Import Interface is built on top of the Streaming Import Interface. It provides methods that internally invoke the streaming functions of the C-API. The Java methods allows client connections to import data into a running server. It also provides methods that:

- allow obtaining meta-information about tables and columns, start an insert operation, write data, commit an import (or rollback),
- allow converting Java abstract data types to Cisco ParStream column data types. Such methods ensure range validity during conversion, and are optional to use.

The following sections describe the general concept of the Cisco ParStream Java Streaming Import Interface in more detail.

General Concept

Connecting to Cisco ParStream

A client application can use the Java Wrapper to establish a connection with a Cisco ParStream database.

Thread Safety

Multiple Java connections can run in parallel provided each connection uses its own instance of a `ParstreamConnection`. These parallel connections can even insert data on the same table.

Inserting data

Before inserting data, the client application must specify the target table to which, data from the current connection goes to. Such table must exist in the database prior to inserting into it.

Data to be inserted from Java must be stored in an `Object[]`. The length of the `Object[]` must have a length identical to the number of columns in the target table.

While the API provides the ability to insert row by row, internally multiple rows are transferred in bigger blocks of data to the server for better performance.

The inserted data is subject to ETL statements specified in the table definition like CSV-based import data (see section 10.6, page 104). For this reason, columns marked with `CSV_COLUMN ETL` don't count as valid column for streaming imports.

In addition, skipped columns (columns marked with `SKIP TRUE`, see section 24.2.4, page 285) also don't count as valid column for streaming imports.

Commit and Rollback

Data that should be inserted needs to be confirmed by a *commit* command. All data that is inserted between *commit* commands is written to partitions. A success of the *commit* command indicates the successful creation of all resulting partitions. In case of an exception no partitions are created.

A client can also abort the creation and activation of partitions by a *rollback* command. In this case all data since the last commit is ignored by the server.

Exception Handling

Methods in this driver may throw two types of exceptions when invoked.

- A `com.parstream.driver.ParstreamFatalException` means that you no longer can use this connection.

    1. all started but uncommitted insertions are ignored (as if a rollback would have been called).
    2. The `close()` method is the only method that can be invoked following a fatal exception, which invalidates the connection handle and free all associated memory.

- A `com.parstream.driver.ParstreamException` means that a particular call failed but the connection is still usable.

    1. Example: if you attempt to insert an `Object[]` that has a length not equal to the number of columns in the target table.

Mapping Data Types between Cisco ParStream and Java

For each item in the `Object[]` to be inserted, the data type must be compatible with the corresponding column type in the Cisco ParStream table (see chapter 23, page 267). Table 19.1 shows which Java data types are compatible with which Cisco ParStream data types.

To insert a `NULL`, set the corresponding element in the `Object[]` to `null`.

Optionally, the Java Streaming Import provides a package, that contains class helpers to convert Java data types into Cisco ParStream's Java column types. These classes can be found in the package: `com.parstream.driver.datatypes`. For example, the `PsUint8` class, provides methods to convert from Java `Byte, Short, Integer, Long` into Java type `Short`, which subsequently can be inserted into a Cisco ParStream `UINT8` column.

Note that Cisco ParStream internally uses special values to represent a SQL `NULL`. For this reason you can't for example insert the value 255 as ordinary UINT8 (see section 23.2, page 267). Inserting these values instead of `NULL` results in an exception. See section 19.5.21, page 228 for the defined constants.

Cisco ParStream Data Type	Java Data Type
`UINT8`	`Short`
`UINT16`	`Integer`
`UINT32`	`Long`
`UINT64`	`Long`
`INT8`	`Byte`
`INT16`	`Short`
`INT32`	`Integer`
`INT64`	`Long`
`DOUBLE`	`Double`
`FLOAT`	`Float`
`VARSTRING`	`String`
`SHORTDATE`	`ParstreamShortDate`
`DATE`	`ParstreamDate`
`TIME`	`ParstreamTime`
`TIMESTAMP`	`ParstreamTimestamp`
`BITVECTOR8`	`Short`
`BLOB`	`String`

Table 19.1: *Mapping between Cisco ParStream and Java data types*

Java Driver Limitations

Please note the following limitations that currently apply:

- UINT64 datatype supports a range larger than Java's long data type (signed 64-bit).

Using the Java Streaming Import Interface

The following example provides a rough overview how to use the Java driver for the C-API.

The Cisco ParStream distribution provides a complete example with a corresponding database. It can be found in the directory `examples/importapi_java` (see section B.1, page 394 for details).

Dealing with Connections

First, you have to instantiate and create a handle for each connection:

```
ParstreamConnection psapi = new ParstreamConnection();
psapi.createHandle();
```

`createHandle()` attempts to create a handle, which will be used by the driver internally for all subsequent method calls dealing with the connection, until it is closed. This method throws a `ParstreamFatalException` in case the handle creation fails, which happens of the application is out of memory.

Optionally, you can set connection options before attempting to connect to the database. For example:

```
psapi.setTimeout(20000);
```

You may also set the import priority. The priority may either be: high, medium, or low. For example:

```
psapi.setImportPriority(ImportPriority.MEDIUM);
```

The possible ImportPriority ENUM values can be found in `com.parstream.driver.ParstreamConnection.ImportPriority`.

Following that, you can establish a connection with the server using `connect()` method:

```
psapi.connect(host, port, username, password, useSSL, caFilePath);
```

The `username` is the login name of the registered database user, `password` its PAM pass phrase (see section 9.2, page 78). `useSSL` configures whether to enable SSL encryption for the client. `caFilePath` configures a file path to a CA file for the certificate verification process.

The `close()` method is used to close the connection:

```
psapi.close();
```

Following this, the initially created handle is no longer valid. Invoking `connect()` is not possible.

Retrieving Version Information

If you want to know, which Cisco ParStream version the used API has:

```
String version = psapi.getVersion();
System.out.println("Using psapi version: " + version);
```

If you want to know the database version for a specific connection:

```
String dbVersion = psapi.getDbVersion();
System.out.println("Cisco ParStream db version: " + dbVersion);
```

If you want to know the database metadata version for a specific connection:

```
String metadataVersion = psapi.getMetadataVersion();
System.out.println("Cisco ParStream metadata version: " + metadataVersion);
```

Querying Metadata (Tables and Columns)

You can list the tables in the database:

```
String[] tableNames = psapi.listTables();
for (int i = 0; i < tableNames.length; i++) {
        System.out.println("table " + i + ": " + tableNames[i]);
}
```

You may also list the columns of a specific table:

```
ColumnInfo[] columns = psapi.listImportColumns("MyTable");
for (int i = 0; i < columns.length; i++) {
        System.out.println("column " + i + ": " + columns[i].getName());
}
```

Notes for both methods:

- Multiple calls of this method are possible but will always return the same data. Thus, to retrieve the effect of a schema change on the table, you have to use a new connection.
- A `ParstreamException` is thrown if listImportColumns is provided with a non-existent table name.
- A `ParstreamFatalException` is thrown if the handle is not usable, or if there are no tables in the database

Inserting Rows

To insert rows, first you have to start an insertion. You have to specify the column names to insert. If you leave columns out, the specified default values will be filled in for that columns. The order of the given columns has to be respected by the data insertion with the `rawInsert()` method.

```
String names[] = { "age", "weight", "id" };
psapi.prepareInsert("MyTable", names);
```

This method throws a `ParstreamFatalException` if the handle is invalid. Throws a `ParstreamException` if the specified table name does not exist.

After the connection is prepared for insertion, you have to insert data row by row. An Object[] containing the data is passed to the `rawInsert()` method. Types of array elements must match corresponding types of the table (see section 19.2.5, page 217).

```
short age = 35;
double weight = 67.5;
long id = 12345678l;

Object[] data = new Object[]{age, weight, id};
psapi.rawInsert(data);
```

This method throws a `ParstreamFatalException` if the handle is invalid and connection must be closed. It throws `ParstreamException` if the row insertion failed. This may be because:

- Object[] length not equal to number of columns
- Invalid values especially apply to *values* that internally represent `NULL` values (see section 23.2, page 267).
- The order given by the `prepareInsert` method was not respected by the `rawInsert` method

To insert a `NULL` value for a specific column, assign the corresponding array element to `null`. For example:

```
short age = 35;
double weight = 67.5;

Object[] data = new Object[]{age, weight, null};
psapi.rawInsert(data);
```

Finally, you have to commit the insertion:

```
psapi.commit();
```

Alternatively, you can rollback the insertion:

```
psapi.rollback();
```

If a problem occurs with a commit or a rollback, a `ParstreamFatalException` is thrown.

Dealing with Multivalues

To insert multivalues, an array needs containing the multivalues need to be constructed. The data type of the array depends on the data type of the multivalue column in Cisco ParStream. For example, if Cisco ParStream has a multivalue column of data type `INT16`, then a `Short[]` array need to be constructed to hold the multivalues (see section 19.1, page 218 for the table of all data type mappings).

For example:

```
Short[] multivalues = new Short[] {1, 2, null, 3};
Object[] rawData = new Object[] {...., multivalues,... };
psapi.rawInsert(rawData);
```

Dealing with data type mapping

Conversion from a Java type to a Cisco ParStream type can be done manually by the user. Alternatively the Java Streaming Import package provides helper classes to perform such conversions and ensure that the value to be converted is within range. This is optional. For example, when inserting a value into a Cisco ParStream UINT8 column:

```
String originalValue = "12";
short psValue = PsUint8.valueOf(originalValue);
Object[] rawData = new Object[] {...., psValue,...};
psapi.rawInsert(rawData);
```

In addition the Java Streaming Import package provides an insert method, which tries to convert each column to the expected Java data type matching the Cisco ParStream data type. This is optional. Table 19.2 shows which Java data types are compatible with which Cisco ParStream data types. Multivalues will not be converted to the Cisco ParStream data type.

For example, when inserting an unknown input type.

Cisco ParStream Data Type	Java Data Type(s)
UINT8	Byte, Short, Integer, Long
UINT16	Byte, Short, Integer, Long
UINT32	Byte, Short, Integer, Long
UINT64	Byte, Short, Integer, Long
INT8	Byte, Short, Integer, Long
INT16	Byte, Short, Integer, Long
INT32	Byte, Short, Integer, Long
INT64	Byte, Short, Integer, Long
DOUBLE	Float, Double
FLOAT	Float, Double
VARSTRING	String
SHORTDATE	ParstreamShortDate, java.util.Date, GregorianCalendar
DATE	ParstreamDate, java.util.Date, GregorianCalendar
TIME	ParstreamTime, java.util.Date, GregorianCalendar
TIMESTAMP	ParstreamTimestamp, java.util.Date, GregorianCalendar
BITVECTOR8	Short
BLOB	String

Table 19.2: *Mapping between Cisco ParStream and Java data types*

```
Object[] data = new Object[]{35, 67.5f, null};
psapi.insert(data);
```

Dealing with TIMESTAMP Columns

As described in section 23.4, page 268 the timestamp is stored uninterpreted without a time zone. The example timestamp "2016-06-14T12:30:45.567+0200" will be stored as "2016-06-14 12:30:45.567". The zone offset information for a specific day will be lost.

Java date time classes derived from `java.util.Date` internally store their value as milliseconds since the Epoch and are always handled as if they are timestamp values of the default JVM time zone. For example 1465911045128 milliseconds since the Epoch will be represented:

- in JVM time zone `UTC` as "2016-06-14 13:30:45.128"
- in JVM time zone `Europe/Berlin` as "2016-06-14 15:30:45.128"

The class `com.parstream.driver.ParstreamTimestamp` implements following constructors:

- `GregorianCalendar`
- `java.util.Date`
- `java.sql.Date`
- `long epoc`: number of milliseconds since the Epoch as UTC timestamp
- `long seconds, int milliseconds`: number of seconds since the Epoch as UTC timestamp and the milliseconds part (0-999)

With the constructor `java.util.Date` we implicitly accept `java.sql.Timestamp` for constructing `ParstreamTimestamp` instances. Invoking any of the above constructors with a `NULL` argument will insert a `NULL` into the database.

Dealing with TIME Columns

As described in section 23.4, page 268 the time is stored uninterpreted without a time zone.

The class `com.parstream.driver.ParstreamTime` implements following constructors:

- `GregorianCalendar`
- `java.util.Date`
- `java.sql.Date`
- `int hour, int minute, int second, int millisecond`

With the constructor `java.util.Date` we implicitly accept `java.sql.Time` for constructing `ParstreamTime` instances. Invoking any of the above constructors with a `NULL` argument will insert a `NULL` into the database.

The class `GregorianCalendar` and all derivations of the class `java.util.Date` contain a time zone. This time zone and its offset will be ignored while inserting the time value into Cisco ParStream.

In case of a `java.util.Date` the time part of the local representation of the timestamp in sense of the JVM time zone will be inserted. For example 1465911045128 milliseconds since the Epoch ("2016-06-14 13:30:45.128 UTC") will be inserted in JVM time zone `Europe/Berlin` as "15:30:45.128".

Dealing with DATE Columns

The DATE column type cannot respect any time zone due to its SQL Standard definition.

The class `com.parstream.driver.ParstreamDate` implements following constructors:

- `GregorianCalendar`
- `java.util.Date`
- `java.sql.Date`
- `int year, int month, int day`

With the constructor `java.util.Date` we implicitly accept `java.sql.Date` for constructing `ParstreamDate` instances. Invoking any of the above constructors with a `NULL` argument will insert a `NULL` into the database.

The class `GregorianCalendar` and all derivations of the class `java.util.Date` contain a time zone. This time zone and its offset will be ignored while inserting the date value into Cisco ParStream.

In case of a `java.util.Date` the date part of the local representation of the timestamp in sense of the JVM time zone will be inserted. For example 1465947000000 milliseconds since the Epoch ("2016-06-14 23:30:00 UTC") will be inserted in JVM time zone `Europe/Berlin` as "2016-06-15".

Java driver for Streaming Import Interface Reference

Running the example

Detailed instructions for compiling and running the example can be found in `examples/importapi_java/README.txt`.

ColumnInfo

This class provides information about a specific column in the database. A `ColumnInfo[]` can be obtained by invoking the method `listImportColumns("MyTable")`. There are four methods that provide information of a specific column.

- `public String getName()`
- `public Type getType()`
- `public Singularity getSingularity()`
- `public int getLength()`

ColumnInfo.Type

`Type` is an enum defining the possible data types of various columns in a Cisco ParStream table.

```
public enum Type {
        UINT8, UINT16, UINT32, UINT64,
        INT8, INT16, INT32, INT64,
        FLOAT, DOUBLE, VARSTRING,
        SHORTDATE, DATE, TIME, TIMESTAMP,
        BITVECTOR8, BLOB;
};
```

ColumnInfo.Singularity

`Singularity` is an enum defining the possible column singularity in a Cisco ParStream table.

```
public enum Singularity {
  SINGLE_VALUE, MULTI_VALUE;
};
```

PsUint8

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type UINT8. It provides two static methods:

- PsUint8.valueOf(long value)
- PsUint8.valueOf(String value)

PsUint16

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type UINT16. It provides two static methods:

- PsUint16.valueOf(long value)
- PsUint16.valueOf(String value)

PsUint32

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type UINT32. It provides two static methods:

- PsUint32.valueOf(long value)
- PsUint32.valueOf(String value)

PsUint64

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type UINT64. It provides two static methods:

- PsUint64.valueOf(long value)
- PsUint64.valueOf(String value)

PsInt8

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type INT8. It provides two static methods:

- PsInt8.valueOf(long value)
- PsInt8.valueOf(String value)

PsInt16

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type INT16. It provides two static methods:

- PsInt16.valueOf(long value)
- PsInt16.valueOf(String value)

PsInt32

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type INT32. It provides two static methods:

- PsInt32.valueOf(long value)
- PsInt32.valueOf(String value)

PsInt64

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type INT64. It provides two static methods:

- PsInt64.valueOf(long value)
- PsInt64.valueOf(String value)

PsDouble

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type DOUBLE. It provides two static methods:

- PsDouble.valueOf(long value)
- PsDouble.valueOf(String value)

PsFloat

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type FLOAT. It provides two static methods:

- PsFloat.valueOf(long value)
- PsFloat.valueOf(String value)

PsBitVector8

This class provides a set of static methods that act as helpers when converting Java types into the Cisco ParStream column type BITVECTOR8. It provides two static methods:

- PsBitVector8.valueOf(long value)
- PsBitVector8.valueOf(String value)

ParstreamShortDate

This class is a Java representation of the Cisco ParStream `SHORTDATE` datatype. It has four possible constructors:

- ParstreamShortDate(GregorianCalendar calendar)
- ParstreamShortDate(int year, int month, int day)
- ParstreamShortDate(java.sql.Date date)
- ParstreamShortDate(java.util.Date date)

Value range is from 01.01.2000 to 31.12.2178. Throws `ParstreamException` if input value is out of range.

ParstreamDate

This class is a Java representation of the Cisco ParStream `DATE` datatype. It has four possible constructors:

- ParstreamDate(GregorianCalendar calendar)
- ParstreamDate(int year, int month, int day)
- ParstreamDate(java.sql.Date date)
- ParstreamDate(java.util.Date date)

Value range is from 01.01.0000 to 31.12.9999. Throws `ParstreamException` if input value is out of range.

ParstreamTime

This class is a Java representation of the Cisco ParStream `TIME` datatype. It has four possible constructors:

- ParstreamTime(GregorianCalendar calendar)
- ParstreamTime(int hour, int minute, int second, int millisecond)
- ParstreamTime(java.sql.Date date)
- ParstreamTime(java.util.Date date)

Value range is from 00:00:00.000 to 23:59:59.999. Throws `ParstreamException` if input value is out of range.

ParstreamTimestamp

This class is a Java representation of the Cisco ParStream `TIMESTAMP` datatype. It has six possible constructors:

- ParstreamTimestamp(GregorianCalendar calendar)
- ParstreamTimestamp(int year, int month, int day, int hour, int minute, int second, int millisecond)
- ParstreamTimestamp(java.sql.Date date)
- ParstreamTimestamp(java.util.Date date)
- ParstreamTimestamp(long milliseconds): number of milliseconds since the Epoch as UTC timestamp
- ParstreamTimestamp(long seconds, int milliseconds): number of seconds since the Epoch as UTC timestamp and the milliseconds part (0-999)

Value range is from 01.01.0000 00:00:00.000 to 31.12.9999 23:59:59.999. Throws `ParstreamException` if input value is out of range.

Connection Options

Here you can find a list of options that you may set for a Cisco ParStream connection.

- `setTimeout(int t)`: sets the connection time out in milliseconds (see section 19.5.22, page 228)
- `setImportPriority(ParstreamConnection.ImportPriority priority)`: sets the priority of the import, providing an ENUM value (see section 19.5.22, page 229)

NULL Constants

Cisco ParStream uses particular values internally to store `NULL` values. An inserted row that contains one of these values will be rejected, and a `com.parstream.driver.ParstreamException` will be thrown.

To be able to check whether integral values have the special meaning to represent `NULL` (see section 23.2, page 267) the following constants are defined:

```
#define PARSTREAM_INT8_NULL        127
#define PARSTREAM_UINT8_NULL       255
#define PARSTREAM_INT16_NULL       32767
#define PARSTREAM_UINT16_NULL      65535
#define PARSTREAM_INT32_NULL       2147483647
#define PARSTREAM_UINT32_NULL      4294967295
#define PARSTREAM_INT64_NULL       9223372036854775807
#define PARSTREAM_UINT64_NULL      18446744073709551615u
#define PARSTREAM_SHORTDATE_NULL   65535
#define PARSTREAM_DATE_NULL        4294967295
#define PARSTREAM_TIMESTAMP_NULL   18446744073709551615u
#define PARSTREAM_TIME_NULL        4294967295
#define PARSTREAM_BITVECTOR8_NULL  0
```

Note that for floating-point values NaN is used as `NULL` value, which means that this value should not be used in the API.

Methods

In general, the majority of the C-API functions have a corresponding Java method. The following will give a brief description of each Java method.

`void` **createHandle** `()`

- Creates a handle that will be used for all other methods of the streaming api. This must be the first method to call before any subsequent calls are made.
- throws
  - `ParstreamFatalException` if out of handles

`void` **setTimeout** `(int` *`timeout`*`)`

- Sets the connection timeout interval using the given parameter in milliseconds. The default timeout value is 0, which means there is no timeout on communication actions to the server.

- throws
  - `ParstreamFatalException` if handle is invalid

`void` **setImportPriority** `(ParstreamConnection.ImportPriority` *`priority`*`)`

- Sets the priority of the import given an ENUM that can either be (HIGH, MEDIUM, or LOW)
- throws
  - `ParstreamFatalException` if handle is invalid

`void` **connect** `(String` *`server`*`, String` *`port`*`,`
`String` *`username`*`, String` *`password`*`)`

- Initializes a connection with a Cisco ParStream server via a TCP socket. This call must be made before attempting to interact with the server.
- The `username` is the login name of the registered database user, `password` its PAM pass phrase (see section 9.2, page 78).
- throws
  - `ParstreamFatalException` on invalid handle or connection error

`void` **close** `()`

- Closes the connection and clean up connection handle. The handle can no longer be used to establish a new connection. A new handle must be created to reconnect to a Cisco ParStream server

`String` **getVersion** `()`

- Retrieves the Cisco ParStream version of the psapi interface

`String` **getDbVersion** `()`

- Retrieves the Cisco ParStream version of the connected Cisco ParStream database server
- throws
  - `ParstreamFatalException` on invalid handle or connection error

`String` **getMetadataVersion** `()`

- Retrieves the Cisco ParStream metadata version of the connected Cisco ParStream database server
- throws
  - `ParstreamFatalException` on invalid handle or connection error
  - `ParstreamException` if the server version is too old to provide a metadata version

`String` **getConnectionId** `()`

- Retrieves the unique id for this connection. The unique id can be used to check the Cisco ParStream system table ps_info_insert_history for the status of the imports for this connection.

- throws
  - `ParstreamFatalException` on invalid handle or connection error

`String[]` **listTables** `()`

- Retrieves a string array of table names available at the Cisco ParStream server
- throws
  - `ParstreamFatalException` on invalid handle or connection error

`ColumnInfo[]` **listImportColumns** `(String` *`tableName`*`)`

- Retrieve list of column descriptions of specific table
- throws
  - `ParstreamFatalException` on invalid handle or connection error
  - `ParstreamException` if table name is invalid

`void` **prepareInsert** `(String` *`tableName`*`)`

- Prepare connection for inserting rows into a single table using the default column order returned by `listImportColumns`
- throws
  - `ParstreamFatalException` on invalid handle or connection error
  - `ParstreamException` if table name is invalid

`void` **prepareInsert** `(String` *`tableName`*`, String[]` *`columnNames`*`)`

- Prepare connection for inserting rows into a single table using the given column order given by *columnNames*. Left out columns will be filled with default values.
- throws
  - `ParstreamFatalException` on invalid handle or connection error
  - `ParstreamException` if table name is invalid

`void` **rawInsert** `(Object[]` *`insertValues`*`)`

- Insert a single row into a prepared table.
- throws
  - `ParstreamFatalException` on invalid handle or connection error
  - `ParstreamException` if the row is rejected due to erroneous data values

`void` **insert** `(Object[]` *`insertValues`*`)`

- Insert a single row into a prepared table. If possible columns will be cast without data-loss to expected Cisco ParStream data type.
- throws
  - `ParstreamFatalException` on invalid handle or connection error

  - `OutOfRangeException` if a column would not fit into its data type value range
  - `AutoCastException` if a column cannot be cast into the expected Cisco ParStream data type
  - `ParstreamException` if the row is rejected due to erroneous data values

`void` **commit** `()`

- Commit inserted rows to the Cisco ParStream server. After the commit returned successfully, the newly inserted data can be retrieved from the server.
- throws
  - `ParstreamFatalException` on invalid handle or connection error

`void` **rollback** `()`

- Rollback all insert operations since the the connection was prepared for insertion. After a rollback you can start inserting rows again after calling psapi_prepare_insert.
- throws
  - `ParstreamFatalException` on invalid handle or connection error

External User-Defined Table Operators (xUDTO)

This sections describes the ability of Cisco ParStream to define **external user-defined table operators (xUDTO)**:

- **User-defined table operators**, because Cisco ParStream allows to process one or multiple rows of input producing none, one, or multiple output rows.
- **external**, because the functionality is defined as separate process, which is called from the core Cisco ParStream processes if necessary.

This feature especially can be used to use functionality defined by the programming language R (see section 20.4, page 238).

Note that registered xUDTO functions are listed by the `ps_info_udf` system table (see section 26.4, page 324).

Concept of Using User-Defined Table Operators (UDTO)

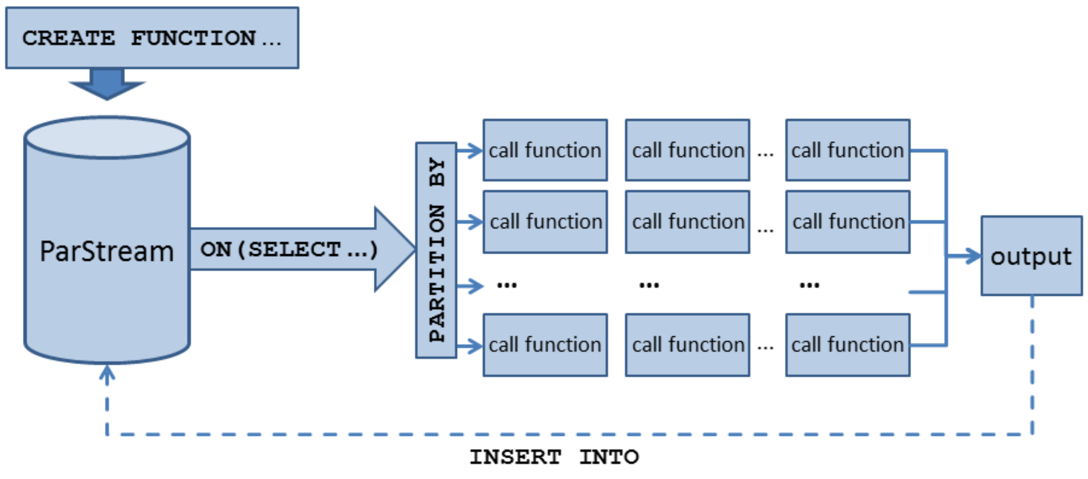


Figure 20.1: *Using external User-Defined Table Operators*

In principle, xUDTOs work as described in figure 20.1:

- With `CREATE FUNCTION` you can define any function name as representing a functionality processed by external processes. Such a function uses the data imported to compute a local output.
- You can use an `ON` clause (with an inner `SELECT` clause) to specify which data has to be processed by the user defined table operator.
- The mandatory `PARTITION BY` clause divides the data for the external processes into multiple partitions so that the data can be processed by the user-defined function in parallel.

- The data is processed by a limited number of external processes. If more partitions than this count need external processing, additional external processes are started sequentially as "slots" become available again until all partitions have been processed.
- The result of applying the xUDTO is a table expression and can be persisted via `INSERT INTO`.

Note that this is just a conceptional overview. Cisco Parstream especially will optimize the whole processing if this is possible. For this reason, partitioning the data for external processes should match the partitioning inside the Cisco ParStream database.

Enabling External Processing of xUDTOs

In general, to enable external processing of external user-defined table operators you have to

- define a *script type*, which you can use and refer to to define specific functions
- define specific functions

Defining Script Types

A *script type* defines the general handling of a specific type of scripts, such as the command to call to start a process dealing with a specific function call. It has to be defined in an `external` section of the INI file(s) of Cisco ParStream (see section 8.1.1, page 73).

For example, to enable calling shell scripts using the command `/bin/sh` you can define the following:

```
[external.sh]
command = /bin/sh %script %args
```

As you can see, you can use the following placeholders here:

- **`%script`** is defined when defining the concrete user-defined table function with `CREATE FUNCTION` It allows to pass arguments defined by the place, where the concrete name of the SQL function is defined. A typical example is the name of the script called, when the command calls a scripting command. See section 20.2.2, page 234 for details.
- **`%args`** can be used to pass arguments even from a specific function call as part of the query using the user-defined function call by a `COMMAND_LINE_ARGS` clause. See section 20.3.3, page 237 for details.

As another example, you can specify a script type `sed`, which calls the standard `sed` utility, using the passed argument as sed command (see its usage below):

```
[external.sed]
command = /bin/sed -e %args
```

Defining External Functions

To define external functions, you need a defined script type (see above). Then you can define a function with a `CREATE FUNCTION` command.

The syntax of the `CREATE FUNCTION` command is as follows:

```
CREATE FUNCTION <name>( <scriptType>, <script>,
                        <inFormat>, <outFormat> )
```

The meaning of the arguments is as follows:

- *scriptType* is the type, for which a corresponding command has to be registered in the INI file (as described above).
- *script* is the script argument passed to the define command as `%script`.
- *inFormat* is the input format that Cisco ParStream uses to provide its data as input for the called command/script. Currently, only the format `ParStream` is supported, which is the format Cisco ParStream uses for output of regular queries (CSV format with semicolon as delimiter and strings in parentheses).
- *outFormat* is the output format that Cisco ParStream requires to read and process output of the called command/script. Currently, only the format `ParStream` is supported, which is the format Cisco ParStream uses for output of regular queries (CSV format with semicolon as delimiter and strings in parentheses).

For example:

```
CREATE FUNCTION myFunc('sh', 'MyScript.sh', 'ParStream', 'ParStream')
```

registers `myFunc` as external user-defined table operator using the `sh` script type calling the script `MyScript.sh` with the standard input and output formats.

Note that the `%script` argument expected to be located in the directory for user-defined function, using the option `udfLibraryPath` (see section 13.2.1, page 122).

If the script type is registered as follows:

```
udfLibraryPath = udf

[external.sh]
command = /bin/sh %script
```

And a server starts in the directory `/opt/cisco/kinetic/parstream-database/bin`, then corresponding function calls defined for this script type as above will result in the following call to execute the external user-defined table operator at runtime:

```
/bin/sh /opt/cisco/kinetic/parstream-database/bin/udf/MyScript.sh
```

The defined command line must be able to execute on all nodes, i.e. all the needed executables, and used scripts and data files need to be present on each node.

Note that the `CREATE FUNCTION` command has to be sent to just **one node** of a cluster.

The registration of an external user-defined table operator creates a new metadata version (see section 5.2, page 34).

Using External User-Defined Table Operators

To use xUDTOs, you have to call the defined function passing the data to process with an `ON` clause. The minimum syntax of such a call using an `ON` clause is roughly as follows:

```
func (ON (SELECT ...)
      PARTITION BY ...
      RETURNS (...)
     )
```

The `SELECT` defines the data to be used as input for the external user-defined table operators.

With the `PARTITION BY` you define how to partition the data as input for the function. For all rows using the same partition value(s) the function will be called once. Ideally, part of the partitioning here should be the general partitioning and distributing used in the database.

The `RETURNS` clause defines the names of the returned columns. The output of the defined functions must have a corresponding number of columns. According to the output format ’`ParStream`’ these columns have to be separated by a semicolon.

For example, for a script type `sh` defined as

```
[external.sh]
command = /bin/sh %script
```

an external user-defined table operator, defined as

```
CREATE FUNCTION myFunc('sh', 'MyScript.sh', 'ParStream', 'ParStream')
```

can be used as follows in an `ON` clause:

```
myFunc (ON (SELECT myStr, myInt FROM MyTable)
        PARTITION BY myInt
        RETURNS (outStr VARSTRING, outVal UINT64)
       )
```

meaning that:

- The data from the query `SELECT myStr, myInt FROM MyTable` is used as input for `myFunc()` passing it in the usual output format to the registered script `MyScript.sh`
- With `PARTITION BY` you specify that the data sent to the external process is partitioned by the values of the passed columns. Passing column `myInt` in the sample above thus leads to `myFunc` being called once for each distinct value in column `myInt` (and receiving all matching rows).
- With `RETURNS` you can specify the name and type of the values returned in each row of the script called. Here, the output of the script must be rows containing a string and an unsigned integer (as usual separated by a semicolon).

Two additional clause are possible in an ON clause:

```
myFunc (ON (SELECT myStr, myInt FROM MyTable)
        PARTITION BY myInt
        ORDER BY myInt           -- optional
        COMMAND_LINE_ARGS('')    -- optional
        RETURNS (outStr VARSTRING, outVal UINT64)
       )
```

- With ORDER BY you can specify that the rows passed to the function are sorted according to the passed columns.
- With COMMAND_LINE_ARGS you can pass strings of command line arguments which are used as %args in the specified script type command (see section 20.2.1, page 233).

In the execution up to maxExternalProcesses will be launched to process the different partitions simultaneously.

Example Using a xUDTO

You can use a xUDTO in different ways. For example as part of a general SELECT statement you can output the resulting data:

```
SELECT * FROM myFunc (ON (SELECT myStr, myInt FROM MyTable)
                      PARTITION BY myInt
                      RETURNS (outStr VARSTRING, outVal UINT64)
                     )
         ORDER BY outStr
```

You can also use this function call to insert the resulting output into a new table using the INSERT INTO statement (see section 10.7, page 107). For example, after defining a table for the output data as follows:

```
CREATE TABLE OutTable (
  outStr VARSTRING,
  outVal UINT64
)
DISTRIBUTE EVERYWHERE
```

or as follows:

```
CREATE TABLE OutTable (
  outStr VARSTRING,
  outVal UINT64 INDEX EQUAL
)
PARTITION BY outVal
DISTRIBUTE OVER outVal
```

the following statement inserts the output/result of myFunc() into table OutTable:

```
INSERT INTO OutTable
  SELECT * FROM myFunc (ON (SELECT myStr, myInt FROM MyTable)
                        PARTITION BY myInt
                        RETURNS (outStr VARSTRING, outVal UINT64)
                       )
```

If for example, the script simply passes the data read through:

```
#!/bin/sh
while read ROW
do
  echo $ROW
done
```

and the first row contains `hello world` as value of column `myStr` and `42` as value of column `myInt`, the function call will pass the corresponding line to standard input as `"hello world";42`. This means that, because this row is also used as output, the corresponding output row will have a column `outStr` with value “hello world” and a column `outVal` with value `42`.

Note that output rows always have to end with a newline character to get processed.

Dealing with Errors in xUDTOs

To be able to detect and deal with errors, any error written to the standard error output by the called scripts will be redirected into the log file of the corresponding server.

For example, the following script will print to standard error whenever it is started, so that the server log files will contain corresponding entries, for each time when the script is called due to a partitioned external user-defined table operator call:

```
#!/bin/sh
echo "MyScript.sh started" 1>&2
while read ROW
do
  echo $ROW
done
```

Using Command Line Args

By using the `COMMAND_LINE_ARGS` clause, you can specify details of the called functionality within the `ON` clause.

If you have e.g. specified the command to be:

```
[external.sed]
command = /bin/sed -e %args
```

and defined a corresponding func `sed()`

```
CREATE FUNCTION sed('sed', '', 'ParStream', 'ParStream')
```

(Note that here the second argument for the `%script` is empty, because it is not used.)

Then, you can pass the specific sed command inside your `ON` clause. Here, for example to replace in each row all occurrences of a digit by the digit `0`:

```
SELECT * FROM sed (ON (SELECT myStr, myInt FROM MyTable)
                   PARTITION BY myInt
                   COMMAND_LINE_ARGS('s/[0-9]/0/g')
                   RETURNS (outStr VARSTRING, outVal UINT64)
                  )
         ORDER BY outStr, outVal
```

of `end` by `begin`, which would mean that an input such as

```
"this 1 world";42
```

will be converted into

```
"this 0 world";00
```

so that the resulting row gets `this 0 world` as value for `outStr` and `0` as value for `outVal`.

Limitations using xUDTOs

Please note the following limitations using user-defined table operators:

- Output lines written by the called commands/scripts must end with a newline character. Otherwise they are not processed.

Integrating R Scripts as xUDTOs

One way to use external user-defined table operators (xUDTOs, see section 20, page 232) is to enable calling R scripts according to the R Project for Statistical Computing (see `http://www.r-project.org/`). Here, we describe how to do that according to the example `xUDTO` provided in the example directory of Cisco ParStream (see section A.5, page 393).

Defining an R Script Type

To enable calling R scripts, you first have to define a corresponding script type. For example:

```
[external.R]
command = Rscript %args %script
```

Here, `%args` is used to be able to pass command-line arguments to the call of the R script associated with a specific xUDTO.

Register R Scripts

Then, you can register a function, calling a specific R script passed as second argument. For example, we could define a function that performs a linear regression and returns the corresponding linear model:

```
CREATE FUNCTION lm_R('R', 'lm.R', 'ParStream', 'ParStream')
```

This command registers an external user-defined table operator `lm_R()` calling the R script `lm.R` (by calling the corresponding command for the script type `R` using `lm.R` as parameter `%script`). The last two parameters require to use the standard Cisco ParStream input and output format (other formats are not supported yet).

The R script has to read input and to write output according to the standard input file format of Cisco ParStream. For example:

```
#!/usr/bin/env Rscript

# we fetch data from stdin
f <- file('stdin')
open(f)
t <- read.table(f,
                # input schema
                colClasses = c('character',
                               'integer',
                               'integer',
                               'integer',
                               'numeric'),
                col.names = c('sensor_id',
                              'week_of_year',
                              'day_of_week',
                              'hour_of_day',
                              'value'),
                # input format
                header = FALSE,
                na.strings = '<NULL>',
                sep = ';',
                dec = '.'
               )
close(f)

# quick-and-dirty argument processing,
# production code would likely use packages like optParse
logging <- '--logging' %in% commandArgs(trailingOnly = TRUE)

if (nrow(t) > 0) {
  if (logging) {
    write('Got some rows, will fit a lm', stderr())
  }
  # for each run of this script sensor_id will be identical
  sensor_id <- t$sensor_id[[1]]

  lmfit = lm(t$hour_of_day ~ t$ week_of_year + t$value)

  result <- c(sensor_id, lmfit$coefficients)
  result[is.na(result)] <- '<NULL>'   # translate N/A into Cisco ParStream convention

  # echo result on stdout
  write(paste(result, collapse = ';'),   # columns are separated by ';'
        stdout())
} else {
  if (logging) {
    write('Did not receive any data to fit', stderr())
  }
}
```

As usual for registered xUDTOs, the R scripts have to be placed in the directory for user-defined function, using the global option `udfLibraryPath` (see section 13.2.1, page 122).

If the library path is defined as follows:

```
udfLibraryPath = udf
```

and the command for the R script type is defined as follows:

```
[external.R]
command = Rscript %script
```

and the xUDTO was registered to call the R script `lm.R` and the server executing the command was started in the directory `/opt/cisco/kinetic/parstream-database/bin`, the resulting command called for each xUDTO call will be:

```
Rscript /opt/cisco/kinetic/parstream-database/bin/udf/lm.R
```

Input Processing

As shown in the example `lm.R` above, R scripts should typically use `read.table()` to process Cisco ParStream input. Beside the mapping of the column names and type, which is described below, you should in general specify the following input format properties in `read.table()`:

```
t <- read.table(...
                header = FALSE,
                na.strings = '<NULL>',
                sep = ';',
                dec = '.'
               )
```

Here we specify that

- We have no header in the input data
- `<NULL>` is the NULL value
- The semicolon is the column separator
- A dot is used as decimal point

Depending of the concrete input not all properties might be necessary in all cases (e.g. you don't have to specify the floating-point character if no floating point values are used).

In addition, we have to map column names and types from Cisco ParStream to R.

The following table describes the suggested best mappings from Cisco ParStream data types to R data types upon input:

ParStream Type	Format	R Type
`UINT8`	unsigned integer	`integer`
`UINT16`	unsigned integer	`integer`
`UINT32`	unsigned integer	`integer`
`UINT64`	unsigned integer	`integer`
`INT8`	signed integer	`integer`
`INT16`	signed integer	`integer`
`INT32`	signed integer	`integer`
`INT64`	signed integer	`integer`
`FLOAT`	floating-point value	`double`
`DOUBLE`	floating-point value	`double`
`VARSTRING`	"*<characters>*"	`character`
`BLOB`	"*<characters>*"	`character`
`SHORTDATE`	YYYY-MM-DD	`Date`
`DATE`	YYYY-MM-DD	`Date`
`TIME`	HH:MM:SS[.MS]	`Date`
`TIMESTAMP`	YYYY-MM-DD HH:MM:SS[.MS]	`Date`

In this example, we map data read from the following input table:

```
CREATE TABLE Sensors (
  sensor_id VARSTRING COMPRESSION HASH64 INDEX EQUAL,
  week_of_year INT8,
  day_of_week INT8,
  hour_of_day INT8,
  xvalue UINT64
)
PARTITION BY sensor_id
DISTRIBUTE EVERYWHERE;
```

as follows:

```
t <- read.table(...
                # input schema
                colClasses = c('character',
                                 'integer',
                                 'integer',
                                 'integer',
                                 'numeric'),
                 col.names = c('sensor_id',
                                'week_of_year',
                                'day_of_week',
                                'hour_of_day',
                                'value'),
                 ...
                )
```

For the first four arguments we use the proposed mappings having the same names in Cisco ParStream and R. For the last input column we map to a different name not using the suggested types mapping. If you want to pass data from an existing table to a pre-defined R script this is a typical case. Here, for example, it makes sense that the R script uses type `numeric` because values might be both integral and floating-point values, so that the mapping from `UINT64` to `numeric` also fits fine.

Output Processing

For output you can use any write statement, given that any `NA` values are mapped to `<NULL>`, and that columns are separated by a semicolon. For example, according to the example above:

```
result[is.na(result)] <- '<NULL>'
write(paste(result, collapse = ';'), stdout())
```

will produce output rows with a sensor id and the three floating-point coefficients of the linear model.

Again R data types have to be mapped to corresponding Cisco ParStream data types to fit a corresponding `RETURNS`. We suggest the following mappings:

R Type	ParStream Type	Comment
`integer`	`[U]INT<8|16|32|64>`	
`double`	`FLOAT, DOUBLE`	
`character`	`VARSTRING`	
`Date`	`TIMESTAMP`	
`logical`	`???`	no direct mapping, needs to be mapped on another (e.g. integer) type
`complex`	–	unsupported, real and imaginary components need to be stored separately
`NULL`	–	unsupported, all values are typed in Cisco ParStream

Note:

- Cisco ParStream types don't have a special type for NULL. Instead, NULL is a special value for the different types. For this reason, to return 8-bit values you should map them to INT16 or UINT16.
- Output rows have to end with a newline to get processed.

As RETURNS clause for the lm_R function compatible with the output sensor id and 3 floating point coefficients the following works:

```
RETURNS(sensor_id VARSTRING, c0 DOUBLE, c1 DOUBLE, c2 DOUBLE)
```

Error Handling

As usual for external user-defined table operators, any output written to stderr will be written to the log file of the server executing the script. Output to stderr can for example be done in R as follows:

```
    write('Did not receive any data to fit', stderr())
```

Calling R Scripts defined as xUDTOs

To call R scripts, registered as xUDTOs, you have to use an ON clause. For example, calling the R script lm.R defined as xUDTO lm_R() above look as follows:

```
SELECT * FROM lm_R(ON (SELECT sensor_id, week_of_year,
                              day_of_week, hour_of_day, xvalue
                              FROM Sensors)
                   PARTITION BY sensor_id
                   RETURNS(sensor_id VARSTRING,
                           c0 DOUBLE, c1 DOUBLE, c2 DOUBLE)
                  )
         ORDER BY sensor_id
```

By adding a COMMAND_LINE_ARGS clause you can pass command-line arguments to the call of the R script (used as %args in the corresponding command defined for this script type). For example:

```
SELECT * FROM lm_R(ON (SELECT sensor_id, week_of_year,
                              day_of_week, hour_of_day, xvalue
                              FROM Sensors)
                   PARTITION BY sensor_id
                   COMMAND_LINE_ARGS('--logging')
                   RETURNS(sensor_id VARSTRING,
                           c0 DOUBLE, c1 DOUBLE, c2 DOUBLE)
                  )
         ORDER BY sensor_id
```

SQL Coverage

In principle, Cisco ParStream complies with the SQL 2008 standard. However, to support special features and deal with specific limitation, Cisco ParStream also has additional commands and restrictions. This chapter gives a brief overview of the supported types and commands. See the *SQL Reference Guide* for details.

Supported Keywords, Functions, and Operators

Keyword(s)	Description	See Section
+, -, *, /	Arithmetic operators	21.2.13 page 249
=, <>, !=	Check for equality	21.2.5 page 247
<, <=, >, >=	Comparisons	21.2.5 page 247
[]	Check whether specific bit is set(v[2] is equivalent to BIT(v,2))	21.2.25 page 252
~, ~*, !~, !~*	Regular expression matching	21.2.9 page 248
ADD COLUMN	Part of `ALTER TABLE` command	21.2.43 page 259
ALL	Part of the `UNION` declaration	21.2.39 page 257
ALTER SYSTEM	Part of a `KILL` or `SET` command	21.2.47 page 259
ALTER TABLE	Schema/Metadata modifications	21.2.43 page 259
AND	Logical `AND` and `BETWEEN AND`	21.2.6 page 248
AS	Alias name	21.2.20 page 251
ASC	Ascending sorting	21.2.3 page 247
AVG	Average value	21.2.14 page 250
BETWEEN AND	Value in specific range	21.2.10 page 249
BIT	Check whether specific bit is set (BIT(v,2) is equivalent to v[2])	21.2.25 page 252
BY	See `ORDER BY`	21.2.3 page 247
CASE WHEN THEN [ELSE] END	Replace multiple values by other values	21.2.21 page 251
CAST	Explicit type conversion	21.2.29 page 254
COALESCE	First non-NULL value	21.2.24 page 252
CONTAINS	Checks whether a string contains a substring	21.2.8 page 248
COUNT	Number of ...	21.2.16 page 250
CREATE TABLE	Create new table	21.2.42 page 259
CROSS	See `JOIN`	21.2.38 page 256
DATE_PART	Part of date or time value/column	21.2.28 page 253

Keyword(s)	Description	See Section
DATE_TRUNC	Date or time rounded	21.2.28 page 253
DAYOFMONTH	Extracts the day out of a date or timestamp value as integer.	21.2.28 page 253
DAYOFWEEK	Extracts the day of the week out of a date or timestamp value as integer.	21.2.28 page 253
DAYOFYEAR	Extracts the day of year out of a date or timestamp value as integer.	21.2.28 page 253
DEFAULT	Allowed as value in `SET` statements to recover the default value for the given setting	21.2.45 page 259
DESC	Descending sorting	21.2.3 page 247
DISTINCT	Without duplicates	21.2.17 page 250
ELSE	See `CASE WHEN`	21.2.21 page 251
END	See `CASE WHEN`	21.2.21 page 251
EPOCH	Extracts the UNIX timestamp out of a date or timestamp value as integer.	21.2.28 page 253
EXTRACT	Part of date or time value/column	21.2.28 page 253
FIRST	First result	21.2.31 page 254
FLOOR	Round down floating-point value to integral value	21.2.26 page 253
FROM	Table identifier	21.2.2 page 247
FULL	See `JOIN`	21.2.38 page 256
GROUP BY	Grouping	21.2.18 page 251
HAVING	Result filter	21.2.19 page 251
HIGH	Value used to set the priority of execution tasks	21.2.37 page 256
HOUR	Extracts the hour out of a time or timestamp value as integer.	21.2.28 page 253
IF	Conditional value (simple CASE clause)	21.2.22 page 252
IFNULL	Replacement for NULL values	21.2.23 page 252
IN	Possible list of values or check against subquery results	21.2.11 page 249
INNER	See `JOIN`	21.2.38 page 256
INSERT INTO	Insert data from other tables	21.2.40 page 258
IMPORTPRIORITY	Priority of subsequently issued import tasks	21.2.37 page 256
JOIN	Combine multiple tables	21.2.38 page 256
IS [NOT] NULL	Check for NULL values	21.2.12 page 249
LEFT	See `JOIN`	21.2.38 page 256
LIKE	(sub)string matching	21.2.7 page 248
LIMIT	Limit resulting rows	21.2.30 page 254

Keyword(s)	Description	See Section
LOW	Value used to set the priority of execution tasks	21.2.37 page 256
LOWER	Lowercase letters	21.2.27 page 253
LOWERCASE	Lowercase letters	21.2.27 page 253
MATCHES	Regular expression matching	21.2.9 page 248
MAX	Maximum value	21.2.14 page 250
MEDIAN	MEDIAN of values	21.2.33 page 255
MEDIUM	Value used to set the priority of execution tasks	21.2.37 page 256
MILLISECOND	Extracts the millisecond out of a time or timestamp value.	21.2.28 page 253
LOW	Value used to set the priority of execution tasks	21.2.37 page 256
MIN	Minimum value	21.2.14 page 250
MINUTE	Extracts the minute out of a time or timestamp value as integer.	21.2.28 page 253
MOD	Modulo operator	21.2.13 page 249
NOT	Logical "NOT"	21.2.6 page 248
NULL	no value defined	21.2.12 page 249
OFFSET	Skip some results	21.2.30 page 254
OR	Logical "OR"	21.2.6 page 248
ORDER BY	Sorting	21.2.3 page 247
OUTER	See `JOIN`	21.2.38 page 256
PERCENTILE_CONT	PERCENTILE_CONT of values	21.2.33 page 255
PERCENTILE_DISC	PERCENTILE_DISC of values	21.2.33 page 255
QUARTER	Extracts the quarter out of a date or timestamp value as integer.	21.2.28 page 253
QUERYPRIORITY	Priority of subsequently issued query tasks	21.2.37 page 256
RIGHT	See `JOIN`	21.2.38 page 256
SECOND	Extracts the second out of a time or timestamp value as integer.	21.2.28 page 253
SELECT	Basic query command	21.2.2 page 247
SET	Change session settings	21.2.45 page 259
SHL	Shift-left	21.2.25 page 252
STDDEV_POP	Population standard deviation	21.2.32 page 255
SUM	Sum of values	21.2.14 page 250
TABLE	See `CREATE TABLE`	21.2.42 page 259
TAKE	Additional columns for MIN and MAX	21.2.15 page 250
THEN	See `CASE WHEN`	21.2.21 page 251

Keyword(s)	Description	See Section
TRUNC	Truncate floating-point value to integral value	21.2.26 page 253
UNION	Combine multiple sets	21.2.39 page 257
UPPER	Uppercase letters	21.2.27 page 253
UPPERCASE	Uppercase letters	21.2.27 page 253
WEEK	Extracts the week out of a date or timestamp value as integer.	21.2.28 page 253
WHEN	See `CASE WHEN`	21.2.21 page 251
WHERE	Basic result filter	21.2.4 page 247
XOR	Bit-wise XOR	21.2.25 page 252
YEAR	Extracts the year out of a date or timestamp value as integer.	21.2.28 page 253

Commands

Simple Select Commands

Basic Selects

For example:

```
SELECT * FROM MyTable;
SELECT a, b, c FROM MyTable;
```

ORDER-BY Clause

For example:

```
SELECT * FROM MyTable ORDER BY ...
SELECT * FROM MyTable ORDER BY Name DESC, Price ASC;
```

WHERE Clause

For example:

```
SELECT * FROM MyTable WHERE ...
```

Comparisons: =, <>, !=, <, <=, >, >=

For example:

```
SELECT * FROM Hotels WHERE Name = 'Marriott';
SELECT * FROM MyTable WHERE NumVal != 1 AND City <> 'Cologne';
SELECT val FROM MyTable WHERE val > 0;
SELECT birthday FROM MyTable WHERE birthday <= date '2014-12-20';
```

Note: Both <>and != are supported.

Logical Operations: AND, OR, NOT

For example:

```
SELECT * FROM MyTable WHERE NOT id = 7
SELECT * FROM MyTable WHERE ( id = 4 OR id = 5 ) AND NOT id = 6
```

LIKE Clause

For example:

```
SELECT * FROM MyTable WHERE City LIKE 'Rom%';
SELECT * FROM MyTable WHERE City LIKE '_om_';
```

Matches against the whole string with the following wildcards:

- Character “_” matches any single character
- Character “%” matches any sequence of zero or more characters

CONTAINS Clause

For example:

```
SELECT * FROM MyTable WHERE Name CONTAINS 'rr';
```

Searches for a substring

Regular Expression Matching: MATCHES, ~, !~, ~*, !~*

For example:

```
SELECT Name FROM MyTable WHERE Name MATCHES '.*rr.*';
SELECT * FROM MyTable WHERE Name MATCHES '[[:alpha:]].*r{2}i?[^qx]*';
SELECT * FROM MyTable WHERE Name ~ '[[:alpha:]].*r{2}i?[^qx]*';
SELECT * FROM MyTable WHERE Name !~* '[[:alpha:]].*R{2}i?[^q-x]*';
```

Note:

The regex syntax follows in all cases an egrep style for the whole value. For example:

`'[[:alpha:]].*r{2}i?[^qx]*'`

means:

alphanumeric letter
followed by any number (“`*`”) of any letter (“`.`”)
followed by 2 `r`’s
optionally followed by an `i`
followed by any letter except `q` and `x`

`~`, `~*`, `!~`, `!~*` have the following meanings

`~` is equivalent to MATCHES
`~*` ignores case
`!~` and `!~*` search for values NOT matching the pattern

BETWEEN-AND Clause

For example:

```
SELECT num FROM MyTable WHERE num BETWEEN 5000 AND 9000;
SELECT name FROM MyTable WHERE name BETWEEN 'A' AND 'zzz';
```

IN Clause

For example:

```
SELECT * FROM MyTable WHERE City IN ('Rome', 'Nice');
SELECT * FROM MyTable WHERE Val NOT IN (5900, 7777);
```

Note: IN also can be used with sub-queries (see section 21.2.35, page 256 for details).

NULL Clause

For example:

```
SELECT * FROM MyTable WHERE ... IS NULL;
SELECT * FROM MyTable WHERE ... IS NOT NULL;
```

Note that for strings empty values are interpreted as NULL values.

Arithmetic Operations: +, -, *, /, MOD

For example:

```
SELECT 2 * 33.3 + (7/2) -10;
SELECT Price*Num,*,3e4-1733 FROM Hotels WHERE Price * Num > 3e4 + -1733;
SELECT (2*33.3+(7/2)-10) MOD 7;
SELECT MOD(2*33.3+(7/2)-10, 7);
SELECT DateVal, DateVal + 7 AS OneWeekLater FROM MyTable;
```

Note that you can call MOD for date values.

Aggregates: MIN, MAX, SUM, AVG

For example:

```
SELECT SUM(Price) FROM MyTable;
SELECT AVG(Price) FROM MyTable;
SELECT MIN(Price) FROM MyTable;
SELECT MAX(Price) FROM MyTable;
```

Note:

- AVG and SUM are applicable on all numeric data types.
- MIN and MAX are callable on numeric data types, strings, hashed values, and date/time types.

TAKE Clause

Where aggregates are used, there is a support to query corresponding other columns where an aggregate applies.

For example:

```
SELECT MIN(Price), TAKE(Hotel), TAKE(City) FROM Hotels;
```

- yields the minimum Price and for one of the hotels with this price the values in columns Hotel and City.
- If no row matching row exists (e.g. because SUM is used), the values are NULL.

COUNT Clause

For example:

```
SELECT COUNT(*) FROM MyTable;
SELECT COUNT(*) AS NumRows FROM MyTable;
SELECT COUNT(Price) FROM MyTable;
SELECT COUNT(Price) AS NumPrices FROM MyTable;
SELECT COUNT(DISTINCT Price) FROM MyTable;
```

DISTINCT Clause

For example:

```
SELECT DISTINCT City FROM MyTable;
SELECT DISTINCT City, Bed, Seaview FROM MyTable WHERE ...
SELECT COUNT(DISTINCT Price) FROM MyTable;
```

GROUP-BY Clause

For example:

```
SELECT City FROM MyTable GROUP BY City;
SELECT Hotel, COUNT(DISTINCT City), MAX(Num) FROM MyTable GROUP BY Hotel;
```

HAVING Clause

For example:

```
SELECT * FROM MyTable HAVING Val < 8800;
SELECT City, SUM(Num) AS Nums FROM Hotels GROUP BY City HAVING Nums > 15;
```

Note: In Cisco ParStream the HAVING clause is a filter working as post processor on the results of a SELECT statement.

Thus, it can be used similar to WHERE clauses with all statements, not only with aggregates and GROUP BY.

AS Clause

AS to rename column

For example:

```
SELECT Price AS P FROM MyTable;
SELECT SUM(Num) AS SumSeaview FROM Hotels WHERE Seaview = 1;
```

AS as ID used in HAVING Clauses

For example:

```
SELECT Price AS P FROM MyTable HAVING P > 10;
```

AS in FROM clause

For example:

```
SELECT * FROM MyTable AS T;
```

CASE WHEN Clause

For example:

```
SELECT CASE Val WHEN 'Single' then 'S'
                WHEN 'Twin' THEN 'T'
                ELSE 'D' END FROM MyTable;
SELECT CASE WHEN Val = 'Single' THEN 1 ELSE 2 END FROM MyTable;
```

IF Clause

For example:

```
SELECT IF(Val = 'Single',1,2) FROM MyTable;
```

Is a shortcut for:

```
SELECT CASE WHEN Val = 'Single' THEN 1 ELSE 2 END FROM MyTable;
```

IFNULL Clause

Replacement for NULL values.

For example:

```
SELECT IFNULL(StringVal,'no value') FROM MyTable;
SELECT IFNULL(IntVal,-1) FROM MyTable;
```

COALESCE Clause

First non-NULL value.

For example:

```
SELECT COALESCE (NewPrice, OldPrice) from Pricelist;
```

Numeric Operations: [], BIT, XOR, SHL

For example:

```
SELECT Val SHL 2 FROM MyTable;
```

yields Val shifted to the left by 2 bits

```
SELECT Val XOR 2 FROM MyTable;
```

yields bit-wise XOR of Val and 000010

```
SELECT BIT(Val,1) FROM MyTable;
```

yields whether second bit (bit with index 1) is set

```
SELECT Val[1] FROM MyTable;
```

yields whether second bit (bit with index 1) is set

Floating-Point Operations: TRUNC, FLOOR

For example:

```
SELECT TRUNC(Val) from MyTable;
```

Truncate floating-point value to integral value (`TRUNC(4.7)` yields `4`, `TRUNC(-7.1)` yields `-7`)

```
SELECT FLOOR(Val) from MyTable;
```

rounds down floating-point value to integral value (`FLOOR(4.7)` yields `4`, `FLOOR(-7.1)` yields `-8`)

String Operations: LOWER, UPPER, LOWERCASE, UPPERCASE

For example:

```
SELECT LOWER(firstname), UPPER(lastname) FROM MyTable;
SELECT LOWERCASE(firstname), UPPERCASE(lastname) FROM MyTable;
```

Date/Time Operations: DATE_PART, DATE_TRUNC, EXTRACT, unary extraction functions

For example:

```
SELECT DATE_PART('MONTH',Val) FROM MyTable;
```

- extracts a specific part out of a date or time value
- first argument can be: 'YEAR', 'MONTH', 'DAY', 'HOUR', 'MINUTE', 'SECOND', 'MILLISECOND', 'DOW', 'DOY', 'EPOCH', 'ISODOW', 'ISOYEAR', 'QUARTER' or 'WEEK'
- second argument Val must be the name of a column/field with type date, shortdate, time, or timestamp

```
SELECT DATE_TRUNC('MONTH',Val) FROM MyTable;
SELECT DATE_TRUNC(2000,Val) FROM MyTable;
```

- rounds down to a specific part out of a date or time value
- first argument can be: 'YEAR', 'MONTH', 'DAY', 'HOUR', 'MINUTE', 'SECOND', 'MILLISECOND', 'QUARTER', 'WEEK' or an integer value interpreted as milliseconds

- second argument Val must be the name of a column/field with type date, time, or timestamp

```
SELECT EXTRACT(WEEK FROM Val) FROM MyTable
```

- extracts a specific part out of a date or time value
- first argument can be: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, MILLISECOND, DOW, DOY, EPOCH, ISODOW, ISOYEAR, QUARTER or WEEK
- second argument Val must be the name of a column/field with type date, shortdate, time, or timestamp

```
SELECT DAYOFMONTH(Val) FROM MyTable
SELECT DAYOFWEEK(Val) FROM MyTable
SELECT DAYOFYEAR(Val) FROM MyTable
SELECT EPOCH(Val) FROM MyTable
SELECT HOUR(Val) FROM MyTable
SELECT MILLISECOND(Val) FROM MyTable
SELECT MINUTE(Val) FROM MyTable
SELECT MONTH(Val) FROM MyTable
SELECT QUARTER(Val) FROM MyTable
SELECT SECOND(Val) FROM MyTable
SELECT WEEK(Val) FROM MyTable
SELECT YEAR(Val) FROM MyTable
```

CAST Clause

For example:

```
SELECT CAST (myTimestamp AS DATE) FROM MyTable;
```

LIMIT and OFFSET Clause

For example:

```
SELECT * FROM Hotels ORDER BY City, Name LIMIT 3;
```

stop output after 3rd row

```
SELECT * FROM Hotels ORDER BY City, Name LIMIT 3 OFFSET 3;
```

start output with 4th row and stop after 6th row

FIRST Clause

For example:

```
SELECT FIRST(name) FROM MyTable WHERE Val IS NULL;
SELECT FIRST(firstname), FIRST(lastname) FROM people WHERE income > 60000;
```

- yields only one of multiple possible results
- If multiple results are possible it is undefined which result is taken
- If no result exists, NULL is returned

STDDEV_POP Clause

For example:

```
SELECT STDDEV_POP(Price) FROM MyTable;
```

Computes the population standard deviation.

MEDIAN, PERCENTILE_CONT, PERCENTILE_DISC

For example:

```
SELECT MEDIAN(Price) FROM MyTable;
```

Computes the median of `Price`.
For example:

```
SELECT PERCENTILE_CONT(Price, 0.22) FROM MyTable;
```

Computes the 0.22 percentile of `Price` after doing linear interpolation.
For example:

```
SELECT PERCENTILE_DISC(Price, 0.22) FROM MyTable;
```

Computes the 0.22 discrete percentile of `Price`, which is a value from the set of input values.

Complex SELECT Commands

This section covers all complex select commands currently officially supported (queries from multiple tables and with sub queries).

IN Statement with Sub-Queries

For example:

```
SELECT * FROM MyTable WHERE Name IN
    (SELECT Name FROM MyTable WHERE Val  IS NOT NULL);
```

Nested SELECT Commands

You can use the result of one SELECT as input for a second SELECT. For example:

```
SELECT * FROM (SELECT * FROM MyTable);
SELECT * FROM (SELECT City, (CASE WHEN Bed = 'Single'
                             THEN 1
                             ELSE 2 END)*Num
                            AS FreeBeds
               FROM Hotels)
       WHERE FreeBeds > 0;
```

PRIORITY Statements

For example:

```
SET QueryPriority TO HIGH;
SELECT COUNT(DISTINCT(lastname)) FROM employee;
SET ImportPriority TO HIGH;
INSERT INTO employee SELECT * FROM employee WHERE lastname = 'Doe';
```

JOIN Statements

For example:

```
SELECT A.*, B.* FROM A
       INNER JOIN B ON A.userid = B.userid;
SELECT lastname, departmentName FROM employee
       LEFT OUTER JOIN department ON employee.departmentID = department.id;
SELECT City.name, Customers.customer FROM Customers
       RIGHT OUTER JOIN City ON Customers.cityId = City.id;
SELECT City.name, Customers.customer FROM Customers
       FULL OUTER JOIN City ON Customers.cityId = City.id;
```

```
SELECT lastname, departmentName FROM employee
       CROSS JOIN department;
```

UNION Statements

You can combine the results of multiple queries into a single result set if the different queries all return matching data.

For example:

```
SELECT id FROM tabA
       UNION ALL SELECT id FROM tabB;

SELECT tabA.a AS id FROM tabA
       UNION DISTINCT SELECT tabB.b AS id FROM tabB;

SELECT A.id AS f FROM tabA A
       UNION SELECT B.ref_id AS f FROM tabB.B;

(SELECT id FROM tabA ORDER BY salary DESC LIMIT 3)
       UNION (SELECT id FROM tabB ORDER BY salary ASC LIMIT 3)
       ORDER BY id LIMIT 2;

SELECT CAST(stock_office AS UINT64) AS numItems FROM tabA
       UNION SELECT stock AS numItems FROM tabB;

SELECT AVG(Rev) AS ARev, CAST(SUM(Rev) AS DOUBLE) AS SRev, PageName
    FROM WebVisits
    GROUP BY PageName
UNION
  SELECT AvgRev AS ARev, CAST(NULL AS DOUBLE) AS SRev, URL AS PageName
    FROM googleStatistics
    GROUP BY PageName
ORDER BY PageName;
```

INSERT INTO Command

Tables can be filled by various methods:

- You can use the `parstream-import` executable, which reads data from CSV files.
- You can use the **Streaming Import Interface**, a C and Java API.
- You can use the **INSERT INTO** statement.
  For example:

```
INSERT INTO AveragePrices
            SELECT city, SUM(price)/COUNT(*) AS price, 1 AS quarter
                   FROM Hotels
                   WHERE week BETWEEN 1 AND 13
                   GROUP BY city;
```

Creating and Changing Tables

Cisco ParStream supports creating tables with an extended standard conforming SQL statement. In addition, you can add additional columns.

Creating Tables

Cisco ParStream's CREATE TABLE statement is an extension of the standard SQL CREATE TABLE statement.

An example of a simple CREATE TABLE statement can be found below. Please refer to the manual for the full syntax.

```
CREATE TABLE Hotels
(
  City VARSTRING (1024) COMPRESSION HASH64 INDEX EQUAL,
  Hotel VARSTRING (100) COMPRESSION HASH64 INDEX EQUAL,
  Price UINT16 INDEX RANGE,
  Num UINT8
)
PARTITION BY City, Hotel
DISTRIBUTE EVERYWHERE;
```

Adding Columns

```
ALTER TABLE Hotels ADD COLUMN Phone VARSTRING;
ALTER TABLE Hotels ADD COLUMN HotelId UINT16 DEFAULT 42;
```

Other Commands

Set Commands

For example:

```
SET LimitQueryRuntime TO 10000
SET ExecTree.MonitoringMinLifeTime = 5000
SET QueryPriority TO DEFAULT
```

Alter System Set Commands

For example:

```
ALTER SYSTEM SET mappedFilesMax TO 10000
ALTER SYSTEM SET mappedFilesCheckInterval = DEFAULT
```

Alter System Kill Commands

For example:

```
ALTER SYSTEM KILL queryId
```

Optimization Settings

Rewrite Optimizations

With the *optimization.rewrite* settings query optimizations can be enabled or disabled explicitly.
For example:

```
SET optimization.rewrite.all = enabled;
SELECT A.*, B.* FROM A INNER JOIN B ON A.userid = B.userid;
SET optimization.rewrite.all = individual;
SET optimization.rewrite.joinElimination = disabled;
SET optimization.rewrite.hashJoinOptimization = disabled;
SET optimization.rewrite.mergeJoinOptimization = disabled;
SELECT A.*, B.* FROM A INNER JOIN B ON A.userid = B.userid;
```

Data Types

Strings

For string, Cisco ParStream provides type VARSTRING. An empty string always is interpreted as NULL. In string literals, you can escape single quotes. For example:

```
'Say \'Hello\' to me'
```

Numeric Values

For integral numeric type you have signed and unsigned types with 8, 16, 32, and 64 bits:

- INT8, INT16, INT32, INT64
- UINT8, UINT16, UINT32, UINT64

Note that NULL is a special value inside the range of these types. For this reason, the range of supported values lacks a value that is typically supported. The resulting ranges are as follows:

Type	Minimum	Maximum	NULL
INT8	-128	+126	+127
UINT8	0	+254	+255
INT16	-32768	+32766	+32767
UINT16	0	+65534	+65535
INT32	-2147483648	+2147483646	+2147483647
UINT32	0	+4294967294	+4294967295
INT64	-9223372036854775808	+9223372036854775806	+9223372036854775807
UINT64	0	+18446744073709551614	+18446744073709551615

Thus, if you need all bits for your values, you have to use the next larger type. For example, to have all values from -128 til +127 you have to use type int16.

For floating-point numeric values you have the following types:

- float, double

You can use scientific notation to specify values. For example:

```
SELECT * from Hotels WHERE Price > 1e4;
```

Date and Time Values

For Date and Time values you have types time, timestamp, date, shortdate:

- Type date:
  - 4 byte data type counting days since 0.0.0000
  - Supported minimum: 00.00.0000
  - Supported maximum: 31.12.9999
  - Literals: date ’2010-02-22’ or: date ’0222-1-1’
    (4 digits for year required)
- Type shortdate:
  - 2 byte data type counting days since 01.01.2000
  - Minimum: 01.01.2000
  - Maximum: 05.06.2178
  - Literals: shortdate ’2010-02-22’ or: shortdate ’2000-1-1’
    (4 digits for year required)
- Type time:
  - 4 byte data type counting milliseconds
  - Minimum: 00:00:00 or 00:00:00.000
  - Maximum: 23:59:59.999
  - Format: hh:mm:ss[.uuu]
  - Literals: time ’00:00:00’ or: time ’22:25:44.007’ or: time ’22:25:44.3’
    (2 digits for hour, minutes, seconds required; “.3” means 300 milliseconds)
- Type timestamp:

  - 8 byte data type counting milliseconds
  - Supported minimum: 01.01.0000 00:00:00
  - Supported maximum: 31.12.9999 23:59:59.999
  - Format: YYY-MM-DD hh:mm:ss[.uuu]
  - Literals: timestamp ’2012-1-1 00:00:00’ or: timestamp ’2012-01-31 22:25:44.07’
    (4 digits for year required; 2 digits for hour, minutes, seconds required;
    “.07” means 70 milliseconds)

Note:

- Type `date` supports + and −, adding/subtracting days.
- Types `time` and `timestamp` support + and −, adding/subtracting milliseconds.

Other Types

In addition, Cisco ParStream provides the following data types:

- **Blobs**: `BLOB`
- **Bit-Fields**: `BITVECTOR8`
- **Numeric arrays**: `MULTI_VALUE` of integral and floating-point types

SQL Language Elements

Cisco ParStream SQL

Cisco Parstream SQL implementation is Entry Level SQL-92 compliant.

Cisco ParStream supports elements of Core SQL-2003.

This chapter defines Cisco ParStream's "data structures and basic operations on SQL-data" according to Standard SQL. It provides functional capabilities for creating, accessing, and maintaining SQL-data. [see SQL/Foundation, 9075-2, 2008], Scope 1.

For syntactic elements which are not defined here, please look into the Standard SQL. Any restrictions of Standard SQL, particularly concerning mandatory, or optional features according to the Conformance Rules, apply for Cisco ParStream, except, it is explicitly mentioned here.

Syntactic elements of Standard SQL which are differently realized in Cisco ParStream are additionally marked here with a warning like this.

Syntactic elements of Cisco ParStream which are not part of the SQL Standard are additionally marked here in the text with a dot like this.

Supported SQL Keywords

The following table gives an overview of the supported keywords, functions, and operators.

Keyword(s)	Description	See
`+, -, *, /`	Arithmetic operators	
`=, <>, !=`	Check for equality	
`<, <=, >, >=`	Comparisons	
`[]`	Check whether specific bit is set (`v[2]` is equivalent to `BIT(v,2)`)	
`~, ~*, !~, !~*`	Regular expression matching	
`ADD COLUMN`	Part of `ALTER TABLE` command	page 296
`ALL`	Part of the UNION declaration	page 327
`ALTER SYSTEM`	Part of a KILL or SET command	page 375
`ALTER TABLE`	Schema/Metadata modifications	page 370
`AND`	Logical "and" and see `BETWEEN AND`	page 337
`AS`	Alias name	page 328
`ASC`	Ascending sorting	page 332
`AVG`	Average value	page 347
`BETWEEN AND`	Value in specific range	
`BIT`	Check whether specific bit is set (BIT(v,2) is equivalent to v[2])	

Keyword(s)	Description	See
`BY`	See `ORDER BY`	
`CASE WHEN THEN [ELSE] END`	Replace multiple values by other values	page 350
`CAST`	Explicit type conversion	page 299
`COALESCE`	First non-NULL value	page 350
`CONTAINS`	Check whether a string contains a substring	
`COUNT`	Number of	
`CREATE TABLE`	Create new table	page 278
`CROSS`	See `JOIN`	page 330
`CURRENT_DATE`	Gives the date the query started as UTC	page 300
`CURRENT_TIME`	Gives the time the query started as UTC	page 300
`CURRENT_TIMESTAMP`	Gives the timestamp the query started as UTC	page 300
`DESC`	Descending sorting	page 332
`DATE_PART`	Extract part of a date/time value	page 300
`DATE_TRUNC`	Round down a date/time value	page 301
`DAYOFMONTH`	Extracts the day out of a date or timestamp value as integer.	page 301
`DAYOFWEEK`	Extracts the day of the week out of a date or timestamp value as integer.	page 301
`DAYOFYEAR`	Extracts the day of year out of a date or timestamp value as integer.	page 301
`DEFAULT`	Default setting for a variable	page 373
`DELETE`	Delete values from a table	page 361
`DISTINCT`	Without duplicates	
`DISTVALUES`	Distinct Values	page 301
`ELSE`	See `CASE WHEN`	
`END`	See `CASE WHEN`	
`EPOCH`	Extracts the UNIX timestamp out of a date or timestamp value as integer.	page 302
`EXTRACT`	Extract part of a date/time value	page 302
`FALSE`	Boolean value	page 337
`FIRST`	First result	page 349
`FLOOR`	Round down floating-point value to integral value	page 303
`FROM`	Table identifier	page 329
`FULL`	See `JOIN`	page 330
`GROUP BY`	Grouping	page 331
`HASH64`	Convert string to hash value	page 303

Keyword(s)	Description	See
`HAVING`	Result filter	page 332
`HIGH`	High priority setting	page 373
`HOUR`	Extracts the hour out of a time or timestamp value as integer.	page 303
`IF`	Conditional value	page 304
`IFNULL`	Replacement for NULL values	page 304
`IMPORTPRIORITY`	Priority for import execution	page 373
`IN`	Possible list of values	
`INNER`	See `JOIN`	page 330
`INSERT INTO`	Insert values from other tables	page 360
`IS [NOT] NULL`	Check for NULL values	
`JOIN`	Combine multiple tables	page 330
`LEFT`	See `JOIN`	page 330
`LIKE`	(sub)string matching	page 341
`LIMIT`	Limit resulting rows	page 332
	Limit partition access	
`LOW`	Low priority setting	page 373
`LOWER`	Lowercase letters	page 304
`LOWERCASE`	Lowercase letters	page 304
`MATCHES`	Regular expression matching	
`MAX`	Maximum value	page 347
`MEDIUM`	Medium priority setting	page 373
`MILLISECOND`	Extracts the millisecond out of a time or timestamp value.	page 304
`MIN`	Minimum value	page 347
`MINUTE`	Extracts the minute out of a time or timestamp value as integer.	page 304
`MOD`	Modulo operator	page 346
`NOT`	Logical “not”	page 337
`NULL`	no value defined	
`NOW`	Gives the timestamp the query started as UTC	page 304
`OFFSET`	Skip some results	page 332
`ON`	See `JOIN`	
`OR`	Logical “or”	page 337
`ORDER BY`	Sorting	page 332
`OUTER`	See `JOIN`	page 330
`QUARTER`	Extracts the quarter out of a date or timestamp value as integer.	page 304

Keyword(s)	Description	See
QUERYPRIORITY	Priority for query execution	page 373
RIGHT	See JOIN	page 330
SECOND	Extracts the second out of a time or timestamp value as integer.	page 304
SELECT	Basic query command	page 327
SET	Change session settings	page 373
SHL	Shift-left	
STDDEV_POP	Population Standard Deviation	page 347
SUM	Sum of values	page 347
TABLE	See CREATE TABLE	
TAKE	Additional columns for MIN and MAX	page 348
THEN	See CASE WHEN	
TRUE	Boolean value	page 337
TRUNC	Truncate floating-point value to integral value	page 305
UNION	Combine multiple sets	page 327
UPPER	Uppercase letters	page 305
UPPERCASE	Uppercase letters	page 305
WEEK	Extracts the week out of a date or timestamp value as integer.	page 305
WHEN	See CASE WHEN	
WHERE	Basic result filter	page 331
XOR	Bit-wise XOR	
YEAR	Extracts the year out of a date or timestamp value as integer.	page 305

SQL Data Types

The following sections list the available and supported data types.

In the description: Bold means the data type is part of Standard SQL. If an alias is given for a data type, then the importer configuration uses the alias and not the name (see section Value Types for details of how to specify these types).

Supported Data Types

Currently Cisco ParStream supports the following data types:

- **Integral types** for signed and unsigned integral values of different size and granularity: `INT8`, `INT16`, `INT32`, `INT64`, `UINT8`, `UINT16`, `UINT32`, `UINT64`
- **Floating-point types** of different size and granularity: `FLOAT`, `DOUBLE`
- **Date/time types:** `DATE`, `SHORTDATE`, `TIME`, `TIMESTAMP`
- **Strings and character types:** `VARSTRING`, which might be stored as hashed value ("hashed string")
- **Blobs**: `BLOB`
- **Bit-Fields**: `BITVECTOR8`
- **Numeric arrays**: `MULTI_VALUE` of integral and floating-point types

For Boolean values, integral types or bit-fields can be used. Integral types and floating-point types are so-called *numeric types*.

Integral Types

Cisco ParStream supports the following integral types:

SQL Name	ParStream Type	Size	Description
`bool`	–	–	not available, use `int8` instead
`byte`	`INT8`	1 byte	very small integer
`unsigned byte`	`UINT8`	1 byte	very small unsigned integer
`smallint`	`INT16`	2 bytes	small-range integer
`unsigned smallint`	`UINT16`	2 bytes	small-range unsigned integer
`integer`	`INT32`	4 bytes	typical choice for integer
`unsigned integer`	`UINT32`	4 bytes	typical choice for unsigned integer
`bigint`	`INT64`	8 bytes	large-range integer
`unsigned bigint`	`UINT64`	8 bytes	large-range unsigned integer
`decimal`	–	–	not available, use `uint64` instead

Note that the numeric types of Cisco ParStream use a special value to represent `NULL`, which limits the range of the storable values. As a consequence, the types have the following values:

ParStream Type	Size	Range	NULL
INT8	1 byte	-128 to +126	+127
UINT8	1 byte	0 to +254	+255
INT16	2 bytes	-32768 to +32766	+32767
UINT16	2 bytes	0 to +65534	+65535
INT32	4 bytes	-2147483648 to +2147483646	+2147483647
UINT32	4 bytes	0 to +4294967294	+4294967295
INT64	8 bytes	-9223372036854775808 to +9223372036854775806	+9223372036854775807
UINT64	8 bytes	0 to +18446744073709551614	+18446744073709551615

Thus, to be able to store all values from -128 to +127 into a column, you should use type `int16` rather than type `int8`.

See section 24.2.5, page 287 for details about defining these types.
See section 10.4.1, page 94 for details about importing values of these types.

Floating-Point Types

ParStream Type	Size	Description
DOUBLE	8 bytes	double-precision floating-point type
FLOAT	4 bytes	single-precision floating-point type

Note: We strongly recommend to usually use type `DOUBLE` for floating-point values. The reason is that due to the limited amount of bits, `FLOAT` values easier get more significant rounding errors and even small values with only a few digits after the dot might become slightly different values as they internally are mapped to the next possible valid floating-point value. For example, after inserting a `FLOAT` such as `1234.7` a query for this value might return `1234.699951`, while when using type `DOUBLE`, the output will be `1234.7`.

The types follow IEEE 754, with the exception that +/- infinity and NaN are not supported.

You can use scientific notation to specify values. For example:

```
SELECT * FROM Hotels WHERE Price > 1e4;
```

See section 24.2.5, page 287 for details about defining these types.
See section 10.4.2, page 94 for details about importing values of these types.

Date and Time Types

Cisco ParStream provides the following types for dates and times:

ParStream Type	Size	Description	NULL	Internal Format
`DATE`	4 bytes	32-bit date	`uint32 NULL`	days since 24.11.-4713
`SHORTDATE`	2 bytes	16-bit date	`uint16 NULL`	days since 01.01.2000
`TIME`	4 bytes	time of day	`uint32 NULL`	milliseconds since 00:00:00
`TIMESTAMP`	8 bytes	both date and time	`uint64 NULL`	milliseconds since 01.01.0000 00:00:00

Types `TIMESTAMP` and `TIME` are defined according to the SQL Standard and stored without any interpretation of time zones. Therefore these column types are equivalent to SQL Standard column types `TIMESTAMP WITHOUT TIME ZONE` and `TIME WITHOUT TIME ZONE`.

These types support the following ranges:

ParStream Type	Supported Minimum	Supported Maximum
`DATE`	01.01.0000	31.12.9999
`SHORTDATE`	01.01.2000	31.12.2178
`TIME`	00:00:00 or 00:00:00.000	23:59:59.999
`TIMESTAMP`	01.01.0000 00:00:00	31.12.9999 23:59:59.999

Date/Time Literals

The date/time literals have the following standard SQL conforming syntax:

Type	Literal Examples
`DATE`	`date'2010-02-22'`
	`date '222-1-1'`
`SHORTDATE`	`shortdate'2010-02-22'`
	`shortdate '2000-1-1'`
`TIME`	`time'00:00:00'`
	`time '22:25:44.007'`
	`time '9:3:0'`
`TIMESTAMP`	`timestamp'2012-1-1 00:00:00'`
	`timestamp '2012-1-31 22:5:44.7'`

Thus:

- For years, 1 to 4 digits are required.
- For time and timestamp, hours, minutes, and seconds with 1 or 2 digits are required.
- Milliseconds are optional.
- `time'23:0:0.3'` is equivalent to `time'23:00:00.300'`

In addition, you can define your own format when importing date/time values (see section 10.4.3, page 94).

Date/Time Operations

- For types date and shortdate you can use:
  - Operators = , !=, <>, <, <=, >, >= to compare dates
  - Operator +, - to add/subtract days passed as integer values
  - Operator - to process the difference of two dates (date/shortdate - date/shortdate)
  - Operator MOD to process a modulo of the date value
- For types time and timestamp you can use:
  - Operators = , !=, <>, <, <=, >, >= to compare times/timestamps
  - Operator +, - to add/subtract milliseconds passed as integer values
  - Operator - to process the difference of two times (time - time)
  - Operator - to process the difference of two timestamps (timestamp - timestamp)
- In addition, you can:
  - Add a date/shortdate with a time to get a timestamp

Interval Literals

Cisco ParStream supports day-time intervals as defined in the SQL standard ISO 9075 with a few limitations, which are listed below.

Fields in Day-time Intervals

Cisco ParStream supports day-time intervals with the following keywords:

Keyword	Description
`DAY`	Number of days
`HOUR`	Number of hours
`MINUTE`	Number of minutes
`SECOND`	Number of seconds and possibly fractions of a second

The first field in an interval expression is only limited by <interval leading field precision>. All subsequent fields are restrained as follows:

Keyword	Valid Values of Interval Fields
`DAY`	Unconstrained except by <interval leading field precision>
`HOUR`	Number of hours within the day (0-23)
`MINUTE`	Number of minutes within the hour (0-59)
`SECOND`	Number of seconds within the minute (0-59.999)

The interval literals have the following standard SQL conforming syntax:

Type	Literal Examples
DAY TO SECOND	interval ’10 10:10:10’ day to second
	interval ’-10 10:10:10.111’ day to second
DAY TO MINUTE	interval ’10 10:10’ day to minute
	interval ’-10 10:10’ day to minute
DAY TO HOUR	interval ’10 10’ day to hour
	interval ’-10 10’ day to hour
HOUR TO SECOND	interval ’10:10:10’ hour to second
	interval ’-10:10:10’ hour to second
HOUR TO MINUTE	interval ’10:10’ hour to minute
	interval ’-10:10’ hour to minute
MINUTE TO SECOND	interval ’10:10’ minute to second
	interval ’-10:10’ minute to second
DAY	interval ’10’ day
	interval ’-10’ day
HOUR	interval ’10’ hour
	interval ’-10’ hour
MINUTE	interval ’10’ minute
	interval ’-10’ minute
SECOND	interval ’10’ second
	interval ’-10’ second

The leading field precision defines how many digits the leading field supports. The default value is two. If we need to define an interval with three or more digits, we have to set the leading field precision accordingly:

```
SELECT * FROM examples WHERE ts < NOW() - INTERVAL '100' DAY(3);
SELECT * FROM examples WHERE ts < NOW() - INTERVAL '1000' DAY(4);
```

The fractional seconds precision defines how many digits the fractional seconds part supports. The default and maximum value is three. If we want to limit the digits of the fractional seconds part of the interval to fewer digits, we have to set the fractional seconds precision accordingly:

```
SELECT * FROM examples WHERE ts < NOW() - INTERVAL '0.12' SECOND(2,2);
SELECT * FROM examples WHERE ts < NOW() - INTERVAL '00:00.12' MINUTE TO
    SECOND(2);
SELECT * FROM examples WHERE ts < NOW() - INTERVAL '0.1' SECOND(2,1);
SELECT * FROM examples WHERE ts < NOW() - INTERVAL '00:00.1' MINUTE TO
    SECOND(1);
```

Valid Operators Involving Intervals

Operand 1	Operation	Operand 2	Result Type
`Timestamp`	`+ or -`	`Interval`	`Timestamp`
`Date`	`+ or -`	`Interval`	`Timestamp`
`ShortDate`	`+ or -`	`Interval`	`Timestamp`
`Time`	`+ or -`	`Interval`	`Time`
`Interval`	`+`	`Timestamp`	`Timestamp`
`Interval`	`+`	`Date`	`Timestamp`
`Interval`	`+`	`ShortDate`	`Timestamp`
`Interval`	`+`	`Time`	`Time`
`Interval`	`+`	`Interval`	`Interval`

Limitations

- Cisco ParStream only supports day-time intervals, no year-month intervals.
- Interval fractional seconds precision has a default and maximum value of three digits.
- Day-time intervals cannot be stored in a table.
- Day-time intervals cannot be the final result of a SQL result, i.e., day-time intervals can only be used in conjunction with a datetime type.
- Akin to other datetime types, intervals do not support time zone configurations.

Note:

The leading field precision has to be set if a leading field with more than two digits is used. This is different from PostgreSQL where the leading field precision cannot be set and any number of digits for the leading field is accepted.

String and Character types

Cisco ParStream provides the following types for strings

ParStream Type	Description
`VARSTRING`	non-hashed string
`VARSTRING COMPRESSION HASH64`	hashed string (with index support)

Thus, for string types, you have two options:

- use **non-hashed strings**, which is useful to hold just string values without searching for them
- use **hashed strings**, which allows to use special indices to optimize search operations

In addition, you can

- specify the length of the string

Note:

- The global option `blobbuffersize` (see section 13.2.1, page 124) defines the maximum number of bytes a string is allowed to have on import. The default value is 1,048,576 (1024*1024 or 2^{20}).

Dealing with Empty Strings

Note that empty strings are by Cisco ParStream internally stored as `NULL` values. Thus, when data is stored in Cisco ParStream, there is no difference between empty and `NULL` strings.

In fact, empty strings and `NULL` strings are handled as follows:

- Importing empty strings has the same effect as importing `NULL` as values for string columns. Because rows to import are ignored in case all the values are `NULL` this implies that you can't import rows where all string values are empty (and all other values `NULL` if any).
- To filter for empty strings, you have to use `IS NULL` or `IS NOT NULL`: For example:

  ```
  SELECT * FROM MyTable WHERE str IS NULL
  ```

  or

  ```
  SELECT * FROM MyTable WHERE str IS NOT NULL
  ```

- How queries return `NULL` strings depends on the channel and output format (see section 16.3, page 199):
  - Using the netcat port if the output format is `ASCII`, `NULL`/empty strings are returned empty strings. For example, a query for a non-hashed string `str` and a hashed string `hstr` column might return:

    ```
    #str;hstr
    "";"str is empty"
    "hstr is empty";""
    ```

  - Using the netcat port if the output format is `JSON`, `NULL`/empty strings are returned as `null`: For example:

    ```
    {"rows" : [
    {"str" : null,"hstr" : "str is empty"},
    {"str" : "hstr is empty","hstr" : null}
    ]
    }
    ```

  - Using the netcat port if the output format is `XML`, `NULL`/empty strings are returned as empty values. For example:

    ```
    <output>
      <dataset>
        <str></str>
        <hstr><![CDATA[str is empty]]></hstr>
      </dataset>
      <dataset>
    ```

```
    <str><![CDATA[hstr is empty]]></str>
    <hstr></hstr>
  </dataset>
</output>
```

– When using Postgres to connect to Cisco ParStream, `NULL`/empty strings are returned as `NULL`. For example using psql:

```
=> \pset null NULL
Null display is "NULL".
=> select * from MyTable;
       str      |   hstr
----------------+-------------
 NULL           | str is empty
 hstr is empty  | NULL
(2 rows)
```

Blob Types

There is also a type `blob` for “binary large objects” provided, which allows the storage of binary data. Blobs are always hashed compressed:

ParStream Type	Description
`BLOB`	implicitly hashed blobs (with index support)
`BLOB COMPRESSION HASH64`	explicitly hashed blobs (with index support)

Note the following:

- The global option `blobbuffersize` (see section 13.2.1, page 124) defines the maximum number of bytes a blob is allowed to have on import. The default value is 1,048,576 (1024*1024 or 2^{20}).

Bit-Field Types

ParStream Type	Size	Description	NULL
`BITVECTOR8`	1 byte	8 bit bitvector, where each bit can be either TRUE or FALSE	All bits set to FALSE

Note how this is different from common bitvector implementations. It is not possible to set individual bits to NULL. If the NULL-behavior is undesired, treat the bitvector8 as a bitvector7, and have the remaining bit always set to TRUE.

See section 24.2.5, page 293 for details about defining these types.
See section 10.4.7, page 98 for details about importing values of these types.

MultiValues (Numeric Arrays)

A MultiValue is a field in a table that can contain an arbitrary number of numeric values of the same type (so, basically a numeric array). This feature is currently only supported for integer and floating-point values.

The benefit of MultiValues is that you don't need to know in advance how many elements you expect and that you can index and filter them. This gives the ability to represent a 1:n relation.

Suppose you are tracking some sort of flow which spans multiple servers (identified by a numeric ID). Then, you can store all of them in a multivalue holding the IDs:

```
CREATE TABLE track
(
  ....
  serverIdMultiValue UINT32 MULTI_VALUE INDEX EQUAL
  .....
)
```

Import Conventions for MultiValues

In CSV files, multivalues (see section 23.8, page 275) are expressed as comma-separated list of numeric values:

```
# flowId; serverIdMultiValue
233423;345,23454,7789,2334
```

Using MultiValues

On the SQL level, you can now run queries like the following examples:

Example 1: Identify flows which involved servers 345 and the value 2334 in the multivalue list:

```
SELECT flowId, serverIdMultiValue
       FROM track
       WHERE serverIdMultiValue = 345
             AND serverIdMultiValue = 2334;
```

The result would be:

```
# flowId; serverIdMultiValue
233423;345,23454,7789,2334
```

You can also expand the result by using a `GROUP BY` for the multivalues. For example, the query:

```
SELECT flowId, serverIdMultiValue
       FROM track
       WHERE serverIdMultiValue = 345 AND
```

```
            serverIdMultiValue = 2334
        GROUP BY flowId, serverIdMultiValue;
```

will yield the following result:

```
# flowId; serverIdMultiValue
233423;345
233423;23454
233423;7789
233423;2334
```

With `DISTVALUES()` you can process the distinct values of multivalue columns (see section 25, page 301 for details).

Limitations for MultiValues

Note the following limitations for multivalues:

- Multivalues currently should be used with an equal index.
- JOINs (see section 27.3.1, page 330) on multivalues are not supported and result in an exception.
- Multivalues currently must not be used on the right side of an import ETL-join. Neither can a tree merge be made on a table which contains one or more multivalue columns.

Table Statements

This chapter describes the details about how to define, modify and delete tables using `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, and other statements.

Overview of Table Statements

To be able to use a database table in Cisco ParStream, you have to define it with a `CREATE TABLE` command. In addition, you can modify tables with `ALTER TABLE` commands. Tables not used any more can be deleted with the `DROP TABLE` command. This section describes the principle usage of these commands.

Formulating Table Definitions

To define a table a `CREATE TABLE` statement has to be used. For example:

```
CREATE TABLE Hotels
(
  City VARSTRING COMPRESSION HASH64 INDEX EQUAL,
  Hotel VARSTRING(100) COMPRESSION HASH64 INDEX EQUAL,
  Price UINT16 COMPRESSION DICTIONARY INDEX RANGE,
)
DISTRIBUTE EVERYWHERE
```

For details of the format of this statement see section 24.2, page 278.

Performing Table Definitions

To define a table the most convenient way is to create a SQL file with the `CREATE TABLE` statement and send it as command to **one** of the nodes in a cluster.

The easiest way to do this is to use the `pnc` client (see section 12.1.1, page 112). For example after defining a corresponding command in a file `MyTable.sql`, you can send this to a server/node listening on port `9988` as follows:

```
pnc -p 9988 < MyTable.sql
```

Note: You should **always wait for and check the result** of such a request, because due to the distributed organization of Cisco ParStream databases, it might happen that the call is not possible or does not succeed. When creating a table “Address” the answer signaling success is:

```
Table 'Address' successfully created.
```

If multiple `CREATE TABLE` commands are defined in multiple files of a subdirectory, you can simply call:

```
cat table-definitions/*.sql | pnc -p 9988
```

Alternatively you can send such a command interactively from the Cisco ParStream client.

For backward compatibility, you can also define a table is a so-called PSM file in the config directory to be used as initial table definition.

For details of the `pnc` client see section 12.1.1, page 110.

Table Modifications

You can modify existing tables using a `ALTER TABLE` statement. For example:

```
ALTER TABLE Hotels ADD COLUMN Phone VARSTRING DEFAULT NULL;
```

Note: Again, you should **always wait for and check the result** of such a request, because due to the distributed organization of Cisco ParStream databases, it might happen that the call is not possible or does not succeed. When performing an `ALTER TABLE` command the answer signaling success is:

```
ALTER OK
```

Again, you have to send this statement as command to one of the nodes in a cluster.

For details of the `ADD COLUMN` command see section 24.3.2, page 296.

Table Deletion

You can delete existing tables using the `DROP TABLE` statement. For example:

```
DROP TABLE Hotels;
```

Note: You cannot drop tables that are `REFERENCED` by other tables (e.g. colocation). For details of the `DROP TABLE` command see section 24.4, page 297.

CREATE TABLE Statements

CREATE TABLE Statement Overview

This subsection describes the details of the `CREATE TABLE` statement provided by Cisco ParStream. In principle, it designed after the SQL Standard (ISO 9075) using several extensions to be able to provide the specific abilities and features Cisco ParStream offers. See section 27.7.2, page 364 for a description of the detailed grammar of `CREATE TABLE` statement in BNF (Backus-Naur Form).

Note that these definitions include schema aspects as well as import aspects (such as the column in the CSV file), which means that the syntax is extended by Cisco ParStream specific elements.

The `CREATE TABLE` statement can be passed in batch mode to a database by passing it to a client interface such as `netcat` or `pnc`. See section 12.1.2, page 114 and section 12.1.1, page 112 for examples how to use SQL files as input for data sent over the socket interface.

Simple Example

As a simple example, the following statement creates a table Hotels with 4 columns (two of them used as partitions):

```
CREATE TABLE Hotels
(
  City VARSTRING COMPRESSION HASH64 INDEX EQUAL,
  Hotel VARSTRING(100) COMPRESSION HASH64 INDEX EQUAL,
  Price UINT16 COMPRESSION DICTIONARY INDEX RANGE,
  Num UINT8 COMPRESSION SPARSE SPARSE_DEFAULT 1
)
DISTRIBUTE EVERYWHERE
```

Note that the order of both the table clauses and the column attributes matters (see the tables below and the grammar in section 27.7.2, page 364).

The minimum `CREATE TABLE` statement has to define the table name, at least one column with name and type and a distribution clause (see section 6.3, page 53). For example:

```
CREATE TABLE MinTable
(
  col UINT32
)
DISTRIBUTE EVERYWHERE;
```

All other `TABLE` clauses are optional.

Note that the command contains both pure schema information and information about how to import the data. For example, you can specify where to find input CSV files for a table:

```
CREATE TABLE SmallTable
(
  col UINT32
)
DISTRIBUTE EVERYWHERE
IMPORT_DIRECTORY_PATTERN '.*'
IMPORT_FILE_PATTERN '.*\.csv';
```

The default is to look for files that begin with the name of the table and end with `.csv`. You can still overwrite the directory and file pattern for a table via the command line (see section 13.1.1, page 116).

All clauses and attributes for the table as a whole and for specific columns are described in the upcoming subsections. See section 24.2.1, page 281 for some restrictions regarding table and column names.

Complex Example

As a more complex example, here is a CREATE TABLE statement, where almost all possible attributes are used:

```
CREATE TABLE ComplexTable (
  id UINT64 SINGLE_VALUE PRELOAD_COLUMN NOTHING SEPARATE BY NOTHING
    INDEX EQUAL MAX_CACHED_VALUES 20000 CACHE_NB_ITERATORS FALSE
    CSV_COLUMN 0 SKIP FALSE,

  binnedNumerical INT32 SINGLE_VALUE PRELOAD_COLUMN MEMORY_EFFICIENT SEPARATE
   BY NOTHING
    INDEX RANGE MAX_CACHED_VALUES 20000 CACHE_NB_ITERATORS TRUE
    INDEX_BIN_COUNT 20 INDEX_BIN_MIN MIN INDEX_BIN_MAX MAX
    CSV_COLUMN 1 SKIP FALSE,

  manualBoundaries INT32 SINGLE_VALUE PRELOAD_COLUMN MEMORY_EFFICIENT
   SEPARATE BY etlDay
    INDEX RANGE MAX_CACHED_VALUES 20000 CACHE_NB_ITERATORS TRUE
    INDEX_BIN_BOUNDARIES (10, 20, 30, 40, 50)
    CSV_COLUMN 2 SKIP FALSE,

  skippedColumn DOUBLE SINGLE_VALUE
    CSV_COLUMN 3 SKIP TRUE,

  dayColumn DATE COMPRESSION SPARSE SINGLE_VALUE PRELOAD_COLUMN COMPLETE
   SEPARATE BY NOTHING
    INDEX EQUAL MAX_CACHED_VALUES 20000 CACHE_NB_ITERATORS FALSE
   INDEX_GRANULARITY YEAR
    CSV_COLUMN 4 CSV_FORMAT 'YYYY-MM-DD' SKIP FALSE,

  etlDay UINT16 SINGLE_VALUE PRELOAD_COLUMN NOTHING SEPARATE BY NOTHING
    INDEX EQUAL MAX_CACHED_VALUES 20000 CACHE_NB_ITERATORS FALSE
    CSV_COLUMN ETL,

  nonHashedString VARSTRING (20) COMPRESSION NONE MAPPING_FILE_GRANULARITY
   256 SINGLE_VALUE
    PRELOAD_COLUMN NOTHING SEPARATE BY NOTHING
    CSV_COLUMN 5 SKIP FALSE,

  hashedString VARSTRING (255) COMPRESSION HASH64, DICTIONARY MAPPING_TYPE
   AUTO
    SINGLE_VALUE PRELOAD_COLUMN NOTHING SEPARATE BY NOTHING
    INDEX EQUAL PRELOAD_INDEX COMPLETE
    CSV_COLUMN 6 SKIP FALSE,

  bitvectorColumn BITVECTOR8 SINGLE_VALUE PRELOAD_COLUMN NOTHING SEPARATE BY
   NOTHING
    INDEX EQUAL MAX_CACHED_VALUES 20000 CACHE_NB_ITERATORS FALSE INDEX_MASK 3

  multiValueNumeric UINT8 MULTI_VALUE PRELOAD_COLUMN NOTHING SEPARATE BY
   NOTHING
    INDEX EQUAL MAX_CACHED_VALUES 20000 CACHE_NB_ITERATORS FALSE
)
PARTITION BY (id MOD 10), etlDay
```

```
DISTRIBUTE OVER etlDay WITH REDUNDANCY 2
ORDER BY binnedNumerical
IMPORT_DIRECTORY_PATTERN '.*'
IMPORT_FILE_PATTERN 'complex.*\.csv'
ETL (
     SELECT ComplexTable.id, ComplexTable.binnedNumerical,
            ComplexTable.manualBoundaries, ComplexTable.dayColumn,
            ComplexTable.nonHashedString, ComplexTable.hashedString,
            ComplexTable.bitvectorColumn, ComplexTable.multiValueNumeric,
            DATE_PART('day', dayColumn) AS etlDay
            FROM CSVFETCH(ComplexTable)
     )
;
```

Note again that the order of both the table clauses and the column attributes matters (see the tables below and the grammar in section 27.7.2, page 364).

Table and Column Names

Cisco ParStream maps columns directly to files using table and column names as file path elements (see section 5.1, page 29). For this reason, there are restrictions regarding table and column names, which are listed below.

If table and column identifiers conflict with SQL keywords (see section 28, page 378), you have to put them between quotation marks. For example, the following statement creates a table named "TABLE" with columns called "PARTITION" and "SORTED":

```
CREATE TABLE "TABLE" (
  "PARTITION" UINT8 INDEX EQUAL,
  "SORTED" UINT32 INDEX EQUAL
)
PARTITION BY "PARTITION", "SORTED"
DISTRIBUTE EVERYWHERE;
```

Restrictions for Table Names

Table names have the following restrictions:

- Table names must have at least 2 characters.
- Table names may use only the following characters: the underscore _, letters, and digits
- Table names may not start with a digit or an underscore.
- Table names may not use umlauts or unicode letters.
- Table names may not start with 'pg_' or 'ps_' because these prefixes are used for internal tables and system table (see chapter 26, page 306).

Table names are case-insensitive. Thus,

```
SELECT * FROM OneTable;
SELECT * FROM onetable;
SELECT * FROM ONETABLE;
```

query values from the same table.

Restrictions for Column Names

Column names have the following restrictions:

- Column names must have at least 2 characters.
- Column names may use only the following characters: the underscore _, letters, and digits
- Column names may not start with a digit or an underscore.
- Column names may not use umlauts or unicode letters.

Column names are case-insensitive.

Possible Clauses and Attributes inside CREATE TABLE statements

As written above, the minimum `CREATE TABLE` statement has to define the table name and at least one column with name and type. All other `TABLE` clauses and column attributes are optional (with a few exceptions where column types require certain attributes).

Note that you can still overwrite the directory and file pattern for a table via the command line (see section 13.1.1, page 116).

All clauses and attributes for the table as a whole and for specific columns are described in the upcoming subsections.

All Table Attributes

Tables can have the following clauses and attributes, specified in the order of their usage after:

`CREATE TABLE` *tabname* `(`
  columns
`)`

Clause	Description
`PARTITION BY` *columns*	List of columns or expressions that are used to partition the data. This is a comma separated list. See section 5.1, page 30 for details.
`PARTITION ACCESS` *columns*	Column names to optimize the partition access tree. See section 15.5, page 166 for details.
`LIMIT` *num*	Defines the maximum number of branched subnodes for the resulting access tree. All columns given here MUST have an index. See section 15.5, page 166 for details.
`DISTRIBUTE` ...	A specification for the distribution of the data (see section 6.3, page 53)
`ORDER BY` *columns*	A list of column names that are used to sort the physical data in the database for internal optimization (see section 15.4, page 165).
`IMPORT_DIRECTORY_PATTERN` *pat*	Global regex pattern for directory names in the import directory. Default: `.*`
`IMPORT_FILE_PATTERN` *pat*	Global regex pattern for import file names. Default: *tablename*`.*\.csv` (the file has to start with the table name and end with `.csv`)
`ETL` (*query*)	Defines *query* as ETL query for an import. See section 10.6, page 104) for details.
`ETLMERGE` *level* (*query*)	Defines *query* as ETL query for an ETL merge process for level *level* (`HOUR`, `DAY`, `WEEK`, or `MONTH`). See section 14.2, page 154 for details.

Note (again) that:

- The order matters and has to fit the order in the table above.
- You can still overwrite the directory and file pattern for a table as well as an `ETL` or `ETLMERGE` clause for a table via the command line (see section 13.1.1, page 116).
- For backward compatibility you can use here:
  - `PARTITIONED BY` instead of `PARTITION BY`
  - `DISTRIBUTED BY` instead of `DISTRIBUTE BY`
  - `SORTED BY` instead of `ORDER BY`

All Column Attributes

Columns can have the following clauses and attributes (specified in the order of their usage after their name). Only the leading name and the typename are mandatory.

Attribute	Types	Description	Default
typename	all	Column type (see section 24.2.5, page 287)	
(*length*)	strings, blobs	number of characters/bytes	1024
DEFAULT *val*	all	Defines a default value for added columns or sparse column compressions. This is currently **no** general default value used if imports don't provide a value (in that case the value is always NULL)	NULL
NOT NULL	all	Declares that the column may not contain NULL values. This is checked on all import paths. Required for referenced columns (see section 15.15.4, page 192).	Not present
UNIQUE	all	Declares that the column contains unique values only, i.e. each distinct value occurs only once. Please note that this property is not checked by Cisco ParStream because of performance reasons. Required for referenced columns (see section 15.15.4, page 192).	Not present
PRIMARY KEY	all	Declares that the column is the primary key of its table. This implies that the column is declared NOT NULL and UNIQUE. Only one column in a table may be declared as that.	Not present
COMPRESSION *values*	all	Compression (NONE and LZ4 for all types, HASH64 for hashed strings or blobs, and/or SPARSE or DICTIONARY for fixed-width columns (see below).	NONE
MAPPING_TYPE *val*	hashed types	Mapping type of the column (see section 24.2.6, page 294).	AUTO
MAPPING_FILE_GRANULARITY *val*	strings, multivalues	Number of combined entries (see section 24.2.5, page 291)	1
singularity	numeric	SINGLE_VALUE or MULTI_VALUE for multivalues (see section 23.8, page 275 and section 10.4.6, page 98)	SINGLE_VALUE
PRELOAD_COLUMN *val*	all	NOTHING, COMPLETE, or MEMORY_EFFICIENT (see section 24.2.6, page 293)	NOTHING
SEPARATE BY *val*	all	Columns for DSA optimization (see section 15.15.1, page 188)	NOTHING
REFERENCES *vals*	all	Referenced columns for DSJ optimization (see section 15.15.4, page 192)	NOTHING
INDEX *val*	all	Column index (NONE, EQUAL, RANGE; see section 24.2.6, page 293)	NONE

Attribute	Types	Description	Default
INDEX_BIN_COUNT *int*	numeric	Denotes the number of bitmaps that will be created. No binning is used, if this parameter is not set or if it is set to 0 or 1. Binning works for all numeric data types.	
INDEX_BIN_MIN *val*	numeric	Lower bound of the range of values for which bins will be created. If the smallest value of the type shall be used, then the notation "<MIN>" is allowed as well.	
INDEX_BIN_MAX *val*	numeric	Upper bound of the range of values for which bins will be created. If the greatest value of the type shall be used, then the notation "<MAX>" is allowed as well.	
INDEX_BIN_BOUNDARIES(*ints*)	numeric	Enables binning for the bitmap index. The parameter takes a comma-separated list of values; for each range between two values one bitmap will be created. The values must be sorted in ascending order. This parameter cannot be used together with indexbincout, indexbinmin, and indexbinmax.	
INDEX_MASK *int*	bitvectors	Describes which bit of a bitvector to index (only used when valuetype is a bitvector).	
INDEX_GRANULARITY*val*	date/time	Granularity at which index bitmaps are created (see section 24.2.5, page 289)	see below
PRELOAD_INDEX*val*	all	NOTHING or COMPLETE	NOTHING
DYNAMIC_COLUMNS_KEY	all	column is key in a dynamic columns table (see chapter 7, page 61	
DYNAMIC_COLUMNS_VALUE	all	column is possible value in a dynamic columns table (see chapter 7, page 61	
CSV_COLUMN *int*|ETL	all	Column number within the CSV raw data. The numbering must start with 0, and in the whole table the numbers must be consecutive. If ETL is specified, the column data must be filled using an ETL clause (see section 10.6, page 104).	see below
CSV_FORMAT *format*	Date/Time	The CSV_FORMAT clause provides the ability to define a format mask for the import data (see section 10.4.3, page 95)	
SKIP *bool*	all	TRUE means to ignore the column when importing or querying data. This can be useful for CSV imports (see section 10.3.1, page 91). The column must not be an ETL column, then.	FALSE

Note the following:

- COMPRESSION: The compression specified how the database internally stores values. Here, a couple of (type-specific) optimizations are possible:
  - The default is NONE, which cannot get combined with any other value.
  - Strings can have the value HASH64, which handles them as hashed strings (see section 23.5, page 272).
  - For blobs, compression HASH64 is always set (even if not specified).
  - For column types that have a fixed width, COMPRESSION can also be SPARSE (see section 15.10.1, page 178) or DICTIONARY (see section 15.10.2, page 179).

    * Fixed-width column types are all integer and floating-point types, BITVECTOR8, all date/time types, hashed strings, and blobs. Note that only SINGLE_VALUE types are fixed-width.
    * SPARSE compressions have an option default value, which can be specified with SPARSE_DEFAULT. If it is not defined, the column DEFAULT value is used. See section 15.10.1, page 178 for examples.
  - Every column type can have the compression value LZ4 to save space on storage. Note however, that this can cause significant performance drawbacks in other areas; so, use this compression style with care. See section 15.10.3, page 181 for details.

  If multiple compression values are specified, they can have any order separated by a comma. SPARSE and DICTIONARY can be combined with HASH64, but not with LZ4.

  For example:

```
CREATE TABLE MyTable (
  colName VARSTRING COMPRESSION HASH64,
  colSparse VARSTRING(1024) COMPRESSION HASH64, SPARSE SPARSE_DEFAULT 'a
   string',
  colDictionary VARSTRING(1024) COMPRESSION HASH64, DICTIONARY,
  colVal VARSTRING(1024) COMPRESSION LZ4, HASH64,
)
...
```

  See section 15.10, page 177 for further details.

- MULTI_VALUE: Only numeric types can be multivalues (see section 23.8, page 275).
- INDEX: Only single-value numeric types and date/time types can have a RANGE index.
- CSV_COLUMN: The default value for CSV_COLUMN is the maximum CSV_COLUMN value of the previous columns plus 1.
- Columns with NOT NULL can't have NULL as DEFAULT value, thus you have to explicitly provide a default value in this case (although it might never be used).
- It is not supported to have a DEFAULT value with user-provided hash values (MAPPING_TYPE PROVIDED), with NON NULL UNIQUE, or with PRIMARY KEY.
- For backward compatibility you can use
  - SEPARATED BY instead of SEPARATE BY

Column Types

Columns can have the following types (see Chapter 23, page 267 for general information about all data types).

Category	ParStream Type	Size
Signed Integer	`INT8`	1 byte
	`INT16`	2 bytes
	`INT32`	4 bytes
	`INT64`	8 bytes
Unsigned Integer	`UINT8`	1 byte
	`UINT16`	2 bytes
	`UINT32`	4 bytes
	`UINT64`	8 bytes
Floating-Point Types	`FLOAT`	4 bytes
	`DOUBLE`	8 bytes
Date/Time Types	`DATE`	4 bytes
	`SHORTDATE`	2 bytes
	`TIME`	4 bytes
	`TIMESTAMP`	8 bytes
Strings	`VARSTRING`	8 for hashed strings
Blobs	`BLOB`	depends on value
Bit-Arrays	`BITVECTOR8`	1 byte

The following subsections describe details about these types and especially the attributes that can be used with them.

Numeric Types and Multivalues

Numeric columns store simple data types like integers (see section 23.2, page 267) or floating-point values (see section 23.3, page 268). They support all index types and may be multivalues.

Integral columns have one of the following types: `INT8`, `INT16` `INT32`, `INT64`, `UINT8`, `UINT16` `UINT32`, `UINT64`.

Floating-point columns have one of the following types: `FLOAT`, `DOUBLE`.

Attribute	Description	Default
`COMPRESSION` *values*	Compression (`NONE`, `LZ4` for all types. `SPARSE` or `DICTIONARY`for all `SINGLE_VALUE` types, not in combination with `LZ4`).	`NONE`
`MAPPING_FILE_GRANULARITY` *val*	Number of combined entries if `MULTI_VALUE` (see section 24.2.5, page 289)	1
singularity	`SINGLE_VALUE` or `MULTI_VALUE` (see section 23.8, page 275)	`SINGLE_VALUE`
`PRELOAD_COLUMN` *val*	`NOTHING`, `COMPLETE`, or `MEMORY_EFFICIENT` (see section 24.2.6, page 293)	`NOTHING`
`SEPARATE BY` *val*	columns for DSA optimization (see section 15.15.1, page 188)	`NOTHING`
`REFERENCES` *vals*	columns for DSJ optimization (see section 15.15.4, page 191)	`NOTHING`
`INDEX` *val*	Column index (`NONE`, `EQUAL`, `RANGE`; see section 24.2.6, page 293)	`NONE`
`INDEX_BIN_COUNT` *int*	Denotes the number of bitmaps that will be created. No binning is used, if this parameter is not set or if it is set to 0 or 1. Binning works for all numeric data types.	
`INDEX_BIN_MIN` *val*	Lower bound of the range of values for which bins will be created. If the smallest value of the type shall be used, then the notation "<MIN>" is allowed as well.	
`INDEX_BIN_MAX` *val*	Upper bound of the range of values for which bins will be created. If the greatest value of the type shall be used, then the notation "<MAX>" is allowed as well.	
`INDEX_BIN_BOUNDARIES(`*ints*`)`	Enables binning for the bitmap index. The parameter takes a comma-separated list of values; for each range between two values one bitmap will be created. The values must be sorted in ascending order. This parameter cannot be used together with indexbincout, indexbinmin, and indexbinmax.	
`PRELOAD_INDEX` *val*	`NOTHING` or `COMPLETE`	`NOTHING`
`CSV_COLUMN` *int*`|ETL`	Column number within the CSV raw data. The numbering must start with 0, and in the whole table the numbers must be consecutive. If `ETL` is specified, the column data must be filled using an ETL clause (see section 10.6, page 104).	see below
`SKIP` *bool*	`TRUE` means to ignore the column when importing or querying data. This can be useful for CSV imports (see section 10.3.1, page 91). The column must not be an ETL column, then.	`FALSE`

For example:

```
CREATE TABLE NumericTable (
  id UINT64 INDEX NONE,

  binnedNumerical INT32 PRELOAD_COLUMN MEMORY_EFFICIENT
    INDEX EQUAL INDEX_BIN_COUNT 20 INDEX_BIN_MIN MIN INDEX_BIN_MAX MAX,

  manualBoundaries INT64
    INDEX RANGE MAX_CACHED_VALUES 20000 CACHE_NB_ITERATORS TRUE
    INDEX_BIN_BOUNDARIES (10, 20, 30, 40, 50),

  avg DOUBLE,
  values UINT8 MULTI_VALUE,
)
...
```

MAPPING_FILE_GRANULARITY is another time/size trade-off: For multivalues, Cisco ParStream merges several entries internally. The position for each of these merged entries is stored for faster access. This parameter describes the number of entries that are combined as one position. A higher value results in a smaller index, a lower value results in faster access.

Date/Time Types

Date/time columns use one of the following types: DATE, SHORTDATE, TIME, TIMESTAMP (see section 23.4, page 268). Internally, the values are stored as integers. They support indexing and index binning, including a specific attribute INDEX_GRANULARITY.

Attribute	Description	Default
COMPRESSION *values*	Compression (NONE, LZ4 for all types. SPARSE or DICTIONARY for all SINGLE_VALUE types, not in combination with LZ4).	NONE
PRELOAD_COLUMN *val*	NOTHING, COMPLETE, or MEMORY_EFFICIENT (see section 24.2.6, page 293)	NOTHING
SEPARATE BY *val*	columns for DSA optimization (see section 15.15.1, page 188)	NOTHING
REFERENCES *vals*	columns for DSJ optimization (see section 15.15.4, page 191)	NOTHING
INDEX *val*	Column index (NONE, EQUAL, RANGE; see section 24.2.6, page 293)	NONE
INDEX_GRANULARITY *val*	Granularity at which index bitmaps are created (see section 24.2.5, page 289)	see below
PRELOAD_INDEX *val*	NOTHING or COMPLETE	NOTHING
CSV_COLUMN *int*|ETL	Column number within the CSV raw data. The numbering must start with 0, and in the whole table the numbers must be consecutive. If ETL is specified, the column data must be filled using an ETL clause (see section 10.6, page 104).	see below
CSV_FORMAT *format*	The CSV_FORMAT clause provides the ability to define a format mask for the import data (see section 10.4.3, page 95)	
SKIP *bool*	TRUE means to ignore the column when importing or querying data. This can be useful for CSV imports (see section 10.3.1, page 91). The column must not be an ETL column, then.	FALSE

For example:

```
CREATE TABLE DateTimeTable (
  timepoint TIMESTAMP,

  day DATE COMPRESSION SPARSE INDEX EQUAL INDEX_GRANULARITY YEAR
    CSV_FORMAT 'YYYY-MM-DD',
  day2 DATE COMPRESSION DICTIONARY INDEX EQUAL INDEX_GRANULARITY YEAR
    CSV_FORMAT 'YYYY-MM-DD',
)
...
```

INDEX_GRANULARITY is provided for date/time types to control at which granularity indices are created. This trades off index size against the amount of data that needs to be fetched from column stores.

Possible values and defaults are:

Value	Description	Valid for	Default for
YEAR	Indices will be binned for each year.	DATE, TIMESTAMP	
MONTH	Indices will be binned for each month.	DATE, TIMESTAMP	
DAY	Indices will be binned for each day.	DATE, SHORTDATE, TIMESTAMP	DATE, SHORTDATE
HOUR	Indices will be binned for each hour.	TIME, TIMESTAMP	
MINUTE	Indices will be binned for each minute.	TIME, TIMESTAMP	
SECOND	Indices will be binned for each second.	TIME, TIMESTAMP	
MILLISECOND	Indices will be binned for each millisecond.	TIME, TIMESTAMP	TIME, TIMESTAMP
WEEK	Indices will be binned for each week.	DATE, TIMESTAMP	
QUARTER	Indices will be binned for each quarter.	DATE, TIMESTAMP	

The default value for `INDEX_GRANULARITY` is `DAY` for `DATE` and `SHORTDATE` and `MILLISECOND` for `TIME` and `TIMESTAMP`.

String Types

String columns have type `VARSTRING`. The type can have an optional length and an optional compression.

By specifying the compression `HASH64` you can define hashed string. Hashed columns store string data and use hashes to index the strings. This allows to save storage over string columns if the same strings occur frequently in the input data. The actual string data is held in maps that exist locally for each data partition (see section 5.1.3, page 34).

Attribute	Description	Default
(*length*)	number of characters/bytes	1024
COMPRESSION *values*	Compression (NONE, LZ4 for all types, and/or HASH64 for hashed strings. SPARSE or DICTIONARY for hashed strings, not in combination with LZ4)	NONE
MAPPING_TYPE *val*	Mapping type of the column (see section 24.2.6, page 294).	AUTO
MAPPING_FILE_GRANULARITY *val*	Number of combined string entries (see section 24.2.5, page 291)	1
PRELOAD_COLUMN *val*	NOTHING, COMPLETE, or MEMORY_EFFICIENT (see section 24.2.6, page 293)	NOTHING
SEPARATE BY *val*	columns for DSA optimization (see section 15.15.1, page 188)	NOTHING
REFERENCES *vals*	columns for DSJ optimization (see section 15.15.4, page 191)	NOTHING
INDEX *val*	Column index (NONE or EQUAL; see section 24.2.6, page 293)	NONE
PRELOAD_INDEX *val*	NOTHING or COMPLETE	NOTHING
CSV_COLUMN *int*|ETL	Column number within the CSV raw data. The numbering must start with 0, and in the whole table the numbers must be consecutive. If ETL is specified, the column data must be filled using an ETL clause (see section 10.6, page 104).	see below
SKIP *bool*	TRUE means to ignore the column when importing or querying data. This can be useful for CSV imports (see section 10.3.1, page 91). The column must not be an ETL column, then.	FALSE

MAPPING_FILE_GRANULARITY is another time/size trade-off: Cisco ParStream merges several string entries internally. The position for each of these merged entries is stored for faster access. This parameter describes the number of string entries that are combined as one position. A higher value results in a smaller index, a lower value results in faster access.

For example:

```
CREATE TABLE StringTable (
  str1 VARSTRING,
  str2 VARSTRING(100),
  str3 VARSTRING COMPRESSION LZ4, HASH64,
)
...
```

Note the following:

- The global option blobbuffersize (see section 13.2.1, page 124) defines the maximum number of bytes a string is allowed to have on import. The default value is 1,048,576 (1024*1024 or 2^{20}).

Blob Types

Blob columns have type BLOB. They store binary data and use hashes to index them. Currently they are equivalent to hashed strings.

The type can have an option length. Even if not specified they always have HASH64 compression. In addition, you can specify an additional LZ4 or SPARSE or DICTIONARY compression. As usual, you can set the index type and mapping options.

Attribute	Description	Default
(*length*)	number of characters/bytes	1024
`COMPRESSION` *values*	Compression (implicit `HASH64` and optional `LZ4` or `SPARSE`) or `DICTIONARY`)	`HASH64`
`MAPPING_TYPE` *val*	Mapping type of the column (see section 24.2.6, page 294).	`AUTO`
`PRELOAD_COLUMN` *val*	`NOTHING`, `COMPLETE`, or `MEMORY_EFFICIENT` (see section 24.2.6, page 293)	`NOTHING`
`SEPARATE BY` *val*	columns for DSA optimization (see section 15.15.1, page 188)	`NOTHING`
`REFERENCES` *vals*	columns for DSJ optimization (see section 15.15.4, page 191)	`NOTHING`
`INDEX` *val*	Column index (`NONE`, `EQUAL`; see section 24.2.6, page 293)	`NONE`
`PRELOAD_INDEX` *val*	`NOTHING` or `COMPLETE`	`NOTHING`
`CSV_COLUMN` *int*|`ETL`	Column number within the CSV raw data. The numbering must start with 0, and in the whole table the numbers must be consecutive. If `ETL` is specified, the column data must be filled using an ETL clause (see section 10.6, page 104).	see below
`SKIP` *bool*	`TRUE` means to ignore the column when importing or querying data. This can be useful for CSV imports (see section 10.3.1, page 91). The column must not be an ETL column, then.	`FALSE`

For example:

```
CREATE TABLE BlobTable (
  str1 BLOB,
  str2 BLOB COMPRESSION HASH64,
  str3 BLOB(512) COMPRESSION HASH64, SPARSE,
  str4 BLOB(512) COMPRESSION HASH64, DICTIONARY,
  str5 BLOB(512) COMPRESSION LZ4, HASH64,
)
...
```

Blob columns are constrained by the following:

- Blobs can only be used as `SINGLE_VALUE`
- The following configuration options can be chosen: In case of user provided hash values, the id can be omitted if blob data is inserted multiple times. Only the first insertion needs to be augmented with an id value. However, this only applies for non-parallel import runs and if the CSV file to be imported contains less than `partitionMaxRows` (see section 13.2.1, page 126) rows.

Note the following:

- The global option `blobbuffersize` (see section 13.2.1, page 124) defines the maximum number of bytes a blob is allowed to have on import. The default value is 1,048,576 (1024*1024 or 2^{20}).

Bitvectors

Attribute	Description	Default
COMPRESSION *values*	Compression (NONE, SPARSE, DICTIONARY, or LZ4)	NONE
PRELOAD_COLUMN *val*	NOTHING, COMPLETE, or MEMORY_EFFICIENT (see section 24.2.6, page 293)	NOTHING
SEPARATE BY *val*	columns for DSA optimization (see section 15.15.1, page 188)	NOTHING
REFERENCES *vals*	columns for DSJ optimization (see section 15.15.4, page 191)	NOTHING
INDEX *val*	Column index (NONE, EQUAL; see section 24.2.6, page 293)	NONE
INDEX_MASK *int*	Describes which bit of a bitvector to index (only used when valuetype is a bitvector).	
PRELOAD_INDEX *val*	NOTHING or COMPLETE	NOTHING
CSV_COLUMN *int*|ETL	Column number within the CSV raw data. The numbering must start with 0, and in the whole table the numbers must be consecutive. If ETL is specified, the column data must be filled using an ETL clause (see section 10.6, page 104).	see below
SKIP *bool*	TRUE means to ignore the column when importing or querying data. This can be useful for CSV imports (see section 10.3.1, page 91). The column must not be an ETL column, then.	FALSE

For example:

```
CREATE TABLE BitsTable (
  bitvector1 BITVECTOR8,
  bitvector2 BITVECTOR8 INDEX EQUAL INDEX_MASK 3,
)
...
```

Details of Other Column Attributes

Index Types

As introduced in section 5.3, page 35, Cisco ParStream provides the ability to specify bitmap indices for better performance. Currently equality encoding and range encoding is supported. Range encoded indices are typically larger than equality encoded indices.

Indextype	Description
NONE	No index will be created for the field.
EQUAL	The index is optimized for queries that check for value equality.
RANGE	The index is optimized for range queries.

Note:

- Only single-value numeric types and date/time types can have a RANGE index.
- Note that strings only support EQUAL indices based on hashes.

Preload Attributes

You can locally overwrite the server settings to preload some columns or indexes:

Option	Value	Description
`preloadcolumn`	`complete`	preload the column and the map-file (for strings and blobs)
	`memoryefficient`	preload the column, but when the column contains strings or blobs only the map-file is preloaded
	`nothing`	preload no column data
`preloadindex`	`complete`	preload the index
	`nothing`	preload no index

Mapping Attributes

Mapping Types

Attribute `MAPPING_TYPE` controls whether hash values are provided automatically or explicitly via the CSV import.

Value	Description
`AUTO`	The hash used for mapping values is generated internally (default).
`PROVIDED`	The hash used for mapping values is provided in the CSV input. The syntax for this is: "*hash*:*value*".

ALTER TABLE Statements

For schema changes, Cisco ParStream provides `ALTER TABLE` statements. Currently only an `ADD COLUMN` statement is supported.

See section 27.8.1, page 370 for a description of the detailed grammar of `ALTER TABLE` in BNF (Backus-Naur Form).

Dealing with ALTER TABLE Statements

As with `CREATE TABLE` statements, you have to send `ALTER TABLE` statements as command to **one** of the nodes in a cluster.

The statements result in a new metadata version as described in see section 5.2, page 34.

Imports are allowed to continue and finish. This means in details:

- Running CSV imports (`parstream-import`, see chapter 10, page 88) will continue until finished using the old metadata version. After a schema/metadata change got activated (which should usually happen almost immediately), the new CSV format has to be used. For this reason you might stop adding new CSV files and finish processing existing CSV files before performing a schema/metadata change.
- Java Streaming Imports (see chapter 19, page 216) will finish their commits. When starting a new commit, the metadata version is verified and results in an error if it doesn't match.
- `INSERT INTO` statements section 10.7, page 107. will finish their statements. New `INSERT INTO` statements will use the new metadata version.

Dealing with Multiple ALTER TABLE Statements

Note that schema/metadata modifications might be triggered by sending `ALTER TABLE` commands to different nodes at the same time. For this reason some rules and limitations exist:

- Cisco ParStream can only perform one modification at a time.
- While a schema modification is not finished, a new modification will be queued. Note that order of multiple queued changes is not guaranteed to be stable.
- If not all nodes of a cluster are online and a modification is still possible, it will be performed informing the missing node later when it becomes active again.

Note: You should **always wait for and check the result** of a schema modification, because due to the distributed organization of Cisco ParStream databases, it might happen that the call is not possible or does not succeed. When performing an `ALTER TABLE` command the answer signaling success is:

```
ALTER OK
```

ADD COLUMN Statements

ADD COLUMN statements allow to add new columns to existing tables. The principle syntax is as follows:

ALTER TABLE *tableName* ADD COLUMN *columnDefinition*

For the *columnDefinition* in principle you have to use the syntax as for CREATE TABLE statements (see section 24.2.4, page 284).

With ADD COLUMN the DEFAULT attribute of a column definition plays an important role. For all data located in existing partitions, imported before the column was added, this defines the default value used. For example:

```
ALTER TABLE Hotels ADD COLUMN HotelId UINT16 DEFAULT 42;
```

This means that queries sent to data imported before adding the column always yield 42 as HotelId. This default value is part of the schema so that the existing partitions are not modified. Note that the default value is only used for "old" partitions. Only "running" imports might still create partitions with the old format. When starting imports with the modified table, a corresponding value has to be provided.

If no default value is provided, the default for values of columns of old partitions before the column was added is NULL.

Note: You should **always wait for and check the result** of such a request, because due to the distributed organization of Cisco ParStream databases, it might happen that the call is not possible or does not succeed. When performing an ALTER TABLE command the answer signaling success is:

```
ALTER OK
```

Limitations of ADD COLUMN Statements

Note the following specific limitations for ADD COLUMN:

- DEFAULT values can only be SINGLE_VALUE literals. Currently, no expressions are supported.
- UNIQUE and PRIMARY KEY attributes are not allowed in ADD COLUMN statements.

Order of Table Modifications in Cluster Scenarios

In systems up-and-running you have to be careful to use the right order of updates for table modifications.

In cluster scenarios, this is handled by Cisco ParStream internally (therefore it is enough to send the update to only one node of the cluster).

DROP TABLE Statements

To remove tables not needed any more, the DROP TABLE statement is available.

See section 27.8.2, page 370 for a description of the detailed grammar of DROP TABLE in BNF (Backus-Naur Form).

NOTE:

- DROP TABLE will remove the whole table with all its current data without any further warning.
- Each client that is able to send queries, can send such a request.

Details of the DROP TABLE Statement

DROP TABLE can be used to delete tables that were created with a CREATE TABLE statement and might have been altered by ALTER TABLE statements.

For example, the following statement deletes the table Hotels:

```
DROP TABLE Hotels
```

On success the command returns with:

```
#OK DROP
```

This means that the whole table with all its data is logically deleted:

- The table is no longer listed in system tables such as ps_info_table (see section 26.3, page 309).
- The table name can be reused immediately to create a new table.

Queries, pending while a DROP TABLE command is performed, will finish without an error, but the values processed or returned are undefined. There will be no warning that the table is about to get dropped.

The physical deletion of all the partitions will happen asynchronously, when pending queries and running merges or imports are finished. This has the following effects:

- Imports on dropped tables will not add data to a new table with the same name in case it is created.

Note that table names are case-insensitive. Thus, the command above would also delete a table named hotels.

DROP TABLE responds with an error message if the table does not exist, is a system table, or if it cannot be deleted because of existing references.

The following additional rules and limitations apply to the usage of DROP TABLE:

- Tables that are referenced by other tables (see section 15.15.4, page 192), cannot be deleted with DROP TABLE.
- System tables can not be deleted with DROP TABLE.
- Scheduled merges for dropped tables are dropped, too.

- When a stopped server or cluster is restarted, any unfinished physical deletion proceeds.
- Importers drop pending partition distribution for dropped tables.
- Importers running in offline-import mode must be stopped before `DROP TABLE` and restarted after the drop.
- If a DBMS Scheduler job's action uses the dropped table, the job will be removed as well.

SQL Functions

Cisco ParStream supports a couple of following special functions, which are described here. Note that additional function (especially standard SQL function) are explained in Chapter 27, page 325.

CAST (*val* AS *type*)

- converts *val* into type *type*.
- *val* can be a value expression in a SELECT list, a WHERE clause, a ON clause, and a HAVING clause.
- The following conversions are possible:

to / from	signed integer	unsigned integer	floating-point	**VARSTRING**	**DATE** **SHORTDATE**	**TIME**	**TIMESTAMP**
signed integer	yes	yes	yes	-	-	-	-
unsigned integer	yes	yes	yes	-	-	-	-
floating-point	yes	yes	yes	-	-	-	-
VARSTRING	-	-	-	yes	yes	yes	yes
SHORTDATE	-	-	-	-	yes	-	yes
DATE	-	-	-	-	yes	-	yes
TIME	-	-	-	-	-	yes	-
TIMESTAMP	-	-	-	-	yes	yes	yes
NULL	yes	yes	yes	yes	yes	yes	yes
TRUE / **FALSE**	yes	yes	yes	-	-	-	-

  Thus, the following types count as *signed integer* target types:
  INT8, INT16, INT32, INT64, BITVECTOR8, SMALLINT, INTEGER
  the following types count as *unsigned integer* target types:
  UINT8, UINT16, UINT32, UINT64, USMALLINT, UINTEGER,
  and FLOAT and DOUBLE count as *floating-point* target types.

- For compatibility the **CAST** function also allows to specify the following target types, which are mapped to internally supported types as follows:

Alias	Internal type
VARCHAR(<length>)	**VARSTRING**
SMALLINT	**INT16**
USMALLINT	**UINT16**
INTEGER	**INT32**
UINTEGER	**UINT32**

  The length value of VARCHAR(<length>) will be ignored and mapped to VARSTRING, which has no internal length limitations.

- Note:
  - Integral conversions throw an exception, if the value is out of range.
  - If *val* is a NULL the function will return NULL for all valid target types. (The casts from NULL are provided, because for Cisco ParStream, NULL has type DOUBLE by default.)
  - The supported casts for TRUE yield 1, while the supported casts for FALSE yield 0.
  - When converting a DATE to TIME the time part is 00:00:00.000.

- When converting a `VARSTRING` value into a date/time value the string must be in the following format:
  * `DATE`/`SHORTDATE`: `'YYYY-MM-DD'` (e.g. `'2011-01-29'`)
  * `TIME`: `'HH:MI:SS[.MS]'` where milliseconds are optional (e.g. `'12:31:59.999'` or `'12:31:59'`)
  * `TIMESTAMP`: `'YYYY-MM-DD HH:MI:SS[.MS]'` where milliseconds are optional (e.g. `'2011-01-29 12:31:59.999'`)

- *`val`* must be a single value. Multi values will lead to an error.
- See section 27.3.4, page 350 for the corresponding grammar.

CONFIGVALUE (*key*)

- Returns the value of the passed configuration entry.
- The following configuration entries are supported as *key*:

`servername`	name of the server node
`nodename`	name of the node (in ETL imports the name of the importer)
`hostname`	name of the host
`port`	basic port (as string)

CURRENT_DATE ()

- Gives the query's starttime as DATE in UTC.

CURRENT_TIME ()

- Gives the query's starttime as TIME in UTC.

CURRENT_TIMESTAMP ()

- Gives the query's starttime as TIMESTAMP in UTC.

DATE_PART (*partname, col*)

- Extracts a specific part out of a date, time or timestamp value as integer.
- *partname* can be:
  - `DAY` - day of month
  - `DOW` - day of week, Sunday(0) - Saturday(6)
  - `DOY` - day of year, 1-365 (leap year 366)
  - `EPOCH` - UNIX timestamp, seconds since 1970-01-01 00:00:00
  - `HOUR`
  - `ISODOW` - day of week as defined in ISO 8601, Monday(1) - Sunday(7)
  - `ISOYEAR` - year that the date falls in, ISOYEAR of 2006-01-01 would be 2005
  - `MILLISECOND`
  - `MINUTE`
  - `MONTH`
  - `QUARTER` - quarter of year, 1-4
  - `SECOND`

  - `WEEK` - week of year, 1-53
  - `YEAR`
- *col* must be the name of a column/field with type DATE, SHORTDATE, TIME, or TIMESTAMP
- For example: For a column/field named *`val`* with value `2001-05-12 20:03:05.00`:
  the result of `DATE_PART('MONTH', val)` is `5` and
  the result of `DATE_PART('HOUR', val)` is `20`
- Note: This function is currently very slow, because *partname* can be different for every value of *col*. If the part field is the same for all values of the specific column it is recommended to use `EXTRACT` (see section 25, page 302 for details) or one of the provided unary extraction functions like `MONTH(column)`.

`DATE_TRUNC` `(`*`truncval, col`*`)`

- rounds the date or time value down according to a given trunc value
- *truncval* can be:
  - an integer, truncating the value according to the passed value in milliseconds
  - one of the following string values: `'YEAR'`, `'MONTH'`, `'DAY'`, `'HOUR'`, `'MINUTE'`, `'SECOND'`, `MILLISECOND`, `'WEEK'` or `'QUARTER'`
- *col* must be the name of a column/field with value type DATE, SHORTDATE, TIME, or TIMESTAMP
- For example: For a column/field named *`val`* with the value `2001-05-12 20:03:05.005` the result of `DATE_TRUNC('MONTH', val)` is `2001-05-01 00:00:00` and the result of `DATE_TRUNC(2000, timestamp)` is `2001-05-12 20:03:04`.

`DAYOFMONTH` `(`*`column`*`)`

- Extracts the day out of a date or timestamp value as integer.
- Note: This is the short writing of the `EXTRACT(DAY FROM column)`.

`DAYOFWEEK` `(`*`column`*`)`

- Extracts the day of the week out of a date or timestamp value as integer.
- Sunday results in 1 and Saturday results in 7.
- Note: `EXTRACT(DOW FROM column)` and `EXTRACT(ISODOW FROM column)` have different return values.

`DAYOFYEAR` `(`*`column`*`)`

- Extracts the day of year out of a date or timestamp value as integer.
- Range from 1 to 365 or 366 in leap years.
- Note: This is the short writing of the `EXTRACT(DOY FROM column)`

`DISTVALUES` `(`*`col`*`)`

- yields the distinct values of all values as one multivalue
- If the column itself is a multivalue column, it yields a multivalue of all distinct values these multivalues contain.
- The resulting multivalue does not contain NULL values (unless there are only `NULL` values).

- There is no defined order for the resulting elements. The order might change from call to call.
- This function is callable for all column types, although multivalue columns can only be defined for integer and floating-point values.
- For example:
  - If `SELECT col FROM MyTable` yields:

    ```
    #col
    0
    0
    0
    1
    2
    2
    0
    ```

    then `SELECT DISTVALUES(col) FROM MyTable` yields:

    ```
    #auto_alias_1___
    0,1,2
    ```

  - If `SELECT multivalue_col FROM MyTable` yields:

    ```
    #multivalue_col
    3,6,9
    6,7,8,9,10
    <NULL>
    0
    1
    1,2
    ```

    then `SELECT DISTVALUES(multivalue_col) FROM MyTable` yields:

    ```
    #auto_alias_1___
    0,1,2,3,6,7,8,9,10
    ```

`EPOCH` `(`*`column`*`)`

- Extracts the UNIX timestamp out of a date or timestamp value as integer.
- Returns the seconds counted since 1979-01-01 00:00:00.
- Note: This is the short writing of the `EXTRACT(EPOCH FROM column)`

`EXTRACT` `(`*`part FROM column`*`)`

- Extracts a specific part out of a date or time value as integer.
- *partname* can be:
  - `DAY` - day of month
  - `DOW` - day of week, Sunday(0) - Saturday(6)
  - `DOY` - day of year, 1-365 (leap year 366)

  - `EPOCH` - UNIX timestamp, seconds since 1970-01-01 00:00:00
  - `HOUR`
  - `ISODOW` - day of week as defined in ISO 8601, Monday(1) - Sunday(7)
  - `ISOYEAR` - ISO 8601 year that the date falls in, `ISOYEAR` of 2006-01-01 would be 2005
  - `MILLISECOND`
  - `MINUTE`
  - `MONTH`
  - `QUARTER` - quarter of year, 1-4
  - `SECOND`
  - `WEEK` - week of year, 1-53
  - `YEAR`
- *column* must be the name of a column/field or a single value with type `DATE`, `SHORTDATE`, `TIME`, or `TIMESTAMP`.
- Examples:
  `EXTRACT(MONTH FROM DATE'2001-05-12')` results in `5`.
  `EXTRACT(HOUR FROM TIMESTAMP'2001-05-12 20:03:05.00')` results in `20`.
  `EXTRACT(WEEK FROM SHORTDATE'2014-06-06')` results in `23`.

FIRST (*value*)

- Restricts a query to yield only “the first” of all possible values.
- Note that it is undefined which value/row is used if multiple values/rows match (even multiple calls of the same query can have different results).
- See section 27.3.4, page 349 for details and examples.

FLOOR (*val*)

- Rounds an floating-point value down to the next integral value.
- *val* can be numeric value.
- For example:
  - `FLOOR(4.7)` yields `4`
  - `FLOOR(-7.1)` yields `-8`

HASH64 (*strvalue*)

- Returns the hash value of the passed string *strvalue*
- For non-hashed strings, the value is the same as the `HASH64` compression (see section 24.2.4, page 285) yields for imported string values.
- This function is especially useful to distribute over string values. See section 6.3.1, page 59 for an example.

HOUR (*column*)

- Extracts the hour out of a time or timestamp value as integer.
- Note: This is the short writing of the `EXTRACT(HOUR FROM column)`.

IF (*value, trueresult, falseresult*)

- See section 27.3.4, page 354 for details and examples.

IFNULL (*value, replacement*)

- defines a replacement for NULL values.
- The type of the replacement must match the type of the value.
- For example:

```
SELECT IFNULL(StringVal,'no value') FROM table;
SELECT IFNULL(IntVal,-1) FROM table;
```

LOWER (*value*)

LOWERCASE (*value*)

- lowercases the string value *value*

MILLISECOND (*column*)

- Extracts the millisecond out of a time or timestamp value as integer.
- Note: This is the short writing of the `EXTRACT(MILLISECOND FROM column)`.

MINUTE (*column*)

- Extracts the minute out of a time or timestamp value as integer.
- Note: This is the short writing of the `EXTRACT(MINUTE FROM column)`.

MONTH (*column*)

- Extracts the month out of a date or timestamp value as integer.
- Note: This is the short writing of the `EXTRACT(MONTH FROM column)`.

NOW ()

- Gives the query's starttime as TIMESTAMP in time zone UTC.

QUARTER (*column*)

- Extracts the quarter out of a date or timestamp value as integer.
- Range: quarter 1 to 4.
- Note: This is the short writing of the `EXTRACT(QUARTER FROM column)`

SECOND (*column*)

- Extracts the second out of a time or timestamp value as integer.
- Note: This is the short writing of the `EXTRACT(SECOND FROM column)`.

TAKE (*columnname*)

- yields the value of column *columnname* that corresponds with the value of the previous `MAX()` or `MIN()` command
- See section 27.3.4, page 348 for details and examples.

`TRUNC` `(`*`val`*`)`

- truncates a floating-point value to an integral value
- *val* can be numeric value
- For example:
  - `TRUNC(4.7)` yields `4`
  - `TRUNC(-7.1)` yields `-7`

`UPPER` `(`*`value`*`)`

`UPPERCASE` `(`*`value`*`)`

- uppercases the string value *value*

`WEEK` `(`*`column`*`)`

- Extracts the week out of a date or timestamp value as integer.
- Range: From 1 to 53
- Note: This is the short writing of the `EXTRACT(WEEK FROM column)`

`YEAR` `(`*`column`*`)`

- Extracts the year out of a date or timestamp value as integer.
- Note: This is the short writing of the `EXTRACT(YEAR FROM column)`.

System Tables

Introduction of System Tables

The system tables - also known as "system catalog" or "information schema" - are a mechanism for requesting metadata. These tables are read-only views, which provide information about all of the tables, columns, and custom functions in the running Cisco ParStream instance. They are available through standard SQL commands

For example, you can query all tables and columns with the following command:

```
SELECT table_name,column_name,column_type,SQL_type,column_size
       FROM ps_info_column;
```

It might have the following output:

```
#table_name;column_name;column_type;sql_type;column_size
"Hotels";"City";"hashed";"VARSTRING";1024
"Hotels";"Hotel";"string";"VARSTRING";100
"Hotels";"Seaview";"numeric";"INT8";<NULL>
"Hotels";"Price";"numeric";"UINT16";<NULL>
"Hotels";"Num";"numeric";"UINT8";<NULL>
"Hotels";"ID";"numeric";"UINT32";<NULL>
"Bookings";"ID";"numeric";"UINT32";<NULL>
"Bookings";"Name";"string";"VARSTRING";1024
"Bookings";"Name";"numeric";"UINT32";<NULL>
```

In addition to the static database schema, system tables also serve as monitoring functions, such as RAM consumption or a list of all running queries.

For example:

```
SELECT pid, realtime_sec, total_ram_mb, operating_system
       FROM ps_info_process;
```

might have the following output:

```
#pid;realtime_sec;total_ram_mb;operating_system
17580;59;64257;"3.10.0-327.22.2.el7.x86_64 #1 SMP Thu Jun 23 17:05:11 UTC
   2016 x86_64"
```

Note the following general rules regarding Cisco ParStream system tables:

- The names of all Cisco ParStream system tables start with `ps_info`.
- The order of the columns is not guaranteed and might change. If the order matters, use explicit column names instead of a `*`.

List of all System Tables

Name	Meaning	Page
`ps_info_catalog`	Lists all system tables	307
Static information:		
`ps_info_version`	Yields version information	308
`ps_info_type`	Yields all existing data types	308
Schema and Configuration:		
`ps_info_table`	Yields information about all tables	309
`ps_info_column`	Yields information about all columns	309
`ps_info_compression`	Yields information about the compression attributes of all columns	310
`ps_info_bitmap_index`	Yields index information	311
`ps_info_partitioned_by`	Yields `PARTITION BY` clauses of tables	311
`ps_info_sorted_by`	Yields `ORDER BY` clauses of tables	311
`ps_info_separated_by`	Yields `SEPARATE BY` clauses of columns	311
`ps_info_configuration`	Yields the general configuration of a server	312
`ps_info_user_defined_option`	Yields all options defined via INI files or command line	315
Runtime information:		
`ps_info_user`	Yield the list of current database users	316
`ps_info_dynamic_columns_mapping`	Yields details of the current dynamic columns mapping	316
`ps_info_partition`	Yields details of all existing partitions	316
`ps_info_cluster_node`	Yields the details of all nodes in a cluster	317
`ps_info_remote_node`	Yields the details of all remote nodes of a cluster	317
`ps_info_partition_sync_backlog`	Yields open partition synchronizations	318
`ps_info_custom_query`	Yields names of registered custom queries	319
`ps_info_debug_level`	Yields settings of all debug levels	319
`ps_info_disc`	Yields file system information	319
`ps_info_job`	Yields all configured jobs	319
`ps_info_library`	Yields all loaded libraries	320
`ps_info_mapped_file`	Yields information about all memory mapped files	320
`ps_info_process`	Yields process ID, user name, and other process information of the server	320
`ps_info_running_query`	Yields all running queries	321
`ps_info_query_history`	Yields all recently done queries	321
`ps_info_import`	Yields all running and recently done imports	322
`ps_info_partition_distribution`	Yields the current partition distribution table	323
`ps_info_merge_queue_detail`	Yields all merges running on a cluster node	323
`ps_info_udf`	Yields all loaded user defined functions	324

Table `ps_info_catalog`

- Lists all system tables
- C++ Class: `SystemTablePsCatalog`
- Returned columns:

Column	Type	Meaning
`name`	`VARSTRING`	name of the system table
`description`	`VARSTRING`	description of the system table

The other tables are described in the sections below.

Static Tables

Table `ps_info_version`

- Yields Cisco ParStream version, source revision, source branch, build type and local changes
- C++ Class: `SystemTablePsVersion`
- Returned columns:

Column	Type	Meaning
`parstream_version`	`VARSTRING`	Official Cisco ParStream Version
`source_revision`	`VARSTRING`	internal source code revision ID
`source_branch`	`VARSTRING`	internal source branch name
`build_type`	`VARSTRING`	type of built (e.g. `"debug"`)
`build_host`	`VARSTRING`	host where the built was performed
`build_host_osversion`	`VARSTRING`	operating system version of the host where the built was performed
`build_datetime`	`VARSTRING`	timestamp when the built was performed (e.g. "20141210T075450Z")
`local_changes`	`VARSTRING`	list of changed source files

Table `ps_info_type`

- Yields all existing Cisco ParStream data types. If there is a corresponding Postgres data type, its oid is used.
- C++ Class: `SystemTablePsType`
- Returned columns:

Column	Type	Meaning
`oid`	`INT32`	type ID
`type_name`	`VARSTRING`	name of the type (in upper-case letters)
`type_size`	`INT16`	size of the type (`-1` for multivalue types)
`value_singularity`	`VARSTRING`	singularity (`SINGLE_VALUE`, `MULTI_VALUE`)

Schema and Configuration Tables

The following tables provide information about all tables and columns according to all column attributes described in section 24.2.4, page 284.

Table `ps_info_table`

- Yields which tables are in the database
- C++ Class: `SystemTablePsTable`
- Returned columns:

Column	Type	Meaning
`table_name`	`VARSTRING`	Name of the table
`import_directory_pattern`	`VARSTRING`	Value of `IMPORT_DIRECTORY_PATTERN`
`import_file_pattern`	`VARSTRING`	Value of `IMPORT_FILE_PATTERN`
`etl`	`VARSTRING`	ETL statement if any (or empty)
`distribution_algorithm`	`VARSTRING`	Algorithm used for partition distribution
`distribution_column`	`VARSTRING`	Column used for partition distribution
`distribution_redundancy`	`UINT32`	Redundancy value of partition distribution
`colocation_source`	`VARSTRING`	Referred table for co-located partition distribution
`metadata_version`	`VARSTRING`	Metadata version number of the last modification of this table

Table `ps_info_column`

- Yields table name, column name, and column attributes of all columns in the database
- C++ Class: `SystemTablePsColumn`
- Returned columns:

Column	Type	Meaning
table_name	VARSTRING	name of the table
column_name	VARSTRING	name of the column
column_type	VARSTRING	general type category of the column (string, numeric, datetime, ...)
value_type_oid	INT32	type ID (OID)
sql_type	VARSTRING	column type as in CREATE TABLE (INT32, VARSTRING, ...)
column_size	UINT64	number of characters (for strings and blobs)
mapping_level	VARSTRING	value of MAPPING_LEVEL (deprecated attribute)
mapping_type	VARSTRING	value of MAPPING_TYPE AUTO, PROVIDED
mapping_file_granularity	UINT32	value of MAPPING_FILE_GRANULARITY
singularity	VARSTRING	singularity (SINGLE_VALUE, MULTI_VALUE)
preload_column	VARSTRING	preload value (see section 15.9.3, page 177)
csv_column	VARSTRING	column in CSV imports (number or ETL)
csv_column_no	UINT32	column in CSV imports (NULL for ETL columns)
skip	VARSTRING	whether to skip this column on imports and queries (TRUE or FALSE, see section 10.3.1, page 91)
unique	VARSTRING	deprecated: whether this column is declared UNIQUE (TRUE or FALSE)
has_unique_constraint	UINT8	1 if this column is declared UNIQUE, 0 otherwise
not_null	VARSTRING	deprecated: whether this column is declared NOT NULL (TRUE or FALSE)
has_not_null_constraint	UINT8	1 if this column is declared NOT NULL, 0 otherwise
primary_key	VARSTRING	deprecated: whether this column is declared PRIMARY KEY (TRUE or FALSE)
is_primary_key	UINT8	1 if this column is declared PRIMARY KEY, 0 otherwise
default_value	VARSTRING	the default value for this column
dynamic_columns_type	VARSTRING	role for the dynamic columns feature (REGULAR_COLUMN, DYNAMIC_COLUMNS_KEY, or DYNAMIC_COLUMNS_VALUE; see section 7.2.2, page 69)

For the compression attributes of a column, see system table ps_info_compression (page 310).

Table ps_info_compression

- Yields the compression attributes of all columns in the database
- C++ Class: SystemTablePsCompression
- Returned columns:

Column	Type	Meaning
table_name	VARSTRING	name of the table
column_name	VARSTRING	Name of the column the compression belongs to
sequence_number	UINT32	The position of this compression in the list of compressions of this column
compression	VARSTRING	The kind of compression (NONE, HASH64, LZ4, SPARSE)
default_value	VARSTRING	SPARSE_DEFAULT value if compression is SPARSE, else empty
lower_bound	UINT8	Minimum bit-width in INDEX_BITS if compression is DICTIONARY, else empty
upper_bound	UINT8	Maximum bit-width in INDEX_BITS if compression is DICTIONARY, else empty

Table `ps_info_bitmap_index`

- Yields index attributes of all columns in the database
- C++ Class: `SystemTablePsBitmapIndex`
- Returned columns:

Column	Type	Meaning
`table_name`	`VARSTRING`	Name of the table the index belongs to
`column_name`	`VARSTRING`	Name of the column the index belongs to
`index_type`	`VARSTRING`	index type (`EQUAL` or `RANGE`)
`max_cached_values`	`UINT64`	value of `MAX_CACHED_VALUES`
`cache_nb_iterators`	`VARSTRING`	value of `CACHE_NB_ITERATORS`
`index_granularity`	`VARSTRING`	value of `MAX_CACHED_VALUES`
`index_bin_count`	`UINT64`	value of `INDEX_BIN_COUNT`
`index_bin_min`	`VARSTRING`	value of `INDEX_BIN_MIN`
`index_bin_max`	`VARSTRING`	value of `INDEX_BIN_MAX`
`index_bin_boundaries`	`VARSTRING`	value of `INDEX_BIN_BOUNDARIES`
`index_mask`	`UINT64`	value of `INDEX_MASK`

Table `ps_info_partitioned_by`

- Yields `PARTITIONED BY` clauses of tables
- C++ Class: `SystemTablePsPartitionedBy`
- Returned columns:

Column	Type	Meaning
`table_name`	`VARSTRING`	Name of the table
`column_name`	`VARSTRING`	Name of the column if the table is partitioned by a column name
`expression`	`VARSTRING`	Expression if the table is partitioned by an expression
`sequence_number`	`UINT32`	The position of this `PARTITION BY` clause in the list of all clauses

Table `ps_info_sorted_by`

- Yields `SORTED BY` clauses of tables
- C++ Class: `SystemTablePsSortedBy`
- Returned columns:

Column	Type	Meaning
`table_name`	`VARSTRING`	Name of the table
`column_name`	`VARSTRING`	Name of the column
`sequence_number`	`UINT32`	The position of this `ORDER BY` clause in the list of all clauses

Table `ps_info_separated_by`

- Yields `SEPARATED BY` clauses of columns
- C++ Class: `SystemTablePsSeparatedBy`
- Returned columns:

Column	Type	Meaning
`table_name`	`VARSTRING`	Name of the table
`column_name`	`VARSTRING`	Name of the column
`separating_column`	`VARSTRING`	
`sequence_number`	`UINT32`	The position of this `SEPARATE BY` clause in the list of all clauses

Table `ps_info_configuration`

- Yields configuration the requested Cisco ParStream server as a list of key/value pairs
- C++ Class: `SystemTablePsConfiguration`
- Returned columns:

Column	Type	Meaning
`key`	`VARSTRING`	Name of the configuration value
`value`	`VARSTRING`	Value of the configuration value as string

Currently, the following keys are supported:

Key	**Value**
`version`	Cisco ParStream version
`argv0`	Name of the started server process (`argv[0]`)
`cmdlineArgs`	Commandline arguments of the started server process (without `argv[0]`)
`startTime`	ISO 8601 representation of the server startup time
`locale`	Locale
`encoding`	Client encoding used
`isServer`	`"true"` if server, `"false"` if importer
`nodeName`	Name of the node
`hostName`	Name of the host
`port`	Basic port (as string)
`workingDir`	Working directory at startup
`confDir`	Directory where we read config files from
`journalDir`	Directory of journal files
`dataDir`	Partitions directory
`clusterId`	Cluster ID
`clusterRank`	Cluster rank
`mergeDisabled`	`"true"` if merges are disabled, `"false"` otherwise
`outputFormat`	Value of session settable option `outputFormat`, which is the format in which query results will sent to the client of this session (see section 13.2.1, page 121 and see section 27.10.1, page 373)
`queryHistoryMaxEntries`	Value of option `queryHistoryMaxEntries`, which is the maximum number performed queries are kept to see them in the query history (see section 13.2.1, page 129)
`queryHistoryMaxSeconds`	Value of option `queryHistoryMaxSeconds`, which is the maximum duration in seconds performed queries are kept to see them in the query history (see section 13.2.1, page 129)

Key	Value
importHistoryMaxEntries	Value of option importHistoryMaxEntries, which is the maximum number performed imports are kept to see them in the import history (see section 13.2.1, page 129)
importHistoryMaxSeconds	Value of deprecated option importHistoryMaxSeconds, which is the maximum duration in seconds performed imports are kept to see them in the import history
mappedFilesMax	Value of server option mappedFilesMax (see section 13.3.2, page 138)
mappedFilesCheckInterval	Value of server option mappedFilesCheckInterval (see section 13.3.2, page 138)
mappedFilesOutdatedInterval	Value of server option mappedFilesOutdatedInterval (see section 13.3.2, page 138)
mappedFilesAfterUnmapFactor	Value of server option mappedFilesAfterUnmapFactor (see section 13.3.2, page 138)
maxExecutionThreads	Value of option maxExecutionThreads which is the total number of threads available for task execution (see section 13.2.1, page 127)
maxQueryThreads	Value of option maxQueryThreads which is the maximum number of threads, out of the maxExecutionThreads available, which can be assigned to query tasks with 0 meaning that there is no limit (see section 13.2.1, page 127)
maxImportThreads	Value of option maxImportThreads which is the maximum number of threads, out of the maxExecutionThreads available, which can be assigned to import tasks with 0 meaning that there is no limit (see section 13.2.1, page 127)
maxMergeThreads	Value of option maxMergeThreads which is the maximum number of threads, out of the maxExecutionThreads available, which can be assigned to merge tasks with 0 meaning that there is no limit (see section 13.2.1, page 127)
maxExternalProcesses	Value of option maxExternalProcesses, which is the maximum number of concurrent external processes for execution of UDT statements (see section 13.2.1, page 127)
connection_id	Connection ID of the session that sent the request
queryPriority	Value of session settable option queryPriority, which is the priority query tasks issued in this session initially will run (see section 13.2.1, page 128 and see section 27.10.1, page 374)
importPriority	Value of session settable option importPriority, which is the priority import tasks issued in this session initially will run (see section 13.2.1, page 128 and see section 27.10.1, page 374)
mergePriority	Value of option mergePriority, which is the priority with which internal merge tasks run (see section 13.2.1, page 128)
queryThrottlingInterval	Value of option queryThrottlingInterval, which is the interval after which (periodically) a running query will be reduced in priority (see section 13.2.1, page 128)
limitQueryRuntime	Query runtime limit of the session that sent the request (see section 13.2.1, page 120)
numBufferedRows	Number of buffered rows for output (see section 13.2.1, page 120)

In addition, the values of all ExecTree options (see section 13.3.4, page 140) and the value of all optimization options (see section 13.5, page 149) are returned by the system table:

```
ExecTree.AsyncRemoteProxyExecution
ExecTree.BitmapAggregationLimit
... optimization.rewrite.all
optimization.rewrite.joinElimination
...
```

Note the following:

- Directories are absolute paths.
- `mergeDisabled` is the local status of the queried server. In a cluster, you have to ask the **leader** to see, whether currently merges in the cluster are disabled.
- The system table `ps_user_defined_option` (see page 315) lists all options as defined via INI and command line, which allows to find out, why a configuration entry does not have its default value.
- The order of the rows is undefined and might vary from call to call.

For example:

```
SELECT * FROM ps_info_configuration;
```

might have the following result:

```
#key;value
"version";"3.2.0"
"argv0";"parstream-server"
"cmdlineArgs";"first"
"startTime";"2014-07-23T19:00:05"
"locale";"C"
"encoding";"ASCII"
"isServer";"true"
"nodeName";"first"
"workingDir";"/parstream-testdata/2.1/HotelDemo"
"confDir";"/parstream-testdata/2.1/HotelDemo/conf"
"journalDir";"/parstream-testdata/2.1/HotelDemo/journals"
"dataDir";"parstream-testdata/2.1/HotelDemo/./partitions/"
"clusterId";""
"mergeDisabled";"false"
"outputFormat";"ASCII"
"queryHistoryMaxEntries";"1000"
"queryHistoryMaxSeconds";"600"
"importHistoryMaxEntries";"1000"
"importHistoryMaxSeconds";"600"
"maxExecutionThreads";"20"
"maxQueryThreads";"0"
"maxImportThreads";"5"
"maxMergeThreads";"5"
...
"connection_id";"first-190005-9077-0"
"queryPriority";"4"
"importPriority";"4"
"mergePriority";"4"
```

```
"queryThrottlingInterval";"0"
"limitQueryRuntime";"0"
"numBufferedRows";"32"
"ExecTree.BitmapAggregationLimit";"40000"
"ExecTree.GroupByBitmapLimit";"40000"
"ExecTree.IterativeGroupByAggregation";"false"
...
"optimization.rewrite.all";"individual"
"optimization.rewrite.joinElimination";"disabled"
...
```

Table `ps_info_user_defined_option`

- Yields all options defined via INI files or command line arguments.
- C++ Class: `SystemTablePsUserDefinedOption`
- Returned columns:

Column	Type	Meaning
`key`	`VARSTRING`	Name of the user defined option
`value`	`VARSTRING`	Value of the option as string
`source`	`VARSTRING`	Name of the INI file or “`<commandline>`” for command line arguments

Note the following:

- If an option is not set a INI file or as command line argument, the default value will be used and the option won’t be listed in the table.
- Thus, this system table allows to see what the user or system administrator has requested as non-default option value. System table `ps_info_configuration` (see page 312), provides the ability to query the resulting and current status of the running server. Note that options and the internal configuration might not match directly to each other.

For example:

```
SELECT * FROM ps_info_user_defined_option;
```

might have the following result:

```
#key;value;source
"import.first.sourcedir";"./import";"parstream.ini"
"merge";"false";"parstream.ini"
"reimportInterval";"1";"parstream.ini"
"server.first.datadir";"./partitions/";"parstream.ini"
"server.first.host";"127.0.0.1";"parstream.ini"
"server.first.port";"9042";"parstream.ini"
"servername";"first";"<commandline>"
"verbosity";"0";"parstream.ini"
```

Runtime Tables

Table `ps_info_user`

- Yield the list of current database users.
- C++ Class: `SystemTablePsUser`
- Returned columns:

Column	Type	Meaning
`login_name`	`VARSTRING`	name of the database user
`user_name`	`VARSTRING`	name of the associated PAM user

See section 9.2, page 78 for details.

Table `ps_info_dynamic_columns_mapping`

- Yields details of the current dynamic columns mapping.
- C++ Class: `SystemTablePsDynamicColumnsMapping`
- Returned columns:

Column	Type	Meaning
`table_name`	`VARSTRING`	name of the raw table
`dynamic_name`	`VARSTRING`	value of the dynamic column (column marked with `DYNAMIC_COLUMNS_KEY`)
`key_column`	`VARSTRING`	name of the `DYNAMIC_COLUMNS_KEY` column in the table
`value_column`	`VARSTRING`	name of the `DYNAMIC_COLUMNS_VALUE` column used by this dynamic name in the table
`is_valid`	`VARSTRING`	whether the dynamic name is a valid column name (`TRUE` or `FALSE`)
`is_conflicting`	`VARSTRING`	whether the dynamic name is conflicting with another dynamic name (`TRUE` or `FALSE`)

See section 7.2.2, page 69 for details.

Table `ps_info_partition`

- Yields details of all existing partitions
- C++ Class: `SystemTablePsPartition`
- Returned columns:

Column	Type	Meaning
`base_directory`	`VARSTRING`	root data directory (`datadir`)
`table_name`	`VARSTRING`	name of the table
`relative_path`	`VARSTRING`	path from root data directory
`num_records`	`INT64`	number of records
`partition_condition`	`VARSTRING`	condition for this partition e.g.: `Price < 100 AND City =` *`hashvalue`*
`status`	`VARSTRING`	status of this partition (see section 5.1.3, page 34)
`metadata_version`	`VARSTRING`	metadata version number of the table when this partition was created
`parstream_version`	`VARSTRING`	Cisco ParStream version used to write this partition
`access_time`	`TIMESTAMP`	time stamp of the last access

Table `ps_info_cluster_node`

- Yields the details of all nodes in a cluster from a node/server.
- C++ Class: `SystemTablePsClusterNode`
- Returned columns:

Column	Type	Meaning
`name`	`VARSTRING`	name of the node
`type`	`VARSTRING`	type of the node (`QUERY` or `IMPORT`)
`host`	`VARSTRING`	hostname
`port`	`UINT16`	basic (query) port
`leader`	`INT8`	1 if leader
`follower`	`INT8`	1 if follower
`active`	`INT8`	1 if active (only possible if online)
`online`	`INT8`	1 if online (connection channels open)
`follower_rank`	`UINT16`	rank (if no leader)
`parstream_version`	`VARSTRING`	version of the software the node uses (format: *ParStreamVersion*–*SourceCodeKey*, e.g.: `2.2.0-3fdf00a`)
`data_version`	`UINT64`	version number of the cluster node state (incremented with each update)
`pid`	`INT64`	the process identifier of the server process on the executing node
`node_status`	`VARSTRING`	status of the given node `active` (node is online and active, and can process queries) `online` (node is online and has to be synchronized; cannot process queries) `shutting_down` (node is shutting down) `offline` (node is offline and can neither import data nor process queries) The value might be followed by “`???`” (see below)
`merge_status`	`VARSTRING`	merge status of the node `enabled` (merging is enabled) `disabled` (merging is disabled)
`import_status`	`VARSTRING`	data import status of the node `enabled` (data import is enabled) `disabled` (data import is disabled)

If the node receiving this request is offline, the status returned is more or less the last known state of the cluster. To signal this, all `node_status` values of other nodes will have “`???`” at the end.

Note that the information about whether a node is active does not mean that requests could not have any problems. System table `ps_info_remote_note` (see section 26.4, page 317) lists details about (recent) errors when using connections to a specific node.

See section 6.2.1, page 41 for details about possible cluster node states.

See section 6.2.4, page 49 for details about how to use this system table.

Table `ps_info_remote_node`

- Yields the details of all remote nodes (if any) and their connection pool (see section 15.1, page 162).
- C++ Class: `SystemTablePsRemoteNode`
- Returned columns:

Column	Type	Meaning
name	VARSTRING	name of the node
host	VARSTRING	hostname
port	UINT16	basic (query) port
online	INT8	1 if online (connection channels open)
connection_available_count	UINT32	available connections in the connection pool
connection_check_count	UINT32	number of double-checks currently performed for available connections
connection_success_count	UINT64	sum of all established connection to the node
connection_error_count	UINT64	sum of all failed trials to connect to the node
num_connections_in_use	UINT64	number of connections currently in use by queries/inserts
connection_reuse_success_count	UINT64	number of connections which were fed back to the pool after use
connection_reuse_failure_count	UINT64	number of connections which could not be fed back to the pool after use
last_error_message	VARSTRING	last error message (or `NULL`/`""` if none)
last_error_time	TIMESTAMP	time stamp of the last error (or `NULL` if none)
parstream_version	VARSTRING	Cisco ParStream version of the node

You have to send this request to a node that is online.

Also note the following:

- Each node has its own set of counters. Restarting a node re-initializes the counters.
- The connection counts and `parstream_version` are 0 or empty until the information is available (this is usually after the first distributed query).
- Some features such as DHSGB (see section 15.15.2, page 189) may internally create additional connections. For this reason, distributed queries might increase some counters more than just 1.

Table `ps_info_partition_sync_backlog`

- Yields from a cluster leader (or follower) information about the partitions that still need to be synchronized to nodes of the cluster
- C++ Class: `SystemTablePsPartitionSyncBacklog`
- Returned columns:

Column	Type	Meaning
node_name	VARSTRING	name of the node
type	VARSTRING	type of the open synchronization (`merge`, `import`, or `delete`)
table_name	VARSTRING	name of the table to synchronize
relative_path	VARSTRING	relative path of partition to sync
node_name_source	VARSTRING	in case of an active sync the name of the node processing the sync, empty otherwise
sync_status	VARSTRING	sync status of the partition `pending` (no active sync ongoing) `active` (sync in progress)
metadata_version	UINT64	Metadata version number of the table it relates to

An empty result signals no open synchronizations. Note, however, that you have to send this request **to the leader** (or a follower) node. Otherwise, the result will be empty although open synchronizations exist or not up-to-date (if the follower doesn't have the newest state yet). See section 6.2.4, page 49 for details about how to use this table.

Table `ps_info_custom_query`

- Yields names of registered custom queries.
- C++ Class: `SystemTablePsCustomQuery`
- Returned columns:

Column	Type	Meaning
`registered_custom_query`	`VARSTRING`	

Table `ps_info_debug_level`

- Show settings of all debug levels (See section 9.5.4, page 86)
- C++ Class: `SystemTablePsDebugLevel`
- Returned columns:

Column	Type	Meaning
`class_name`	`VARSTRING`	name of the class to set the debug level for
`debug_level`	`INT64`	value of the debug level setting
`is_default`	`UINT8`	1 if the debug level of this class is tracking see section 13.2.1, page 120

Table `ps_info_disc`

- Prints disc usage of the partition folder
- C++ Class: `SystemTablePsDisc`
- Returned columns:

Column	Type	Meaning
`path`	`VARSTRING`	path of the partition
`file_name`	`VARSTRING`	name of the file in the partition
`type`	`VARSTRING`	type of the information in the file
`size_byte`	`INT64`	size of the file in Bytes
`status`	`VARSTRING`	status (`active` etc. (see section 5.1.3, page 34) or `"no partition.smd found"` or `"unknown"`)

Table `ps_info_job`

- Shows all configured jobs
- C++ Class: `SystemTablePsJob`
- Returned columns:

Column	Type	Meaning
`name`	`VARSTRING`	unique name of the job
`action`	`VARSTRING`	SQL command to be executed
`timing`	`VARSTRING`	Timing defining when the job is executed
`comment`	`VARSTRING`	A user comment about this job.
`enabled`	`VARSTRING`	Status of the job.

Table `ps_info_library`

- Shows string with paths/names of loaded libraries.
- C++ Class: `SystemTablePsLoadedLibrary`
- Returned columns:

Column	Type	Meaning
`loaded_library`	`VARSTRING`	

Table `ps_info_mapped_file`

- Yields path, size, role, and type of all memory mapped files of the Cisco ParStream process and the table/partition/column they belong to.
- C++ Class: `SystemTablePsMappedFile`
- Returned columns:

Column	Type	Meaning
`file_path`	`VARSTRING`	path of the file
`file_size`	`UINT64`	size of the file
`table`	`VARSTRING`	table the files belongs to
`column`	`VARSTRING`	column the files belongs to
`partition`	`VARSTRING`	partition the file belongs to
`type`	`VARSTRING`	type of the file (string with one of the following values: `column_data` (no filetype suffix) `column_data_map` (suffix `.map`) `index` (suffix `.sbi`) `hashed_string_lookup_data` (suffix `.hs`) `hashed_string_lookup_map` (suffix `.hsm`)
`access_time`	`TIMESTAMP`	last time the mapped file was accessed by the system

Table `ps_info_process`

- Yields process ID, user name, and other process information of the server.
- C++ Class: `SystemTablePsProcess`
- Returned columns:

Column	Type	Meaning
`pid`	`INT32`	process ID
`user`	`VARSTRING`	user name
`host_name`	`VARSTRING`	host name
`operating_system`	`VARSTRING`	operating system
`threads`	`INT32`	current number of threads
`realtime_sec`	`DOUBLE`	real time used so far
`utime_sec`	`DOUBLE`	user time used so far
`stime_sec`	`DOUBLE`	system time used so far
`vsize_mb`	`DOUBLE`	size of virtual memory usage of this process
`used_ram_mb`	`DOUBLE`	size of resident memory usage of this process
`total_ram_mb`	`DOUBLE`	size of physical memory of the machine
`free_ram_mb`	`DOUBLE`	size of free memory of the machine

Table ps_info_running_query

- Yields all running queries
- C++ Class: `SystemTablePsRunningQuery`
- Returned columns:

Column	Type	Meaning
connection_id	VARSTRING	connection ID
query_id	VARSTRING	query ID
execution_id	VARSTRING	execution ID (relates to the execution_ID column of the `INSPECT THREADPOOL` output see section 27.6, page 362)
execution_type	VARSTRING	type of query ("QUERY", "IMPORT", or "MERGE")
command	VARSTRING	command string
starttime	TIMESTAMP	starting time of query
runtime_msec	UINT64	running time so far in milliseconds
realtime_msec	UINT64	amount of time spent in this query accumulated across all threads assigned to it.
cputime_msec	UINT64	amount of CPU time (i.e. not counting I/O wait cycles etc.) spent executing this query accumulated across all threads assigned to it.
min_num_threads	UINT64	minimum number of threads requested by execution of this query before other tasks issued after it shall be considered (>0).
max_num_threads	UINT	maximum number of threads that is allowed to be assigned to execution of this query (0 means no limit).
current_num_threads	UINT64	number of threads currently assigned to this task.
execution_priority	UINT64	priority used for executing the query (see section 15.1, page 158)
channel	VARSTRING	used command channel (POSTGRESQL or NETCAT)
role	VARSTRING	MASTER or SLAVE
master_node	VARSTRING	node the query was originally sent to
slave_nodes	VARSTRING	list of slave nodes if any (only set if role is MASTER)
slave_sub_id	VARSTRING	sub-ID of the slave node (only set if role is SLAVE)

Note the following:

- The command string may have SQL or JSON format.
- Any request from a client is a `MASTER` request. If this results into sub-requests sent to executing servers, they are listed in `slave_nodes`. `master_node` is part of the list of `slave_nodes` if a part of the distributed execution happens there.
- Any sub-request from a server is a `SLAVE` request. In that case the command string always has JSON format.

Table ps_info_query_history

- Yields (recent) done queries.
- C++ Class: `SystemTablePsQueryHistory`
- Returned columns:

Column	Type	Meaning
connection_id	VARSTRING	connection ID
query_id	VARSTRING	query ID
execution_id	VARSTRING	execution ID (relates to the execution_ID column of the INSPECT THREADPOOL output see section 27.6, page 362)
execution_type	VARSTRING	type of query ("QUERY", "IMPORT", or "MERGE")
command	VARSTRING	command string
starttime	TIMESTAMP	starting time of query
runtime_msec	UINT64	running time in milliseconds
realtime_msec	UINT64	amount of time spent in this query accumulated across all threads assigned to it.
cputime_msec	UINT64	amount of CPU time (i.e. not counting I/O wait cycles etc.) spent executing this query accumulated across all threads assigned to it.
execution_priority	UINT64	priority used for executing the query (see section 15.1, page 158)
channel	VARSTRING	used command channel (POSTGRESQL or NETCAT)
role	VARSTRING	MASTER or SLAVE
master_node	VARSTRING	nodes the query was originally sent to
slave_nodes	VARSTRING	list of slave nodes if any (only set if role is MASTER)
slave_sub_id	VARSTRING	sub-ID of the slave node (only set if role is SLAVE)
result_size	UINT64	number of rows of query result
error	VARSTRING	error message if query failed

Note the following:

- With the global options queryHistoryMaxSeconds and queryHistoryMaxEntries you can control how long and how many queries are kept to be returned by this request. See section 13.2.1, page 129 for details.
- The command string may have SQL or JSON format.
- Any request from a client is a MASTER request. If this results into sub-requests sent to executing servers, they are listed in slave_nodes. master_node is part of the list of slave_nodes if a part of the distributed execution happened there.
- Any sub-request from a server is a SLAVE request. In that case the command string always has JSON format.

For example after sending a request to srv3 a query might return:

```
#query_id;command;starttime;runtime_msec;channel;role;master_node;slave_nodes;slave_sub_id;result_size;error
"srv3-160409-9130-2-4";"select * from Hotels";2014-04-02
    16:09:58.954;68;"NETCAT";"MASTER";"srv3";"srv1, srv2, srv3, srv4, srv5";"";26;""
```

As you can see, the command was forwarded to 5 servers (including itself). The corresponding query history at srv5 might look as follows:

```
#query_id;command;starttime;runtime_msec;channel;role;master_node;slave_nodes;slave_sub_id;result_size;error
"srv3-160409-9130-2-4";"select {...}";2014-04-02
    16:09:58.968;41;"NETCAT";"SLAVE";"srv3";"";"1";10;""
```

The dots represent the internal JSON command from master to slave node.

Table `ps_info_import`

- Yields running and recently done imports. In a cluster, this request has to be send to a leader or follower node. The entries are listed as soon as the imports have been accepted as valid commands to import data. They might still collect the data to import (import state `STARTED`) or might have all data but not finished the activation of the resulting partitions (import state `INSERTED`). Successful imports have state `ACTIVATED`; failed/canceled imports have state `CANCELED`. **Note:** This system table is currently in **beta state** so that with upcoming versions columns and the exact definition about which entries are listed might change.
- C++ Class: `SystemTablePsImport`
- Returned columns:

Column	Type	Meaning
`connection_id`	`VARSTRING`	connection ID
`query_id`	`VARSTRING`	query ID
`execution_id`	`VARSTRING`	execution ID (relates to the execution_ID column of the `INSPECT THREADPOOL` output see section 27.6, page 362)
`table_name`	`VARSTRING`	name of the table the insert applies to
`import_state`	`TIMESTAMP`	`STARTED`, `INSERTED`, `ACTIVATED`, or `CANCELED`
`starttime`	`TIMESTAMP`	starting time of query
`connected_host`	`VARSTRING`	host that caused the insert
`connected_port`	`VARSTRING`	port of the host that caused the insert
`connected_user`	`VARSTRING`	user that caused the insert if known
`master_node`	`VARSTRING`	nodes the insert was originally sent to

Note the following:

- You can use the global option TimportHistoryMaxEntries to control how many of the *historical* entries are kept to be returned by this request (see section 13.2.1, page 129 for details).

Table `ps_info_partition_distribution`

- Shows the current partition distribution table (see section 6.3, page 53) denormalized to have for each value of a column in a table one row for each node
- C++ Class: `SystemTablePsPartitionDistribution`
- Returned columns:

Column	Type	Meaning
`table_name`	`VARSTRING`	name of the table
`column_name`	`VARSTRING`	name of the column
`column_type`	`VARSTRING`	type of the column
`distribution_value`	`VARSTRING`	value for which the distribution exists
`server_name`	`VARSTRING`	one of the nodes for the distribution value
`distribution_rank`	`UINT32`	priority of the node (smaller is higher)

Table `ps_info_merge_queue_detail`

- Shows detailed information on the merge job queue for the current node

- C++ Class: `SystemTablePsMergeQueueDetail`
- Returned columns:

Column	Type	Meaning
`node_name`	`VARSTRING`	name of the node the queue belongs to
`distribution_group`	`VARSTRING`	distribution value of the merge as string
`table_name`	`VARSTRING`	target table
`merge_level`	`VARSTRING`	merge level (H | D | W | F)
`relative_path_target`	`VARSTRING`	relative path of target partition
`queue_position`	`INT64`	position of job in the merge queue
`relative_path_source`	`VARSTRING`	relative path of source partition

To identify which source partitions belong to one merge job the `queue_position` can be used in addition to `relative_path_target`.

Table `ps_info_udf`

- Shows detailed information about the loaded user defined functions xUDTOs (see section 20, page 232) and user defined procedures (UDPs). Note that these UDFs are provided by the user or application programmer and Cisco ParStream is not responsible for its functionality.
- C++ Class: `SystemTableUDF`
- Returned columns:

Column	Type	Meaning
`created_at`	`TIMESTAMP`	Empty/NULL.
`type`	`VARSTRING`	Type of user defined function
`name`	`VARSTRING`	Name of the routine that can be used in a SQL request
`internal_name`	`VARSTRING`	Empty/NULL for xUDTOs/UDPs.
`version`	`VARSTRING`	Empty/NULL for xUDTOs/UDPs.
`parameters`	`VARSTRING`	. Empty/NULL for xUDTOs.
`definition`	`VARSTRING`	. Commands for UDPs, empty/NULL for xUDTOs.
`file_name`	`VARSTRING`	Name of the file that contains this function (xUDTO: script file)
`external_tool`	`VARSTRING`	Fox xUDTOs: Type of the script file.

- Types that are supported as function parameter(s) are: *UINT8, UINT16, UINT32, UINT64, INT8, INT16, INT32, INT64, FLOAT, DOUBLE, SHORTDATE, DATE, TIMESTAMP, TIME, VARSTRING*.

SQL Grammar

BNF Notation

The syntactic notation used in ISO/IEC 9075 is an extended version of the BNF (Backus Normal Form or Backus Naur Form). In the version of BNF used in ISO/IEC 9075, the notation symbols have the meaning as follows:

- <...> is used to name syntactic elements
- `::=` is used to break a composed statement of the SQL language into sub-parts
- `|` means "or"
- `[...]` means "optional"

For example, just below "preparable SQL data statement" is the name of a syntactic element, and it is written here as <preparable SQL data statement>.

SQL Statements

According to Standard SQL, 20.6 <prepare statement>: we can divide statements for execution into data statements (such as `SELECT`, `INSERT`, and `DELETE`) schema statements (such as `CREATE TABLE`), session statements (such as `SET`), and system statements (currently only `ALTER SYSTEM`):

```
<preparable statement> ::=
   <preparable SQL data statement>
   | <preparable SQL schema statement>
   | <preparable SQL control statement>
   | <preparable SQL session statement>
   | <preparable SQL system statement>
   | <preparable SQL user administration statement>

<preparable SQL data statement> ::=
   <dynamic select statement>
   | <insert statement>
   | <delete statement>
   | <inspect statement>

<preparable SQL schema statement> ::=
   <SQL schema statement>

<SQL schema statement> ::=
   <SQL schema definition statement>
   <SQL schema manipulation statement>

<preparable SQL control statement> ::=
   <SQL control statement>

<preparable SQL session statement> ::=
   <SQL session statement>

<preparable SQL system statement> ::=
   <SQL system statement>

<preparable SQL user administration statement> ::=
   <SQL user administration statement>
```

For details about `<dynamic select statement>` (`SELECT` statements), see section 27.3, page 327.
For details about `<insert statement>` (`INSERT` statements), see section 27.4, page 360.
For details about `<delete statement>` (`DELETE` statements), see section 27.5, page 361.
For details about `<inspect statement>` (`INSPECT` statements), see section 27.6, page 362.
For details about `<SQL schema definition statement>` (`CREATE TABLE` statements), see section 27.7, page 364.
For details about `<SQL schema manipulation statement>` (`ALTER TABLE` and `DROP TABLE` statements), see section 27.8, page 370.
For details about `<SQL session statement>` (`SET` statements), see section 27.10, page 373.
For details about `<SQL system statement>` (`ALTER SYSTEM` statements), see section 27.11, page 375. For details about `<SQL user administration statement>` (`CREATE USER` statements), see section 27.12, page 377.

SELECT Statements

In principle, SELECT statements have the following syntax (from Standard SQL, 20.6 <prepare statement>: Prepare a statement for execution):

```
<dynamic select statement> ::=
   "SELECT"
   [ <set quantifier> ]
   <select list>
   <from clause>
   [ <where clause> ]
   [ <group by clause> ]
   [ <having clause> ]
   [ <order by clause> ]
   [ <limit clause> ]
```

In more detail with some clauses expanded:

```
<dynamic select statement> ::=
   "SELECT"
   [ "DISTINCT" | "ALL" ]
   { "*" | <select sublist> [ { "," <select sublist>}... ] }
   "FROM" <table reference>
   [ "WHERE" <search condition> ]
   [ "GROUP" "BY" <column name> [ { "," <column name> }... ] ]
   [ "HAVING" <search condition> ]
   [ "ORDER" "BY" <column reference> [ "ASC" | "DESC" ]
             [ { "," <column reference> [ "ASC" | "DESC" ] }... ] ]
   [ "LIMIT" { [ <offset> "," ] <row_count> | <row_count> "OFFSET" <offset> } ]
```

Note the following:

- Unless explicitly requested, **the order of columns and rows is undefined and might vary from query to query**. For this reason:
  - Use specific column names instead of the wildcard * to get a specific column order.
  - Use a GROUP BY clause (see section 27.3.1, page 331) to get a a specific row order.

The following subsections explain this syntax in detail.

SELECT Statement and UNION Clause

From Standard SQL, 20.6 <prepare statement>: Prepare a statement for execution.

```
<dynamic select statement> ::=
   <cursor specification>
```

From Standard SQL, 14.3 <cursor specification>: Define a result set.

```
<cursor specification> ::=
   <query expression>
```

Combined syntax:

```
<dynamic select statement> ::=
   <query expression>
```

From Standard SQL, 7.13 <query expression>: Specify a table.

```
<query expression> ::=
   <query expression body> [ <order by clause> ] [ <limit clause> ]

<query expression body> ::=
   <query term>
   | <query expression body> "UNION" [ "ALL" | "DISTINCT" ] <query term>

<query term> ::=
   <query specification>
   | "(" <query expression body>
      [ <order by clause> ] [ <limit clause> ] ")"
```

Please note that the names and types of the returned fields of each query of a UNION statement have to be identical in Cisco ParStream. This can be accomplished by using CAST (see section 27.3.4, page 350) and aliasing.

Query Specifications and Alias Handling

From Standard SQL, 7.12 <query specification>: Specify a table derived from the result of a table expression and from Standard SQL, 7.4 <table expression>: Specify a table or a grouped table.

```
<query specification> ::=
   "SELECT" [ <set quantifier> ] <select list> <table expression>

<select list> ::=
   "*"
   | <select sublist> [ { "," <select sublist> }... ]

<select sublist> ::=
   <derived column>

<derived column> ::=
   <value expression> [ <as clause> ]

<as clause> ::=
   "AS" <column name>

<table expression> ::=
   <from clause>
   [ <where clause> ]
   [ <group by clause> ]
   [ <having clause> ]
```

- Each `<select sublist>` indicates a value that you want to display.
- For <value expression> see section 27.3.3, page 344.

- For <column name> see section 27.3.2, page 333.
- For <column reference> see section 27.3.2, page 342.
- For <set quantifier> see section 27.3.4, page 347.

Note the following:

- Again, note that the order of columns and rows is undefined and might vary from query to query (see above).
- Note that an alias name specified in the <as clause> is only known outside the clause where it is defined. For this reason, instead of

  ```
  SELECT id AS s FROM Addr1 JOIN Addr2 ON s = id;
  ```

  you have to write:

  ```
  SELECT Addr1.id AS s FROM Addr1 JOIN Addr2 ON Addr1.id = Addr2.id;
  ```

  Using the alias column name in the <group by clause> is possible.

FROM Clause

From Standard SQL, 7.5 <from clause>: Specify a table derived from one table.

```
<from clause> ::=
   "FROM" <table reference>
```

From Standard SQL, 7.6 <table reference>: Reference a table.

```
<table reference> ::=
   <table primary>
   | <joined table>

<table primary> ::=
   <table name> [ [ "AS" ] <correlation name> ]
   | "(" <dynamic select statement> ")" [ "AS" ] <correlation name>
   | <table operator> [ [ "AS" ] <correlation name> ]
   | <custom_procedure_call>
```

Note:

- For <table name> see section 27.3.2, page 333.
- For <table operator> see section 27.3.1, page 333.
- For <correlation name> see section 27.3.2, page 333.
- For <dynamic select statement> see section 27.3, page 327.

A <custom_procedure_call> is a call to a predefined stored query. It consists of the procedure name and arguments in parentheses. The <custom_procedure_call> within the <table primary> is a de facto standard enhancement to Standard SQL. Currently this is only used for ETL Imports with CSVFETCH (see section 10.6, page 104)and PARTITIONFETCH see section 14.2.1, page 155.

JOIN Clause

From Standard SQL, 7.7 <joined table>: Specify a table derived from a Cartesian product, inner join, or outer join.

```
<joined table> ::=
   <cross join>
   | <qualified join>

<cross join> ::=
   <table reference> "CROSS" "JOIN" <table reference>

<qualified join> ::=
   <table reference> [ <join type> ] "JOIN" <table reference> <join condition>

<join type> ::=
   "INNER"
   | <outer join type> [ "OUTER" ]

<outer join type> ::=
   "LEFT"
   | "RIGHT"
   | "FULL"

<join condition> ::=
   "ON" <search condition>
```

For details about `<search condition>` see section 27.3.2, page 337.

For example:

```
SELECT A.*, B.* FROM A
       INNER JOIN B ON A.userid = B.userid;
SELECT lastname, departmentName FROM employee
       LEFT OUTER JOIN department ON employee.departmentID = department.id;
SELECT City.name, Customers.customer FROM Customers
       RIGHT OUTER JOIN City ON Customers.cityId = City.id;
SELECT City.name, Customers.customer FROM Customers
       FULL OUTER JOIN City ON Customers.cityId = City.id;
SELECT lastname, departmentName FROM employee
       CROSS JOIN department;
```

Note:

- JOINs on multivalues (see section 23.8, page 275) are not supported and result in an exception.
- There a restrictions on alias column names (see section 27.3.1, page 328).

See section 15.7, page 168 for a discussion of how to optimize JOIN applications when using Cisco ParStream.

WHERE Clause

From Standard SQL, 7.8 <where clause>: Specify a table derived by the application of a <search condition> to the result of the preceding <from clause>.

```
<where clause> ::=
   "WHERE" <search condition>
```

For details about `<search condition>` see section 27.3.2, page 337..

For example:

```
SELECT * FROM mytable
         WHERE mykey > 10;
```

GROUP BY Clause

From Standard SQL, 7.9 <group by clause>: Specify a grouped table derived from the application of the <group by clause> to the result of the previously specified clause.

```
<group by clause> ::=
   "GROUP" "BY" <column name> [ { "," <column name>}... ]
```

- For <column name> see section 27.3.2, page 333.
- A <column name> of the <group by clause> must either correspond to a <column reference> of the table resulting from the <from clause> or be a <column name> of an <as clause> in a <select sublist>.
- A <column reference> within the <value expression> of a <select sublist>

```
<select sublist> ::= <derived column> ::= <value expression> [ <as clause> ]
                                               |
             +--------------------------+------+------+---------------+
             |                          |             |               |
<boolean value expression> <numeric value expression> |               |
             |                          | <string value expression>   |
        <predicate>                     |             | <datetime value expression>
             |                          |             |               |
e.g.: <column reference> = 10           +-------------+---------------+
                                                      |
      +--------------+------------------------+-------+-------------+
      |              |                        |                     |
<case expression><if expression><set function specification><column reference>
               |  |
e.g.:          |  IF <column reference> = 3, 'yes','no'
e.g.:          CASE  <column reference> WHEN 3 THEN 'three'
```

  has to be additionally within the <value expression> of a <general set function> or a <column reference> of the <group by clause>.
- For <column reference> see section 27.3.2, page 342.

HAVING Clause

From Standard SQL, 7.10 <having clause>: Specify a grouped table derived by the elimination of groups that do not satisfy a <search condition>.

```
<having clause> ::=
 "HAVING" <search condition>
```

For details about `<search condition>` see section 27.3.2, page 337.

ORDER BY Clause

The `ORDER BY` clause has the following syntax to specify a sort order (see Standard SQL, 10.10 <sort specification list>):

```
<order by clause> ::=
   "ORDER" "BY" <column reference> [ "ASC" | "DESC" ]
       [ { "," <column reference> [ "ASC" | "DESC" ] }... ]
```

Note:

- Without `ORDER BY` the order of the results is undefined and might even change from call to call.
- `ASC` is the default sort order.

For example:

```
SELECT myfield1, myfield2
       FROM mytable
       ORDER BY myfield2 DESC, myfield1;
```

LIMIT Clause

`LIMIT` is a supported language opportunity for Standard SQL to limit the number of rows in a result:

```
<limit clause> ::=
   "LIMIT" { [ <offset> "," ] <row_count> | <row_count> [ "OFFSET" <offset> ] }

<offset> ::=
   <unsigned integer>

<row_count> ::=
   <unsigned integer>
```

Note:

- The <offset> argument specifies the offset of the first row to return, and the <row_count> specifies the maximum number of rows to return. The offset of the first row is 0 (not 1).
- The <limit clause> is a Language Opportunity of Standard SQL. See [SQL/Foundation, 9075-2, Working Draft, 2003], Editor's Notes for WG3:HBA-003 = H2-2003-305, Possible Problems within

SQL/Foundation, Language Opportunities, 5WD-02-Foundation-2003-09.pdf, pp. Notes-25 ff., +931+, p, Notes-59.

- Without using the GROUP BY clause, the contents of the limited rows is still undefined (even from call to call).
- Currently the usage of LIMIT inside IN subqueries without ORDER BY leads to wrong results with multiple cluster nodes or partitions.

For example:

```
SELECT a, b FROM mytable LIMIT 10;

SELECT myfield1, myfield2
       FROM mytable
       WHERE mykey>10
       ORDER BY myfield2 DESC
       LIMIT 10 OFFSET 3;
```

According to Standard SQL, 7.13 <query expression>, you could also specify a table as follows:

```
<fetch first clause> ::=
 "FETCH" "FIRST" [ <unsigned integer> ] { "ROW" | "ROWS" } "ONLY"
```

However, this is not realized in Cisco ParStream. Instead, in Cisco ParStream you have two options:

- Use the FIRST function (see section 27.3.4, page 349).
- Use the <limit clause> as described above.

DYNAMIC_COLUMNS Operator

```
<table operator> ::=
  <dynamic columns operator>
  | <external user defined operator>

<dynamic columns operator> ::=
  "DYNAMIC_COLUMNS" "("
    "ON" <table name>
    [ "PARTITION BY" <column list> ]
    "JOIN_COLUMNS" "(" <column list> ")"
    [ "DYNAMIC_COLUMN_NAMES" "(" <select sublist> ")" ]
    [ "STATIC_COLUMN_NAMES" "(" <select sublist> ")" ]
  ")"
```

See section 7.2.2, page 70 for details.

Lexical Elements, Scalar Expressions, and Predicates

Names and Identifiers

From Standard SQL, 5.4 Names and identifiers: Specify name.

```
<table name> ::=
   <regular identifier>

<column name> ::=
   <regular column identifier>

<correlation name> ::=
   <regular identifier>
```

- For <regular identifier> see section 27.3.2, page 334.
- For <regular column identifier> see section 27.3.2, page 334.

Note:

- Cisco ParStream's <column name> definition is a subset of Standard SQL with at least two characters (see section 24.2.1, page 282).
- Furthermore in Cisco ParStream, a column can be defined as <regular column identifier> outside of SQL language, only, so Cisco ParStream has introduced <regular column identifier>.

Token and Separator

<regular column identifier> is used in <column name> (see section 27.3.2, page 333). <regular identifier> is used in <correlation name> and <table name> (see section 27.3.2, page 333).

From Standard SQL, 5.2 <token> and <separator>:

Specify lexical units (tokens and separators) that participate in SQL language.

```
<regular identifier> ::=
   <identifier body>

<identifier body> ::=
   <identifier start> [ <identifier part>... ]

<identifier part> ::=
   <identifier start>
   | <identifier extend>

<regular column identifier> ::=
   <column identifier body>

<column identifier body> ::=
   <identifier start> <identifier part>...
```

Note:

- <regular column identifier> is a subset of Standard SQL - SQL:2003 - <regular identifier> definition.
- <column identifier body> is a subset of Standard SQL - SQL:2003 - <identifier body> definition.

```
<identifier start> ::=
   <simple Latin letter>
   !! See the syntax Rules.
```

```
<identifier extend> ::=
   "_"
   | <digit>
   !! See the Syntax Rules.
```

- <identifier start> is a subset of Standard SQL - SQL:2003 - definition, corresponding SQL:1992/SQL-92.
- <identifier extend> is a subset of Standard SQL - SQL:2003 - definition, corresponding SQL:1992/SQL-92.

```
<delimiter token> ::=
   <character string literal>
   | <date string>
   | <time string>
   | <timestamp string>
   | <SQL special character>
   | <not equals operator>
   | <greater than or equals operator>
   | <less than or equals operator>
```

- For <character string literal>, see section 27.3.5, page 355.
- For <date string>, <time string>, <timestamp string>, see section 27.3.5, page 355.
- Note that Cisco ParStream also supports a `shortdate` string (see section 23.4, page 268).

As a Cisco ParStream proprietary extension of Standard SQL, Cisco ParStream defines:

```
   | <match insensitive operator>
   | <not match sensitive operator>
   | <not match insensitive operator>
```

```
<not equals operator> ::=
   "<>" | "!="

<greater than or equals operator> ::=
   ">="

<less than or equals operator> ::=
   "<="
```

The ability also to use operator `!=` as <not equals operator> is an extension by Cisco ParStream.

As a Cisco ParStream proprietary extension of Standard SQL, Cisco ParStream defines:

```
<match insensitive operator> ::=
  <match sensitive operator><asterisk>

<not match sensitive operator> ::=
  <not operator><match sensitive operator>

<not match insensitive operator> ::=
  <not operator><match insensitive operator>
```

- <match sensitive operator> As a Cisco ParStream proprietary extension of Standard SQL, see section 27.3.2, page 336.
- <not operator> As a Cisco ParStream proprietary extension of Standard SQL, see section 27.3.2, page 336.

Note:

- <identifier body> is **not** case sensitive.

SQL Terminal Character

From Standard SQL, 5.1 <SQL terminal character>:

Define the terminal symbols of the SQL language and the elements of strings.

```
<simple Latin letter> ::=
   <simple Latin upper case letter>
   | <simple Latin lower case letter>

<simple Latin upper case letter> ::=
   "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N"
    | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

<simple Latin lower case letter> ::=
   "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
    | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<digit> ::=
   "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<SQL special character> ::=
   <space>
   | <quote>
   | <double quote>
   | "("
   | ")"
   | ","
   | "."
   | "+"
   | "-"
   | "*"
   | "/"
   | ":"
   | <equals operator>
   | <less than operator>
   | <greater than operator>
   | ">"
   | ">="
   | "_"

<space> ::= " "
```

```
<asterisk> ::= "*"

<equals operator> ::= "="

<less than operator> ::= "<"

<greater than operator> ::= ">"

<quote> ::= "'"

<double quote> ::= """
```

As a Cisco ParStream proprietary extension of Standard SQL, 5.1 <SQL terminal character>, Cisco ParStream defines:

```
   | <match sensitive operator>
   | <not operator>

<match sensitive operator> ::= "~"

<not operator> ::= "!"
```

Search Condition

A <search condition> is used in a <where clause> and <having clause> (see section 27.3, page 327).

From Standard SQL, 8.20 <search condition>: Specify a condition that is TRUE, FALSE, or unknown, depending on the value of a <boolean value expression>.

```
<search condition> ::=
   <boolean value expression>
```

From Standard SQL, 6.34 <boolean value expression>: Specify a boolean value.

```
<boolean value expression> ::=
   <boolean term>
   | <boolean value expression> "OR" <boolean term>

<boolean term> ::=
   <boolean factor>
   | <boolean term> AND <boolean factor>

<boolean factor> ::=
   [ "NOT" ] <boolean test>

<boolean test> ::=
   <boolean primary> [ "IS" [ "NOT" ] <truth value> ]

<truth value> ::=
   "TRUE"
   | "FALSE"

<boolean primary> ::=
   <predicate>
   | <boolean predicand>

<boolean predicand> ::=
   <parenthesized boolean value expression>
   | <nonparenthesized value expression primary>

<parenthesized boolean value expression> ::=
   "(" <boolean value expression> ")"
```

The <search condition> specifies a condition that has to be True for each row to be selected. For allowed operators see the description of predicates below.

The truth value `UNKNOWN` is not supported.

The following table clarifies the behavior of `NOT` in combination with `NULL`:

a	`NOT` a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

The following table clarifies the behavior of `AND` and `OR` in combination with `NULL`:

a	b	a `AND` b	a `OR` b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE
TRUE	NULL	NULL	TRUE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

The operations `AND` and `OR` work with a *short circuit* optimization. This means if the operand *a* of the `AND` operation is `FALSE`, the calculation of the operand *b* will be skipped, because we already know that the result is `FALSE`. In case of the `OR` operation the operand *b* calculation will be skipped, if the operand *a* results in `TRUE`.

Predicates

<predicate> is used in <boolean term> in section 27.3.2, page 337.

From Standard SQL, 8.1 <predicate>: Specify a condition that can be evaluated to give a boolean value.

```
<predicate> ::=
  <comparison predicate>
| <in predicate>
| <like predicate>
| <POSIX regular expression>
```

- <POSIX regular expression> is not part of Standard SQL, see section 27.3.2, page 342.

<comparison predicate>

From Standard SQL, 8.2 <comparison predicate>: Specify a comparison of two row values.

```
<comparison predicate> ::=
   <row value predicand> <comparison predicate part 2>

<comparison predicate part 2> ::=
   <comp op> <row value predicand>

<comp op> ::=
   <equals operator>
   | <not equals operator>
   | <less than operator>
   | <greater than operator>
   | <less than or equals operator>
   | <greater than or equals operator>
```

- For <row value predicand>, see <case expression> in section 27.3.4, page 350.
- For <equals operator>, see section 27.3.2, page 336.
- For <not equals operator>, see section 27.3.2, page 334.
- For <less than operator>, see section 27.3.2, page 336.
- For <greater than operator>, see section 27.3.2, page 336.
- For <less than or equals operator>, see section 27.3.2, page 334.
- For <greater than or equals operator>, see section 27.3.2, page 334.

<in predicate> == IN Subquery Expression

From Standard SQL, 8.4 <in predicate>: Specify a quantified comparison.

```
<in predicate> ::=
   { <column reference> | <column name> }
   "IN"
   { <table subquery> | ( <in value list> ) }
```

IN <table subquery> == Subquery Expression

From Standard SQL, 8.4 <in predicate>: Specify a quantified comparison.

```
   { <column reference> | <column name> }
   "IN"
   <table subquery>
```

From 7.15 <subquery>:

Specify a scalar value, a row, or a table derived from a <query expression>.

```
<table subquery> ::=
   <subquery>
```

```
<subquery> ::=
   "(" <query expression> ")"
```

Or, combined:

```
   { <column reference> | <column name> }
   "IN"
   "(" <query expression> ")"
```

- For <column reference> see section 27.3.2, page 342.
- For <column name> see section 27.3.2, page 333.
- For <query expression> see section 27.3, page 327.
- The <in predicate>_with IN <table subquery> has to be *one* <column reference> or <column name> (of an <as clause>, see section 27.3, page 327) followed by keyword IN and a parenthesized <query expression>. The <column reference> or <column name> is evaluated and compared to each row of the <subquery> result.
- When the query is distributed across multiple servers as in a cluster then either the outer query must be executed only locally (i.e. by involving only EVERYWHERE distributed tables, see section 6.3.1, page 58) or subquery used by the IN clause must be co-located to the outer query. This can be achieved via COLOCATION (see section 6.3.1, page 59) or by using EVERYWHERE distributed tables (see section 6.3.1, page 58).

For example:

```
SELECT a FROM t
         WHERE b IN (SELECT x FROM y WHERE z=1);
```

Selects column `a` of all rows of table `t`, where values of column `b` are in the result set yielded by the statement `SELECT x FROM y WHERE z=1`.

IN (<in value list>) == Row and array comparisons

From Standard SQL, 8.4 <in predicate>: Specify a quantified comparison.

```
   { <column reference> | <column name> }
   "IN"
   "(" <in value list> ")"

<in value list> ::=
   <unsigned literal> [ { "," <unsigned literal> }... ]
```

Or, combined:

```
   { <column reference> | <column name> }
   "IN"
   "(" <unsigned literal> [ { "," <unsigned literal> }... ] ")"
```

- For <column reference> see section 27.3.2, page 342.
- For <column name> see section 27.3.2, page 333.
- For <unsigned literal> see section 27.3.3, page 344.
- The _<in predicate>_ with IN <value list> has to be one <column reference> or <column name> (of an <as clause>, see section 27.3, page 327) followed by keyword `IN` and a parenthesized <in value list>, a list of <unsigned literal>s. The <column reference> or <column name> is evaluated and compared to each <unsigned literal> of the <in value list>. This is the shorthand notation for

```
    { <column reference> | <column name> } = <unsigned literal 1>
    "OR"
    { <column reference> | <column name> } = <unsigned literal 2>
    "OR"
    ...
```

For example:

```
SELECT a FROM t WHERE b IN (1,2,3,5);
```

Selects column `a` of all rows of table `t`, where values of column `b` are either 1, 2, 3, or 5.

<like predicate> == LIKE operator

From Standard SQL, 8.5 <like predicate>: Specify a pattern-match comparison.

```
<like predicate> ::=
   { <character string literal> | <column reference> }
   "LIKE"
   { <character string literal> | <column reference> }
```

- For <character string literal>, see section 27.3.5, page 355.
- For <column reference>, see section 27.3.2, page 342.

The LIKE operator allows string comparison with pattern matching. It supports 2 wildcards:

- _ matches any character
- `%` matches any sequence of characters (zero or more)

Note:

- The pattern is always matched against the entire string, thus substring matching requires % wildcards in the pattern.
- Matching characters are case-sensitive.

Examples:

```
-- match:
'ABCDEFG' LIKE 'ABC%'
'ABCDEFG' LIKE '%EFG'
'ABCDEFG' LIKE 'A_C_E_G'
'ABCDEFG' LIKE '%B%'
```

```
'ABCDEFG' LIKE '%G%'

-- don't mach:
'ABCDEFG' LIKE 'E'
'ABCDEFG' LIKE 'G_'
'ABCDEFG' LIKE 'A_'
'ABCDEFG' LIKE '%A'
'ABCDEFG' LIKE '%b%'
'ABCDEFG' LIKE 'aBCDEFG'
```

<POSIX regular expression>

A <POSIX regular expression> is not part of Standard SQL.

```
<POSIX regular expression> ::=
   { <character string literal> | <column reference> } <POSIX regular expression part
       2>

<POSIX regular expression part 2> ::=
   <POSIX match operator> { <character string literal> | <column reference> }

<POSIX match operator> ::=
   <match sensitive operator>
   | <match insensitive operator>
   | <not match sensitive operator>
   | <not match insensitive operator>
```

- For <character string literal>, see section 27.3.5, page 355.
- For <column reference>, see section 27.3.2, page 342.
- For <match sensitive operator>, see section 27.3.2, page 336.
- For <match insensitive operator>, <not match sensitive operator>, and <not match insensitive operator>, see section 27.3.2, page 334.

Column Reference

In general:

```
<column reference> ::=
  <identifier>
```

In details (from Standard SQL, 6.7 <column reference> Reference a column):

```
<column reference> ::=
  <basic identifier chain>
```

From Standard SQL, 6.6 <identifier chain>
Disambiguate a <period>-separated chain of identifiers.

```
<identifier chain> ::=
  <identifier>

<basic identifier chain> ::=
  <identifier chain>
```

- For <identifier>, see section 27.3.2, page 333.

Usage: <column reference> is used

- in <nonparenthesized value expression primary> (see section 27.3.3, page 344)
- in <group by clause> and <order by clause> (see section 27.3, page 327)

Row Value Predicand

<row value predicand> is used in <comparison predicate>, <comparison predicate part 2>, <in predicate>, and <like predicate> (see section 27.3.2, page 338), as well as in <POSIX regular expression> and <row value predicand> (see section 27.3.2, page 342), as well as in <case operand> and <when operand> in <case expression> and <row value predicand> in <overlaps predicate part 2> in <case expression> (see section 27.3.4, page 350).

In general:

```
<row value predicand> ::=
   <nonparenthesized value expression primary>
   | <common value expression>
```

- For <nonparenthesized value expression primary> and <common value expression>, see section 27.3.3, page 344.

In details (from Standard SQL, 7.2 <row value expression>: Specify a row value):

```
<row value predicand> ::=
   <row value special case>
   | <row value constructor predicand>

<row value special case> ::=
   <nonparenthesized value expression primary>
```

From Standard SQL, 7.1 <row value constructor>: Specify a value or list of values to be constructed into a row.

```
<row value constructor predicand> ::=
   <common value expression>
   | <boolean predicand>
```

From Standard SQL, 6.34 <boolean value expression>: Specify a boolean value.

```
<boolean predicand> ::=
   <nonparenthesized value expression primary>
```

Value Expression

A <value expression> can have different formats:

```
<value expression> ::=
   <common value expression>
   | <boolean value expression>

<common value expression> ::=
   <numeric value expression>
   | <string value expression>
   | <datetime value expression>
```

A <value expression> might be used used as follows:

- <value expression> is used in <case abbreviation> and <result expression> (see section 27.3.4, page 350), in <derived column> (see section 27.3, page 327), in <aggregate function> and <first function> (see section 27.3.4, page 347).
- <common value expression> and <nonparenthesized value expression primary> (a special form of <numeric value expression>) are used in <row value predicand> (see section 27.3.2, page 343).
- <string value expression> and <numeric value expression> are used in <if expression> (see section 27.3.4, page 354).

Numeric Value Expression

From Standard SQL, 6.26 <numeric value expression> Specify a numeric value:

```
<numeric value expression> ::=
   <term>
   | <numeric value expression> "+" <term>
   | <numeric value expression> "-" <term>

<term> ::=
   <factor>
   | <term> "*" <factor>
   | <term> "/" <factor>

<factor> ::=
   [ <sign> ] <numeric primary>

<numeric primary> ::=
   <value expression primary>
   | <numeric value function>
```

- For <numeric value function>, see section 27.3.4, page 346.

String Value Expression

From Standard SQL, 6.28 <string value expression> Specify a character string value or a binary string value:

```
<string value expression> ::=
   <character value expression>

<character value expression> ::=
   <character factor>

<character factor> ::=
   <character primary>

<character primary> ::=
   <value expression primary>
```

Date/Time Value Expression

From Standard SQL, 6.30 <datetime value expression> Specify a datetime value.

```
<datetime value expression> ::=
   <datetime term>
   | [ <unsigned integer> <sign> ] CURRENT\_DATE "(" ")" [ <sign> <unsigned integer> ]
   | [ <unsigned integer> <sign> ] CURRENT\_TIME "(" ")" [ <sign> <unsigned integer> ]
   | [ <unsigned integer> <sign> ] CURRENT\_TIMESTAMP "(" ")" [ <sign> <unsigned
      integer> ]
   | [ <unsigned integer> <sign> ] NOW "(" ")" [ <sign> <unsigned integer> ]

<datetime term> ::=
   <datetime factor>

<datetime factor> ::=
   <datetime primary>

<datetime primary> ::=
   <value expression primary>
```

Value Expression Primary

From Standard SQL,6.3 <value expression primary> Specify a value that is syntactically self-delimited:

```
<value expression primary> ::=
   <parenthesized value expression>
   | <nonparenthesized value expression primary>

<parenthesized value expression> ::=
   "(" <value expression> ")"

<nonparenthesized value expression primary> ::=
   <unsigned value specification>
   | <column reference>
   | <set function specification>
   | <case expression>
   | <if expression>
```

From Standard SQL, 6.4 <value specification> and <target specification> Specify one or more values, host parameters, SQL parameters, dynamic parameters, or host variables:

```
<unsigned value specification> ::=
   <unsigned literal>
```

Numeric Value Functions

From Standard SQL, 6.27 <numeric value function>: Specify a value derived by the application of a function to an argument.

```
<numeric value function> ::=
   <modulus expression>
   | <extract expression>
   | ...

<modulus expression> ::=
   "MOD" "(" <numeric value expression> <comma> <numeric value expression> ")"
   | <numeric value expression> "MOD" <numeric value expression> ")"

<extract expression> ::=
   "EXTRACT" "(" <extract field> "FROM" <extract source> ")"
<extract field> ::=
   <primary datetime field>
<primary datetime field> ::=
   <non-second primary datetime field>
   | "SECOND"
   | <parstream datetime field>
<non-second primary datetime field> ::=
   "YEAR"
   | "MONTH"
   | "DAY"
   | "HOUR"
   | "MINUTE"
<parstream datetime field> ::=
   "DOW"
   | "DOY"
   | "EPOCH"
   | "ISODOW"
   | "ISOYEAR"
   | "MILLISECOND"
   | "QUARTER"
   | "WEEK"
<extract source> ::=
   <datetime value expression>
<datetime value expression> ::=
   <datetime term>
<datetime term> ::=
  <datetime factor>
<datetime factor> ::=
  <datetime primary>
<datetime primary> ::=
   <value expression primary>
```

Note for MOD in Cisco ParStream:

- Cisco ParStream supports both forms of MOD:

  ```
  val1 MOD val2
  MOD(val1,val2)
  ```

- The first operand of a MOD operation (the dividend) can be a integer, floating-point or date value, or NULL.
- The second operand of a MOD operation (the divisor) can be a integer or floating-point value or NULL.

Aggregates and FIRST

From Standard SQL, 6.9 <set function specification>: Specify a value derived by the application of a function to an argument.

```
<set function specification> ::=
   <aggregate function>
   | <first function>
```

<first function> is a proprietary extension of Standard SQL by Cisco ParStream, see section 27.3.4, page 349.

Usage: <set function specification> is used

- in <nonparenthesized value expression primary> (see section 27.3.3, page 344).

Aggregate Functions

From Standard SQL, 10.9 <aggregate function>: Specify a value computed from a collection of rows.

```
<aggregate function> ::=
   "COUNT" "(" "*" ")" [...]
   | <general set function>
   | "MEDIAN" "(" <column name> ")"
   | ("PERCENTILE_DISC" | "PERCENTILE_CONT") "(" <column name> "," <double> ")"

<general set function> ::=
   <set function type> "(" [ <set quantifier> ] <value expression> ")"

<set function type> ::=
   <computational operation>

<computational operation> ::=
   "AVG" | "COUNT" | "MAX" | "MIN" | "STDDEV_POP" | "SUM"

<set quantifier> ::=
   "DISTINCT"
```

The following aggregation functions are supported:

Name	Description
`AVG()`	Calculates the average
`COUNT()`	Counts the number
`MAX()`	Calculates the maximum
`MEDIAN()`	Calculates the median
`MIN()`	Calculates the minimum
`PERCENTILE_CONT()`	Calculates the percentile after doing linear interpolation
`PERCENTILE_DISC()`	Calculates the discrete percentile from the set of input values
`STDDEV_POP()`	Calculates the population standard deviation
`SUM()`	Calculates the sum over

- In Cisco ParStream, the <set function specification> is realized within the <value expression> (see section 27.3.3, page 344) of a <select sublist> (see section 27.3, page 327), only, and not within the <search condition> (see section 27.3.2, page 337).
- In Cisco ParStream, the `DISTINCT` set quantifier is only supported for the `COUNT()` aggregate function.
- In Cisco ParStream, the `ALL` set quantifier is not supported.

Return Types:

Function	Description
`AVG(`*expr*`)`	average of all values as `DOUBLE`
`COUNT(*)`	number of rows as `INT64`
`MAX(`*expr*`)`	maximum of all values as `INT64` or `DOUBLE`
`MEDIAN()`	median as `INT64` or`INT64` or `DOUBLE`
`MIN(`*expr*`)`	minimum of all values as `INT64` or `DOUBLE`
`PERCENTILE_CONT()`	percentile as `DOUBLE`
`PERCENTILE_DISC()`	percentile as `INT64` or `DOUBLE`
`STDDEV_POP(`*expr*`)`	population standard deviation of all values as `INT64` or `DOUBLE`
`SUM(`*expr*`)`	sum of all values as `INT64` or `DOUBLE`

See `http://www.postgresql.org/docs/9.1/interactive/functions.html`

TAKE Clause

If you want to combine aggregated columns with specific values of other columns, you can use the `TAKE` clause, which is a Cisco-ParStream-specific extension. It allows to fill in one of the possible values of rows that match with the aggregated value of the previous `MIN()` or `MAX()` command.

For example:

```
SELECT MIN(Price), TAKE(Hotel), TAKE(City) FROM Hotels
```

This statement yields for one of the hotels with minimum `Price` the corresponding values in columns `Hotel` and `City`.

If multiple matching aggregated values exist, it is undefined which corresponding values are used. Using `TAKE()` without a previous `MIN()` or `MAX()` is an error. However, other aggregates or columns may come in between. You can even have multiple `TAKE`'s after different `MIN` or `MAX`. For example:

```
SELECT MIN(price) AS MinPrice,
       TAKE(comment) as MinComment,
       MAX(updated) AS MaxUpdated,
       hotel,
       duration,
       TAKE(price) AS LatestPrice,
       AVG(price),
       TAKE(comment) AS LatestComment
    FROM MyTable
    GROUP BY hotel, duration
    ORDER BY MaxUpdated ASC
```

is a valid statement (provided the columns exist) and

- yields as `MinComment` the value of column `comment` that corresponds with the minimum price (`MIN(price)`) and
- yields as `LatestPrice` the value of column `price` that corresponds with the latest (maximum) update (`MAX(updated)`) and
- yields as `LatestComment` the value of column `comment` that corresponds with the latest (maximum) update (`MAX(updated)`)

FIRST Function

<first function> is a proprietary extension of Standard SQL by Cisco ParStream: It allows to restrict queries only to "the first" value computed from an collection of rows. The effect is like selecting one arbitrary resulting rows for the corresponding query if the value is requested without using `FIRST`. However, because only one result is requested, the query might run faster.

Format:

```
<first function> ::=
   "FIRST" ( <value expression> )
```

- For <value expression>, see section 27.3.3, page 344.

Note that if the order of resulting values/rows is undefined (which always is the case without using `ORDER BY`), it is undefined, which "first" value/row is used. Even multiple calls of the same query on the same database may have different results. Only if the result is ordered with `ORDER BY` so that only one unique "first" result exists, this expression will always yield the same value.

`FIRST()` returns NULL, if no result exists.

For example:

```
SELECT FIRST(name)
       FROM MyTable
       WHERE val IS NULL
```

yields the name of *one of the rows* where ‘val’ is NULL.

For example:

```
SELECT FIRST(firstname), FIRST(lastname)
       FROM People
       WHERE income > 60000
```

yields first and last name of *one of the rows* where the income is greater than 60000.

For example:

```
SELECT FIRST(Firstname), FIRST(Lastname)
       FROM People
       ORDERED BY income DESC
```

yields first and last name of the person *or one of the persons* (!) with the highest income.

CAST Specification

A <cast specification> allows to explicitly convert a value of one type to another:

```
<cast specification> ::=
   "CAST" ( <cast operand> "AS" <data type> )

<cast operand> ::=
   <value expression>
   | "NULL"
   | "TRUE"
   | "FALSE"
```

Casts can be used in:

- <select list>
- <search condition> of `WHERE` clauses, `ON` clauses, and `HAVING` clauses.

See section 25, page 299 for details about which conversions are supported.

CASE and COALESCE Expressions

The <case expression> (CASE and COALESCE clause) allows to map values to other values or to find the first non-NULL value among different columns.

In general, it has the following format:

```
<case expression> ::=
   "COALESCE" ( <value expression> { "," <value expression> }... )
   | "CASE" <row value predicand> <simple when clause>... [ "ELSE" <result> ] "END"
   | "CASE" <searched when clause>... [ "ELSE" <result> ] "END"

<simple when clause> ::=
   "WHEN" <row value predicand> "THEN" <result>

<searched when clause> ::=
   "WHEN" <search condition> "THEN" <result>

<result> ::=
   <value expression>
```

Usage: <case expression> is used

- in <nonparenthesized value expression primary> (see section 27.3.3, page 344).

In detail (from Standard SQL, 6.11 <case expression> Specify a conditional value):

```
<case expression> ::=
   <case abbreviation>
   | <case specification>

<case abbreviation> ::=
   "COALESCE" "(" <value expression> { "," <value expression>}... ")"

<case specification> ::=
   <simple case>
   | <searched case>

<simple case> ::=
   "CASE" <case operand> <simple when clause> ... [ <else clause> ] "END"

<searched case> ::=
   "CASE" <searched when clause>... [ <else clause> ] "END"

<simple when clause> ::=
   "WHEN" <when operand> "THEN" <result>

<searched when clause> ::=
   "WHEN" <search condition> "THEN" <result>

<else clause> ::=
   "ELSE" <result>

<case operand> ::=
   <row value predicand>
   | <overlaps predicate part 1>

<when operand> ::=
   <row value predicand>

<result> ::=
   <result expression>

<result expression> ::=
   <value expression>
```

From Standard SQL, 8.14 <overlaps predicate>:

Specify a test for an overlap between two datetime periods.

```
<overlaps predicate part 1> ::=
   <row value predicand>
```

- For <value expression> see section 27.3.3, page 344
- For <search condition> see section 27.3.2, page 337.
- For <row value predicand> see section 27.3.2, page 343.

For example:

```
SELECT COALESCE(city,id,housenr,postcode,street,'all columns are NULL') AS
    FirstNotNull,
        city,id,housenr,postcode,street
        FROM Address;
```

might have the following output:

```
   FirstNotNull    |        city        | id | housenr | postcode |        street
-------------------+--------------------+----+---------+----------+---------------------
 Southhood         | Southhood          |  4 |       4 |    54321 | Building Street
 Southhood         | Southhood          |  3 |       3 |    54321 | Building Street
 Sea Coast Village | Sea Coast Village  | 15 |     123 |    77666 | Noidea Place
 Seven Hills       | Seven Hills        |  6 |      13 |    13432 | House Path
 Seven Hills       | Seven Hills        |  7 |      25 |    13432 | House Path
 Capital City      | Capital City       | 12 |       1 |    86843 | Foobar Road
 Capital City      | Capital City       | 14 |       1 |    86843 | Barfoo Street
 Northtown         | Northtown          |  1 |       2 |    12345 | Skypscraper Street
 Northtown         | Northtown          |  2 |       3 |    12345 | Skypscraper Street
 Desert Valley     | Desert Valley      | 11 |      12 |    45323 | Living Street
 Southhood         | Southhood          |  5 |       7 |    54321 | Building Street
 Seven Hills       | Seven Hills        |  8 |      29 |    13432 | House Path
 Capital City      | Capital City       | 13 |       2 |    86843 | Foobar Road
 Desert Valley     | Desert Valley      | 10 |       9 |    45323 | Living Street
 Desert Valley     | Desert Valley      |  9 |       6 |    45323 | Living Street
```

For example:

```
SELECT CASE id*id WHEN   1 THEN 'one'
                  WHEN 144 THEN 'eleven?'
                  WHEN   4 THEN 'two'
                  WHEN 144 THEN 'twelve'
                  ELSE          'different'
       END,
       id
       FROM Address;
```

might have the following output:

```
 auto_alias_1__ | id
----------------+----
 different      |  4
 different      |  5
 different       |  3
 different       |  8
 different       | 15
 eleven?         | 12
 different       | 14
 different       | 13
 two             |  2
 different       |  7
 different       |  9
 different       |  6
 one             |  1
 different       | 11
 different       | 10
```

For example:

```
SELECT CASE id WHEN housenr   THEN 'id = housenr'
               WHEN housenr+1 THEN 'id = housenr+1'
               WHEN housenr-1 THEN 'id = housenr-1'
               ELSE                '|id-housenr|>1'
       END,
       id, housenr
       FROM Address;
```

might have the following output:

```
 auto_alias_1__ | id | housenr
----------------+----+---------
 id = housenr   |  4 |       4
 |id-housenr|>1 |  5 |       7
 |id-housenr|>1 |  8 |      29
 id = housenr   |  3 |       3
 |id-housenr|>1 |  6 |      13
 |id-housenr|>1 | 13 |       2
 id = housenr-1 |  1 |       2
 id = housenr-1 | 11 |      12
 |id-housenr|>1 |  9 |       6
 |id-housenr|>1 | 15 |     123
 |id-housenr|>1 |  7 |      25
 id = housenr+1 | 10 |       9
 id = housenr-1 |  2 |       3
 |id-housenr|>1 | 12 |       1
 |id-housenr|>1 | 14 |       1
```

For example:

```
SELECT housenr, id, postcode,
       CASE WHEN id BETWEEN housenr AND postcode THEN 'yes' END AS yes_or_NULL
       FROM Address
       ORDER BY id DESC;
```

might yield the following values:

```
 housenr | id | postcode | yes_or_NULL
---------+----+----------+-------------
     123 | 15 |    77666 |
       1 | 14 |    86843 | yes
       2 | 13 |    86843 | yes
       1 | 12 |    86843 | yes
      12 | 11 |    45323 |
       9 | 10 |    45323 | yes
       6 |  9 |    45323 | yes
      29 |  8 |    13432 |
      25 |  7 |    13432 |
      13 |  6 |    13432 |
       7 |  5 |    54321 |
       4 |  4 |    54321 | yes
       3 |  3 |    54321 | yes
       3 |  2 |    12345 |
       2 |  1 |    12345 |
```

IF Expressions

The <if expression> is a proprietary extension of Standard SQL by Cisco ParStream to specify a conditional value:

```
<if expression> ::=
  "IF" "(" <search condition> "," <numeric value expression> "," <numeric value
      expression> ")"
  | "IF" "(" <search condition> "," <string value expression> "," <string value
      expression> ")"
```

- For <search condition>, see section 27.3.2, page 337.
- For <numeric value expression>, see section 27.3.3, page 344.
- For <string value expression>, see section 27.3.3, page 344.

The <result> of an <if expression> is the same as the <result> of a

```
<searched case> ::=
  "CASE" "WHEN" <search condition> "THEN" <numeric value expression>
                              "ELSE" <numeric value expression> "END"
  | "CASE" "WHEN" <search condition> "THEN" <string value expression>
                              "ELSE" <string value expression> "END"
```

Thus, for example:

```
SELECT IF (Val = 'Single', 1, 2) FROM MyTable
```

is a shortcut for:

```
SELECT CASE WHEN Val = 'Single' THEN 1 ELSE 2 END FROM MyTable
```

Usage: <if expression> is used

- in <nonparenthesized value expression primary> (see section 27.3.3, page 344).

For example:

```
SELECT id,
       houseno,
       IF (id=housenr OR id+id=housenr, 'yes', 'no') AS IfResult
    FROM Address
    ORDER BY id;
```

might yield the following:

```
 id | houseno | IfResult
----+---------+----------
  1 |       2 | yes
  2 |       3 | no
  3 |       3 | yes
  4 |       4 | yes
  5 |       7 | no
  6 |      13 | no
  7 |      25 | no
  8 |      29 | no
  9 |       6 | no
```

Or as a modification:

```
SELECT id,
       houseno,
       IF (id=housenr OR id+id=housenr, id/housenr, housenr*id) AS IfResult
    FROM Address
    ORDER BY id;
```

might yield the following:

```
 id | housenr | IfResult
----+---------+----------
  1 |       2 | 0.5
  2 |       3 | 6
  3 |       3 | 1
  4 |       4 | 1
  5 |       7 | 35
  6 |      13 | 78
  7 |      25 | 175
  8 |      29 | 232
  9 |       6 | 54
```

Literals

From Standard SQL, 5.3 <literal> Specify a non-null value:

```
<unsigned literal> ::=
   <unsigned numeric literal>
   | <general literal>

<general literal> ::=
   <character string literal>
   | <datetime literal>
   | <interval literal>

<character string literal> ::=
   <quote> [ <character representation> ... ] <quote>

<character representation> ::=
   <nonquote character>
   | <quote symbol>

<nonquote character> ::=
   !! any character other than a <quote>

<quote symbol> ::=
   <quote> <quote>

<unsigned numeric literal> ::=
   <exact numeric literal>
   | <approximate numeric literal>

<exact numeric literal> ::=
   <unsigned integer> [ "." [ <unsigned integer> ] ]

<sign> ::=
   "+" | "-"

<approximate numeric literal> ::=
   <mantissa> E <exponent>

<mantissa> ::=
   <exact numeric literal>

<exponent> ::=
   <signed integer>

<signed integer> ::=
   [ <sign> ] <unsigned integer>

<unsigned integer> ::=
   <digit>...

<datetime literal> ::=
   <date literal>
   | <shortdate literal>
   | <time literal>
   | <timestamp literal>

<date literal> ::=
   <date string>

<shortdate literal> ::=
   <shortdate string>

<time literal> ::=
   <time string>

<timestamp literal> ::=
   <timestamp string>

<date string> ::=
   "DATE" [ <space> ] <quote> <unquoted date string> <quote>

<shortdate string> ::=
   "SHORTDATE" [ <space> ] <quote> <unquoted date string> <quote>

<time string> ::=
   "TIME" [ <space> ] <quote> <unquoted time string> <quote>

<timestamp string> ::=
   "TIMESTAMP" [ <space> ] <quote> <unquoted timestamp string> <quote>

<date value> ::=
   <years value> "-" <months value> "-" <days value>

<time value> ::=
   <hours value> ":" <minutes value> ":" <seconds value>

<interval literal> ::=
   INTERVAL <interval string> <interval qualifier>

<interval string> ::=
   <quote> [ <sign> ] <unquoted interval string> <quote>

<unquoted interval string> ::=
   <day-time literal>

<day-time literal> ::=
   <day-time interval>
   | <time interval>

<day-time interval> ::=
   <days value> [ <space> <hours value> [ <colon> <minutes value> [ <colon> <seconds
       value> ] ] ]

<time interval> ::=
   <hours value> [ <colon> <minutes value> [ <colon> <seconds value> ] ]
   | <minutes value> [ <colon> <seconds value> ]
   | <seconds value>

<interval qualifier> ::=
   <start field> TO <end field>
   | <single datetime field>

<start field> ::=
   <non-second primary datetime field> [ <left paren> <interval leading field
       precision> <right paren> ]
```

```
<end field> ::=
   <non-second primary datetime field>
   | SECOND [ <left paren> <interval fractional seconds precision> <right paren> ]

<single datetime field> ::=
   <non-second primary datetime field> [ <left paren> <interval leading field
       precision> <right paren> ]
   | SECOND [ <left paren> <interval leading field precision> <right paren>
           [ <comma> <interval fractional seconds precision> ] <right paren> ] ]

<primary datetime field> ::=
   <non-second primary datetime field>
   | SECOND

<non-second primary datetime field> ::=
   DAY
   | HOUR
   | MINUTE

<interval fractional seconds precision> ::=
   <unsigned integer>

<interval leading field precision> ::=
   <unsigned integer>

<unquoted date string> ::=
   <date value>

<unquoted time string> ::=
   <time value>

<unquoted timestamp string> ::=
   <unquoted date string> <space> <unquoted time string>

<years value> ::=
   <unsigned integer>

<months value> ::=
   <unsigned integer>

<days value> ::=
   <unsigned integer>

<hours value> ::=
   <unsigned integer>

<minutes value> ::=
   <unsigned integer>

<seconds value> ::=
   <unsigned integer> [ "." [ <unsigned integer> ] ]
```

Note:

- This grammar includes Cisco ParStream's special type `shortdate`.
- A <nonquote character> is any character of the source language character set other than a <quote>.
- Two place a quote in a string literal use two quote characters. For example `'don''t'` represents the value “don't”.
- For <boolean value expression> see section 27.3.2, page 337.
- For <column reference> see section 27.3.2, page 342.
- For <set function specification> see section 27.3.4, page 347.
- For <case expression> see section 27.3.4, page 350.
- For <if expression> see section 27.3.4, page 354.
- For <digit>, <quote>, <double quote>, <space>, see section 27.3.2, page 336.

INSERT Statements

<insert statement> is used in <preparable SQL data statement>, which is used in <preparable statement> (see section 27.2, page 325).

Currently, Cisco ParStream only supports `INSERT INTO` statements:

```
<insert statement> ::=
   "INSERT" "INTO" <insertion target> <insert columns and source>

<insertion target> ::=
   <table name>

<insert columns and source> ::=
   <preparable SQL data statement>
```

For <table name> see section 27.3.2, page 333.
For details about `<preparable SQL data statement>` see section 27.2, page 325.

See section 10.7, page 107 for details, examples, and limitations of `INSERT INTO` statements.

DELETE Statements

<delete statement> is used in <preparable SQL data statement>, which is used in <preparable statement> (see section 27.2, page 325).

Cisco ParStream supports `DELETE` statements:

```
<delete statement> ::=
   "DELETE" <from clause> [ <where clause> ]
```

For details about `<from clause>` see section 27.3.1, page 329.
For details about `<where clause>` see section 27.3.1, page 331.

See section 11, page 109 for details, examples, and limitations of `DELETE` statements.

INSPECT Statements

<inspect statement> is used in <preparable SQL data statement>, which is used in <preparable statement> (see section 27.2, page 325).

The grammar for the INSPECT statement is:

```
<inspect statement> ::=
   "INSPECT" <inspection subject>
```

The INSPECT statement implements a metadata discovery mechanism similar to the system tables see section 26, page 306 but without involving the execution engine and thereby avoiding latencies introduced by resource consumption conflicts with other execution tasks. The downside is that no filtering, JOINs, or UNIONs can be used to analyze INSPECT results.

The following subjects are available for inspection (the values are case-insensitive):

- `ThreadPool`
- `ExecNodes`
- `ClusterConnectionPool`

Details about INSPECT ThreadPool

Subject **THREADPOOL**

- Lists tasks which are currently in execution and their respective resource consumption.
- C++ Class: `ExecThreadPool`
- Returned columns:

Column	Type	Meaning
start_time	TIMESTAMP	The UTC time when this task was issued, measured on the originating cluster-node (i.e. the query master in the case of a query)
execution_id	VARSTRING	execution ID of the running task (can be used to kill the task see section 27.11, page 375)
execution_type	VARSTRING	execution type of the running task, i.e. QUERY, IMPORT or MERGE
min_num_threads	UINT64	The minimum number of threads requested by this task before other tasks issued after it shall be considered (>0).
max_num_threads	UINT	The maximum number of threads that is allowed to be assigned to this task (0 means no limit).
current_num_threads	UINT64	The number of threads currently assigned to this task.
realtime_msec	UINT64	The amount of time spent in this tasks accumulated across all threads assigned to it.
cputime_msec	UINT64	The amount of CPU time (i.e. not counting IO wait cycles etc.) spent in this tasks accumulated across all threads assigned to it.
num_execnodes	UINT64	The number of active ExecNodes currently in this task.

Note the following:

- A task with the same query ID can appear several, non-contiguous times on a single-node cluster. For example when a query master issues several partial slave queries to calculate subquery results needed in large nested `JOIN`s. In this case accumulated resource measurements, currently only the realtime_msec and cputime_msec measurements, are only accumulated during contiguous time-spans and reset if at any moment no execution takes place for the task with the given query ID on this cluster node.

Schema Definition Statements

<SQL schema definition statement> is used in <SQL schema statement>, which is used in <preparable SQL schema statement>, which is used in <preparable statement> (see section 27.2, page 325)

The following subsections lists the grammar of all possible commands for the definition of schemas. Those are currently `CREATE TABLE` and `CREATE PROCEDURE`. See chapter 24, page 277 for further details of their functionality.

Schema Definition Grammar

The schema definition grammar is defined as followed:

```
<SQL schema definition statement> ::=
  <create table statement>
  | <SQL-invoked routine>

<SQL-invoked routine> ::=
  <schema routine>

<schema routine> ::=
  <schema procedure>
```

`<schema procedure>` is defined in see section 27.7.4, page 368

CREATE TABLE Grammar

In general, a `CREATE TABLE` statement has the following format:

```
<create table statement> ::=
   "CREATE" "TABLE" <table name>
   <column definition list>
   [ "PARTITION" "BY" <column or function list> ]
   [ "PARTITION" "ACCESS" <column list> [ "LIMIT" unsigned_int ] ]
   <distribution definition>
   [ "ORDER" "BY" <column list> ]
   [ "IMPORT_DIRECTORY_PATTERN" "'" dir_pattern "'" ]
   [ "IMPORT_FILE_PATTERN" "'" file_pattern "'" ]
   [ "ETL" "(" <etl select query> ")" ]
   [ <etlmerge list> ]
   ";"
```

Note that for backward compatibility you can use `PARTITIONED BY` instead of `PARTITION BY` (but not with `PARTITION ACCESS`) and `SORTED BY` instead of `ORDER BY`.

```
<column definition list> ::=
   "(" <column definition> [ { "," <column definition> }... ] ")"

<column list> ::=
   <column name> [ "," <column list> ]
```

```
<column or function list> ::=
   <column name or function>
   | <column or function list> "," <column name or function>

<column name or function> ::=
   <column name> | <function>

<function> ::=
   SQL function...

<etlmerge list> ::=
   <etlmerge> | <etlmerge list> <etlmerge>

<etlmerge> ::=
   "ETLMERGE" <etlmerge level> "(" <etl select query> ")"

<etlmerge level> ::=
   "MINUTE" | "HOUR" | "DAY" | "WEEK" | "MONTH"
```

A <distribution definition> may have the following format:

```
<distribution definition> ::=
   "DISTRIBUTE" "EVERYWHERE"
   | "DISTRIBUTE" "OVER" <column name> <distribution algorithm>

<distribution algorithm> ::=
   ["BY" "ROUND_ROBIN"] ["WITH" "REDUNDANCY" <redundancy> ] [<initial distribution> ]
   | "BY" "COLOCATION" "WITH" <table name>

<redundancy> ::=
   "1" .. "9"

<initial distribution> ::=
   "WITH" "INITIAL" "DISTRIBUTION" "(" <distribution rule list> ")"

<distribution rule list> ::=
   "(" <distribution rule> ")" [ "," <distribution rule list> ]

<distribution rule> ::=
   <value list> "TO" <server list>

<value list> ::=
   distr_value [ "," <value list> ]

<distr_value> ::=
   <integral literal> | <datetime literal> | "NULL"

<server list> ::=
   servername [ "," <server list> ]
```

Note:

- `<column name>` must be one of the elements of `PARTITION BY` clause, which is not using a function.

- Given a `DISTRIBUTE OVER` in absence of a `BY` ... clause let the algorithm default to `ROUND_ROBIN` (currently no other algorithm is supported, but this may change in future).
- With the `ROUND_ROBIN` algorithm in the absence of `WITH REDUNDANCY` the redundancy defaults to 2. Note that this means that you have to change this value for a cluster with only one query node.
- `COLOCATION WITH` must specify a reference table and the local distribution column must have the same type as one of the referenced table
- `DISTRIBUTE EVERYWHERE` implements a distribution, where all values are distributed over all query nodes.
- For backward compatibility you can use `DISTRIBUTED` instead of `DISTRIBUTE`.

For details of the table attributes see section 24.2.3, page 282.
For details of table distribution see section 6.3, page 53.

For details about ETL queries (`<etl select query>`), see section 27.3, page 327 and section 10.6, page 104.

For details about ETLMERGE queries (`<etlmerge list>`), see section 14.2, page 154.

Column Definitions

Column definitions may occur in CREATE TABLE statements (see section 27.7.2, page 364) and with restrictions in ALTER TABLE statements (see section 27.8.1, page 370).

They have the following format:

```
<column definition> ::=
   <column name> <column type>
   [ "DEFAULT" <default value>]
   [ "NOT" "NULL" ]
   [ "UNIQUE" | "PRIMARY" "KEY" ]
   [ "COMPRESSION" ("NONE" | <compression list>) ]
   [ "MAPPING_LEVEL" <mapping level> ]
   [ "MAPPING_TYPE" ( "AUTO" | "PROVIDED" ) ]
   [ "MAPPING_FILE_GRANULARITY" unsigned_int ]
   [ ( "SINGLE_VALUE" | "MULTI_VALUE" ) ]
   [ "PRELOAD_COLUMN" <preload_column> ]
   [ "SEPARATE" "BY" (<column list> | "NOTHING") ]
   [ "REFERENCES" (<column list> | "NOTHING") ]
   [ "INDEX" <index definition> [ "PRELOAD_INDEX" ( "NOTHING" | "COMPLETE" ) ] ]
   [ "DYNAMIC_COLUMNS_KEY" | "DYNAMIC_COLUMNS_VALUE" ]
   [ "CSV_COLUMN" ( "ETL" | unsigned_int ) ]
   [ "CSV_FORMAT" format_string ]
   [ "SKIP" ( "TRUE" | "FALSE" ) ]
```

Note that for backward compatibility you can use `SEPARATED BY` instead of `SEPARATE BY`.

```
<column type> ::=
   <numeric type> | <character string type> | <datetime type>
   | <binary large object string type> | <bitvector type>

<numeric type> ::=
   "UINT8" | "UINT16" | "UINT32" | "UINT64" | "INT8"
   | "INT16" | "INT32" | "INT64" | "FLOAT" | "DOUBLE"

<default value> ::=
   <literal>

<datetime type> ::=
   "DATE" | "SHORTDATE" | "TIME" | "TIMESTAMP"

<character string type> ::=
   "VARSTRING" [ "(" unsigned_int ")" ]

<binary large object string type> ::=
   "BLOB" [ "(" unsigned_int ")" ]

<bitvector type> ::=
   "BITVECTOR8"

<compression list> ::=
   <compression> [ "," <compression list> ]

<compression> ::=
   "LZ4" | "HASH64" | <sparse compression> | <dictionary compression>

<sparse compression> ::=
   "SPARSE" [ "SPARSE_DEFAULT" <sparse default value> ]

<dictionary compression> ::=
   "DICTIONARY"

<mapping level> ::=
   "PARTITION" | "TABLE" | <column name> | <unsigned integer>

<preload column> ::=
   "NOTHING" | "MEMORY_EFFICIENT" | "COMPLETE"

<column or column list> ::=
   column_name | "(" <column list> ")"

<index definition> ::=
   <numeric index> | <string index> | <datetime index>

<numeric index> ::=
   <index type definition> [ <numeric index bin definition> ] [ "INDEX_MASK"
       unsigned_int ]

<index type definition> ::=
   <index type> [ "MAX_CACHED_VALUES" unsigned_int ] [ "CACHE_NB_ITERATORS" bool ]

<index type> ::=
   "EQUAL" | "RANGE" | "NONE"

<numeric index bin definition> ::=
   <numeric index bin auto> | "INDEX_BIN_BOUNDARIES" "(" <numeric index bin
       boundaries> ")"
```

```
<numeric index bin auto> ::=
   "INDEX_BIN_COUNT" <integer literal>
   [ "INDEX_BIN_MIN" "(" "MIN" | <numeric literal> ")" ]
   [ "INDEX_BIN_MAX" "(" "MAX" | <numeric literal> ")" ]

<numeric index bin boundaries> ::=
   <numeric literal> "," <numeric literal> | <numeric index bin boundaries> ","
       <numeric literal>

<string index> ::=
   <index type>

<datetime index> ::=
   <index type definition> [ "INDEX_GRANULARITY" <datetime index granularity> ]

<datetime index granularity> ::=
   "YEAR" | "MONTH" | "DAY" | "HOUR" | "MINUTE" | "SECOND" | "MILLISECOND"

<sparse default value> ::=
   "NULL" | <literal>
```

Note the following:

- If the default column value `DEFAULT` is not set `NULL` is used.
- If `MAPPING_TYPE PROVIDED` is used the only possible value for `DEFAULT` is `NULL`.
- `DEFAULT NULL` and `NOT NULL` are exclusive.
- `DEFAULT` cannot be set for `NOT NULL` or `PRIMARY KEY` columns.
- The value set as `DEFAULT` is only used for reading existing partitions (e.g. after a DDL change); new data imports e.g. from an `INSERT INTO` still to be provided with values for all needed columns.
- If the default value in `COMPRESSION SPARSE_DEFAULT` (see section 15.10.1, page 178) is omitted, the value of `DEFAULT` is used (which defaults to `NULL`). If a string literal is specified as sparse default value, its hash value is computed and used as default value.
- For SQL literals, see section 27.3.5, page 355.

For details of the column attributes see section 24.2.4, page 284.

CREATE PROCEDURE Grammar

The grammar of a `CREATE PROCEDURE` statement is defined as followed:

```
<schema procedure> ::=
  "CREATE" "PROCEDURE" <routine name> <SQL parameter declaration list>
  <routine characteristics>
  <routine body>

<routine name> ::=
  <qualified identifier>

<SQL parameter declaration list> ::=
  "(" [ <SQL parameter declaration>
    [ { <comma> <SQL parameter declaration> }... ] ] ")"
```

```
<SQL parameter declaration> ::=
  <SQL parameter name> <parameter type>

<parameter type> ::=
  <data type>

<routine characteristics> ::=
  [ <language clause> ]

<language clause> ::=
  "LANGUAGE" <language name>

<language name> ::=
  "SQL"

<routine body> ::=
  "AS" <SQL routine body>

<SQL routine body> ::=
  <SQL procedure statement>

<SQL procedure statement> ::=
  <preparable statement>
```

Schema Manipulation Statements

<SQL schema manipulation statement> is used in <SQL schema statement>, which is used in <preparable SQL schema statement>, which is used in <preparable statement> (see section 27.2, page 325)

Cisco ParStream supports the following schema manipulation statements:

- `ALTER TABLE` statement (used to modify definitions of existing tables)
- `DROP TABLE` statement (used to delete existing tables)
- `DROP PROCEDURE` statement (used to delete existing stored procedures)

```
<SQL schema manipulation statement> ::=
  <alter table statement>
  | <drop table statement>
  | <drop procedure statement>
```

`ALTER TABLE` Statements

```
<alter table statement> ::=
   "ALTER" "TABLE" <table name> <alter table action>

<alter table action> ::=
  <add column definition>

<add column definition> ::=
   "ADD" "COLUMN" <column definition>
```

For <table name> see section 27.3.2, page 333.

<column definition> follows the grammar defined in Section 27.7.3 with the following restrictions:

- added columns cannot be `UNIQUE` or `PRIMARY KEY`, and
- `CSV_COLUMN` cannot be set for added columns.

See section 24.3, page 295 for details about the functionality and usage of `ALTER TABLE`.

`DROP TABLE` Statements

```
<drop table statement> ::=
   "DROP" "TABLE" <table name>
```

For <table name> see section 27.3.2, page 333.

`DROP PROCEDURE` Statements

```
<drop procedure statement> ::=
   "DROP" "PROCEDURE" <routine name>
```

For <routine name> see section 27.7.4, page 368.

CALL Statement (Control Statements)

<SQL control statement> is used in <preparable SQL control statement>, which is used in <preparable statement> (see section 27.2, page 325)

<SQL control statement>

Standard SQL, 13.5 defines control commands like CALL with the following grammar:

```
<SQL control statement> ::=
   <call statement>

<call statement> ::=
   "CALL" <routine invocation>

<routine invocation> ::=
   <routine name> <SQL argument list>

<SQL argument list> ::=
   "(" [ <SQL argument> [ { <comma> <SQL argument> }... ] ] ")"

<SQL argument> ::=
   <literal>
```

Thus SQL arguements are currently restricted to literals instead of accepting value expressions.

For <routine name> see section 27.7.4, page 368.

SET Statements (Session Statements)

<SQL session statement> is used in <preparable SQL session statement>, which is used in <preparable statement> (see section 27.2, page 325)

<SQL session statement>

Standard SQL, 13.5 defines various `SET` commands in <SQL procedure statement>. However, Cisco ParStream defines different commands, so that Cisco ParStream has the following BNF here:

```
<SQL session statement> ::=
   "SET" <variable> ( "TO" | "=" ) ( <literal> | <regular identifier> | "DEFAULT" |
       "LOW" | "MEDIUM" | "HIGH");
```

The following variables are possible to set:

Variable:	**`LimitQueryRuntime`**
Default:	0
Effect:	Session specific ability to overwrite the global option `limitQueryRuntime` (see section 13.2.1, page 120) to interrupt queries running for longer than this time in milliseconds. A value of 0 disables the time limit.

Variable:	**`NumBufferedRows`**
Default:	32
Effect:	Session specific ability to overwrite the global option `numBufferedRows` (see section 13.2.1, page 120) to define the size of the buffer for output.

Variable:	**`OutputFormat`**
Default:	`ASCII`
Effect:	Session specific ability to overwrite the global option `outputformat` (see section 13.2.1, page 121) See section 16.3, page 199 for details about the different output formats.

Variable:	**`AsciiOutputColumnSeparator`**
Default:	`;`
Effect:	Session specific ability to overwrite the global option `asciiOutputColumnSeparator` (see section 13.2.1, page 121) See section 16.3, page 199 for details about the different output formats.

Variable:	**`AsciiOutputMultiValueSeparator`**
Default:	`,`
Effect:	Session specific ability to overwrite the global option `asciiOutputMultiValueSeparator` (see section 13.2.1, page 122) See section 16.3, page 199 for details about the different output formats.

Variable:	**`AsciiOutputNullRepresentation`**
Default:	`<NULL>`
Effect:	Session specific ability to overwrite the global option `asciiOutputNullRepresentation` (see section 13.2.1, page 122) See section 16.3, page 199 for details about the different output formats.

Variable:	**`QueryPriority`**
Default:	defaultQueryPriority
Effect:	Session specific priority value to be used for query tasks issued in this session (i.e. `SELECT` ...). Active for all subsequently issued query tasks in this session until changed. For further details and possible values see section 15.1, page 158.

Variable:	**`ImportPriority`**
Default:	defaultImportPriority
Effect:	Session specific priority value to be used for import tasks issued in this session (i.e. `INSERT INTO` ...) Active for all subsequently issued import tasks in this session until changed. For further details and possible values see section 15.1, page 158.

In addition, you can change the value of all ExecTree options and all optimization options:

Variable	Description
`ExecTree.`*`Option`*	Set parameters that influence the performance of query execution. *Option* is a case-insensitive ExecTree option, for further details see section 13.3.4, page 140.
`optimization.rewrite.`*`Option`*	Set parameters that influence query rewrite optimizations. *Option* is a case-insensitive query rewrite option, for further details see section 13.5, page 149.

When setting the value to the reserved word `DEFAULT`, the value is restored to its default value.

Note that you can set the `OutputFormat` only for the client socket interface, such as netcat or pnc (see section 16, page 199), while the other options can be set using all client interfaces.

Note that for some options you have to use the `ALTER SYSTEM SET` command (see section 27.11.1, page 375).

ALTER SYSTEM Statements (System Statement)

<SQL system statement> is used in <preparable SQL system statement>, which is used in <preparable statement> (see section 27.2, page 325)

```
<SQL system statement> ::=
   "ALTER" "SYSTEM" [<target>] <system action and arguments>
<target> ::=
   "CLUSTER" | "NODE" [<node_name>]
<system action and arguments> ::=
   "SET" <variable> ( "TO" | "=" ) ( <literal> | "DEFAULT" )
   "KILL" <execution_id>
   "DISABLE" ("IMPORT" | "MERGE")
   "ENABLE" ("IMPORT" | "MERGE")
   "SHUTDOWN"
   "EXECUTE" "MERGE" "LEVEL" <merge_level> ["SCOPE" "GLOBAL"]
<execution_id> ::= <literal>
<node_name> ::= <literal>
<merge_level> ::= ("MINUTE" | "HOUR" | "DAY" | "WEEK" | "MONTH")
```

The optional target is only for some commands possible (see below).

The available actions are:

Action	Argument	Target	Description
KILL	executionID	no	Kill task (see section 27.6, page 362)
SET	*option* =/TO *value*	no	Set server option (for all sessions) to new value (see section 27.11.1, page 375
DISABLE	IMPORT/MERGE	no	disable data import or merges (see section 16.4.1, page 200)
ENABLE	IMPORT/MERGE	no	enable data import or merges (see section 16.4.1, page 200)
EXECUTE	MERGE	no	executes the specified merge (see section 16.4.1, page 201)
SHUTDOWN		yes	shut down a server/node or a cluster as a whole (see section 16.4.1, page 200)

ALTER SYSTEM SET

For the ALTER SYSTEM SET command, in the SET clause the same syntax as for SET commands is used (see section 27.10, page 373).

Currently, the following options are possible to set:

Option	Description
`mappedFilesAfterUnmapFactor`	See section 13.3.2, page 138.
`mappedFilesCheckInterval`	See section 13.3.2, page 138.
`mappedFilesMax`	See section 13.3.2, page 138.
`mappedFilesOutdatedInterval`	See section 13.3.2, page 138.
`DebugLevel.`*name*	*name* might be `defaultDebugLevel` or a class name. See section 9.5.4, page 86 for details.

All options can be set to `DEFAULT`.

User Administration Statements

<SQL user administration statement> is used in <preparable SQL user administration statement>, which is used in <preparable statement> (see section 27.2, page 325)

```
<SQL user administration statement> ::=
   "CREATE" "USER" <user name> ["WITH" "LOGIN" <login name>]
   | "DROP" "USER" <login name>

<user name> ::=
   <string literal>

<login name> ::=
   <string literal>
```

- For <regular identifier> see section 27.3.2, page 334.
- See section 9.2.3, page 80 for details.

DBMS Job Scheduler

<SQL scheduler statement> has the following syntax:

```
<SQL scheduler statement> ::=
   "CREATE" "JOB" <job name> <json object>
   | "DROP" "JOB" <job name>
   | "ENABLE" "JOB" <job name>
   | "DISABLE" "JOB" <job name>
   | "RUN" "JOB" <job name>

<json object> ::=
   "{" <double quote> "action" <double quote> ":" <double quote> <SQL statement>
       <double quote> ","
      <double quote> "timing" <double quote> ":" <double quote> <cron timing> <double
          quote>
      ["," <double quote> "comment" <double quote> ":" <double quote> <string> <double
          quote> ]
      ["," <double quote> "enabled" <double quote> ":" <boolean> ]
   "}"
```

For <job name>, see section 27.3.2, page 333.
For <cron timing>, see section 14.1.2, page 154.

Reserved Keywords

SQL forbids the usage of SQL keywords as identifiers. In addition, Cisco ParStream introduces several additional keywords. These two lists of reserved keywords are provided here.

Note that when specifying table or column names, you have to use double quotes when these names case-insensitively conflict with a keyword.

Reserved Standard SQL Keywords

A

- ABS
- ALL
- ALLOCATE
- ALTER
- AND
- ANY
- ARE
- ARRAY
- ARRAY_AGG
- AS
- ASENSITIVE
- ASYMMETRIC
- AT
- ATOMIC
- AUTHORIZATION
- AVG

B

- BEGIN
- BETWEEN
- BIGINT
- BINARY
- BLOB
- BOOLEAN
- BOTH
- BY

C

- CALL
- CALLED
- CARDINALITY
- CASCADED
- CASE
- CAST
- CEIL
- CEILING
- CHAR
- CHAR_LENGTH
- CHARACTER
- CHARACTER_LENGTH
- CHECK
- CLOB
- CLOSE
- COALESCE
- COLLATE
- COLLECT
- COLUMN
- COMMIT
- CONDITION
- CONNECT
- CONSTRAINT
- CONVERT
- CORR
- CORRESPONDING
- COUNT
- COVAR_POP
- COVAR_SAMP
- CREATE
- CROSS
- CUBE
- CUME_DIST
- CURRENT
- CURRENT_CATALOG
- CURRENT_DATE
- CURRENT_DEFAULT_TRANSFORM_GROUP
- CURRENT_PATH
- CURRENT_ROLE
- CURRENT_SCHEMA
- CURRENT_TIME
- CURRENT_TIMESTAMP

CURRENT_TRANSFORM_GROUP_FOR_TYPE
CURRENT_USER
CURSOR
CYCLE

D

DATE
DAY
DEALLOCATE
DEC
DECIMAL
DECLARE
DEFAULT
DELETE
DENSE_RANK
DEREF
DESCRIBE
DETERMINISTIC
DISCONNECT
DISTINCT
DOUBLE
DROP
DYNAMIC

E

EACH
ELEMENT
ELSE
END
ESCAPE
EVERY
EXCEPT
EXEC
EXECUTE
EXISTS
EXP
EXTERNAL
EXTRACT

F

FALSE
FETCH
FILTER
FLOAT
FLOOR
FOR
FOREIGN
FREE
FROM
FULL
FUNCTION
FUSION

G

GET
GLOBAL
GRANT
GROUP
GROUPING

H

HAVING
HOLD
HOUR

I

IDENTITY
IN
INDICATOR
INNER
INOUT
INSENSITIVE
INSERT
INT
INTEGER
INTERSECT
INTERSECTION
INTERVAL
INTO
IS

J

JOIN

L

LANGUAGE
LARGE
LATERAL
LEADING
LEFT
LIKE
LIKE_REGEX
LN
LOCAL
LOCALTIME

LOCALTIMESTAMP
LOWER

M

MATCH
MAX
MEMBER
MERGE
METHOD
MIN
MINUTE
MOD
MODIFIES
MODULE
MONTH
MULTISET

N

NATIONAL
NATURAL
NCHAR
NCLOB
NEW
NO
NONE
NORMALIZE
NOT
NULL
NULLIF
NUMERIC

O

OCTET_LENGTH
OCCURRENCES_REGEX
OF
OLD
ON
ONLY
OPEN
OR
ORDER
OUT
OUTER
OVER
OVERLAPS
OVERLAY

P

PARAMETER
PARTITION
PERCENT_RANK
PERCENTILE_CONT
PERCENTILE_DISC
POSITION
POSITION_REGEX
POWER
PRECISION
PREPARE
PRIMARY
PROCEDURE

R

RANGE
RANK
READS
REAL
RECURSIVE
REF
REFERENCES
REFERENCING
REGR_AVGX
REGR_AVGY
REGR_COUNT
REGR_INTERCEPT
REGR_R2
REGR_SLOPE
REGR_SXX
REGR_SXY
REGR_SYY
RELEASE
RESULT
RETURN
RETURNS
REVOKE
RIGHT
ROLLBACK
ROLLUP
ROW
ROW_NUMBER
ROWS

S

SAVEPOINT
SCOPE
SCROLL

SEARCH
SECOND
SELECT
SENSITIVE
SESSION_USER
SET
SIMILAR
SMALLINT
SOME
SPECIFIC
SPECIFICTYPE
SQL
SQLEXCEPTION
SQLSTATE
SQLWARNING
SQRT
START
STATIC
STDDEV_POP
STDDEV_SAMP
SUBMULTISET
SUBSTRING
SUBSTRING_REGEX
SUM
SYMMETRIC
SYSTEM
SYSTEM_USER

T

TABLE
TABLESAMPLE
THEN
TIME
TIMESTAMP
TIMEZONE_HOUR
TIMEZONE_MINUTE
TO
TRAILING
TRANSLATE
TRANSLATE_REGEX
TRANSLATION
TREAT
TRIGGER
TRUNCATE
TRIM
TRUE

U

UESCAPE
UNION
UNIQUE
UNKNOWN
UNNEST
UPDATE
UPPER
USER
USING

V

VALUE
VALUES
VAR_POP
VAR_SAMP
VARBINARY
VARCHAR
VARYING

W

WHEN
WHENEVER
WHERE
WIDTH_BUCKET
WINDOW
WITH
WITHIN
WITHOUT

Y

YEAR

Z

Reserved Cisco ParStream Keywords

A

ACCESS
AGGREGATE_OR
ALL_AVAILABLE
ANALYZE
ANY_ONE
AUTO

B

BIT
BITMAP_COMBINE
BITVECTOR8

C

CACHE_NB_ITERATORS
CLUSTER
COLOCATION
COMMAND_LINE_ARGS
COMPLETE
COMPRESSION
CONFIGVALUE
CONTAINSTRIPLETS
COUNTDISTINCT
COUNTIF
CSVFETCH
CSV_COLUMN
CSV_FORMAT

D

DATE_PART
DATE_TRUNC
DAYOFMONTH
DAYOFWEEK
DAYOFYEAR
DICTIONARY
DISABLE
DISTRIBUTE
DISTRIBUTED
DISTRIBUTION
DISTVALUES
DOW
DOY
DYNAMIC_COLUMN_NAMES
DYNAMIC_COLUMNS
DYNAMIC_COLUMNS_KEY
DYNAMIC_COLUMNS_VALUE

E

ENABLE
EPOCH
EQUAL
ETL
ETLMERGE
EVERYWHERE
EXECNODES

F

FIXSTRING
FORWARDED

H

HASH64
HASHWORDS
HIGH

I

IF
IFNULL
IMPORT
IMPORT_DIRECTORY_PATTERN
IMPORT_FILE_PATTERN
INDEX
INDEX_BIN_BOUNDARIES
INDEX_BIN_COUNT
INDEX_BIN_MAX
INDEX_BIN_MIN
INDEX_BITS
INDEX_GRANULARITY
INDEX_MASK
INF
INITIAL
INSPECT
INT16
INT32
INT64
INT8
ISODOW
ISOYEAR

J

JOB
JOIN_COLUMNS

K

KILL

L

LEVEL
LIKEFUNC
LIMIT
LOGIN
LOOKUP
LOW
LOWERCASE
LZ4

M

MAPPING_FILE_GRANULARITY
MAPPING_LEVEL
MAPPING_TYPE
MATCHES
MATCHESFUNC
MAX_CACHED_VALUES
MEDIAN
MEDIUM
MEMORY_EFFICIENT
MILLISECOND
MULTI_VALUE

N

NAN
NAND
NODE
NOR
NOTHING
NOW

O

OFFSET

P

PARTITIONED
PG_CLIENT_ENCODING
PRELOAD_COLUMN
PRELOAD_INDEX
PROVIDED

Q

QUARTER
QUERIES

R

REDUNDANCY
ROUND_ROBIN
RUN

S

SEPARATE
SEPARATED
SHL
SHORTDATE
SHUTDOWN
SINGLE_VALUE
SKIP
SLEEP
SORTED
SPARSE
STATIC_COLUMN_NAMES
SUMIF

T

TAKE
THREADPOOL
TRIPLETS
TRUNC

U

UINT16
UINT32
UINT64
UINT8
UINTEGER
UPPERCASE
USMALLINT

V

VARSTRING

W

WEEK

X

XOR

Release Notes

Release Notes Version 6.2

New Features of Version 6.2

- Updated CiscoSSL to fixed CVE-2019-1563
- Added LimitNOFILE to systemd service script template to set the limit correctly for servers started via systemd.

Changes Incompatible with prior Versions

Please note that Version 6.2 adds some fixes and new feature that **might break existing behavior**:

Known Issues for Version 6.2

- If you stop a Cisco ParStream server using `systemctl stop` it will be restarted on the next reboot. If you use systemd to start your server, you should only use systemctl to stop your parstream server. If you use an ALTER SYSTEM CLUSTER shutdown command, the systemd service will immediately restart the server, which might leave it in an undefined state.

Changes Incompatible with prior Versions since 6.2.0

Release Notes Version 6.1

New Features of Version 6.1

Changes Incompatible with prior Versions

Please note that Version 6.1 adds some fixes and new feature that **might break existing behavior**:

- The default value for `partitionMergeRequestTimeout` in the manual was corrected to `60`.
- The default value for `queryThrottlingInterval` in the manual was corrected to `500`.
- The default value for `requestDefaultTimeout` in the manual was corrected to `60`.
- The default value for `synchronizePartitionRequestTimeout` in the manual was corrected to `120`.
- The general options `indicesCacheNbIterators`, `indicesMaxCachedValues`, `partitionSearchConcurrencyLevel`, `requestLongTermTimeout`, and `requestShortTermTimeout` were deprecated and now emit a warning if used

- The import options `csvimport`, `defaultMergePriority`, `defaultImportPriority`, `defaultQueryPriority`, `loadtables`, `maxMergeThreads`, and `serverJournalDir` were deprecated and now emit a warning if used
- The check for the configured number of mapped files and open file handles in the operating system now produces an error instead of a warning. If the limit was ignored delibaretely, you can enable the new option `overrideProcessRequirements` (See section 13.2, page 119) to restore the old behavior.
- Updated the supported PostgreSQL ODBC driver to version 11.00.0000
- Added global option `clientConnectionTimeout` to configure the time in seconds after which the server will drop inactive client connections. (See section 13.2, page 119)

Known Issues for Version 6.1

Changes Incompatible with prior Versions since 6.1.0

Release Notes Version 6.0

New Features of Version 6.0

- Change dependency from system OpenSSL library to packaged CiscoSSL library.
- Add TLS encryption support for client and inter-cluster connections.
- Stored Procedures with `CREATE PROCEDURE`, `CALL` and `DROP PROCEDURE`. For further details see section 9.4.1, page 83.

Changes Incompatible with prior Versions

Please note that Version 6.0 adds some fixes and new feature that **might break existing behavior**:

- Parsing column definitions of `CREATE TABLE` statements require comma separation

Known Issues for Version 6.0

-

Changes Incompatible with prior Versions since 6.0.0

-

Release Notes Version 5.4

New Features of Version 5.4

- Added general option ’ignoreBrokenPartitions’ for single node clusters. For further details, see section 13.2, page 119.
- Added general option ’validatePartitions’. For further details, see section 13.2, page 119.
- Added warning to manual about removing journal directory.
- Added option ’synchronizeFilesystemWrites’ for secure filesystem writes. For further details, see section 13.2, page 119.

Release Notes Version 5.3

New Features of Version 5.3

- Added systemd daemon for Cisco ParStream server (See section 2.9, page 12)

Release Notes Version 5.2

Changes Incompatible with prior Versions

Please note that Version 5.2 adds some fixes and new feature that **might break existing behavior**:

- For improved memory handling Cisco ParStream 5.2.0 now uses tcmalloc. Tcmalloc is required to operate Cisco ParStream. Please refer to the installation guide for additional information.

Release Notes Version 5.1

Changes Incompatible with prior Versions

Please note that Version 5.1 adds some fixes and new feature that **might break existing behavior**:

- Default value for MonitoringMinLifeTime, MonitoringImportMinLifeTime, and MonitoringMergeMinLifeTime were changed to zero, effectively disabling the monitoring by default.

Release Notes Version 5.0

New Features of Version 5.0

- Added support for day-time interval literals
- We add a DBMS scheduler to Cisco ParStream to support periodic task execution.
- Added support for SQL DELETE commands

Changes Incompatible with prior Versions

Please note that Version 5.0 adds some fixes and new feature that **might break existing behavior**:

- Starting with Version 5.0.0, Cisco ParStream no longer supports the old bitmap index file format (*.sbi).

Release Notes Version 4.4

New Features of Version 4.4

- Cisco ParStream is now supported on Ubuntu 16.04
- Unified time zone handling for Java datatypes (See incompatible changes for details).

Changes Incompatible with prior Versions

Please note that Version 4.4 adds some fixes and new feature that **might break existing behavior**:

- The licensing mechanism was completely removed from Cisco ParStream. You no longer need to provide a valid license in order to start the server. In case your license expired please contact your Cisco sales representative.
- Starting a server with `--licensefile` now results in a startup error.
- Cisco ParStream is not supported on Ubuntu 14.04 anymore.
- Migrating tables from prior versions of Cisco ParStream with no `PARTITION BY` clause is no longer supported.
- The Java Streaming Import will handle the time zone of Java Language Date Time values correct for TIMESTAMP, TIME and DATE columns regarding the column data type definition (see section 19.4, page 222).

  The Java Streaming Import was broken while importing date time data running in a Java VM with a time zone other than UTC.

  The import behaviour for the following examples was for a Java VM running with time zone `Europe/Berlin` and a current zone offset +01:00:

  Inserting Java date time values in pre 4.4.0 versions:

Column Type	Java Date Time Instance	Column Data
`TIMESTAMP`	Timestamp.valueOf("2016-06-01 00:30:01.256")	"2016-05-31 22:30:01.256"
`TIMESTAMP`	new GregorianCalendar(2016, 5, 1, 0, 30, 1)	"2016-06-01 00:30:01.000"
`DATE`	Date.valueOf("2016-06-01")	"2016-05-31"
`TIME`	Time.valueOf("00:30:01")	"23:30:01.000"

  Inserting Java date time values starting with Version 4.4.0:

Column Type	Java Date Time Instance	Column Data
`TIMESTAMP`	Timestamp.valueOf("2016-06-01 00:30:01.256")	"2016-06-01 00:30:01.256"
`TIMESTAMP`	new GregorianCalendar(2016, 5, 1, 0, 30, 1)	"2016-06-01 00:30:01.000"
`DATE`	Date.valueOf("2016-06-01")	"2016-06-01"
`TIME`	Time.valueOf("00:30:01")	"00:30:01.000"

- The following messages aren't displayed anymore: write_file, write_partition.
- The following messages aren't displayed anymore on slaves: insert_data and query_send.
- Prot message delete partition is now a debug message.

Known Issues for Version 4.4

- Always set importHistoryMaxEntries to zero. Any other value than zero can lead to memory exhaustion.
- Always use `LIKE` instead of `CONTAINS` in SQL queries wherever possible. `LIKE` offers better performance and a smaller memory footprint during operation.
- In order to improve ease of use in our `DYNAMIC_COLUMNS` feature, we plan to introduce incompatible changes in future versions. Therefore, we advise that all values of key columns are transformed to lower case prior to importing them into Cisco ParStream.

Changes Incompatible with prior Versions since 4.4.4

- Since 4.4.4, the defaults of the server option `connectionpool.connectionFetchTimeout` was increased from 500 ms to 5000 ms to avoid connection losses and instabilities during high load phases.

Examples

This chapter gives an overview of some example demonstrating the abilities and usage of Cisco ParStream.

The examples are:

- The example **cluster** (see section A.1, page 392) is, as the name implies, an example for running Cisco ParStream in a cluster.
- The example **multivalue** (see section A.2, page 392) deals with the usage of multivalue types.
- The example **stringdistr** (see section A.4, page 393) is an example for running Cisco ParStream in a cluster with a predefined distribution according to string values.
- The example **dynamiccolumns** (see section A.3, page 392) is an example for running Cisco ParStream in a cluster with the dynamic columns feature.
- The example **xUDTO** (see section A.5, page 393) demonstrates the usage of external User-Defined Table Operations.

Note that chapter B, page 394 lists further examples to use the Cisco ParStream API's.

The examples can be installed with the provided 'parstream-database-examples' package. All examples will then be located in the `examples` directory of the installed release in a subdirectory with the corresponding name.

Example 'cluster'

The example **cluster** shows how to set up a Cisco ParStream cluster. (see chapter 6, page 40).

For further information about the example and instructions on how to run it, please see the `README.md` in `examples/cluster`.

Example 'multivalue'

The Cisco ParStream example **multivalue** demonstrates the usage of the multivalue field (see section 23.8, page 275).

For further information about the example and instructions on how to run it, please see the `README.md` in `examples/multivalue`.

Example 'dynamiccolumns'

The example **dynamiccolumns** shows how to set up a Cisco ParStream cluster that uses the dynamic column feature (see chapter 7, page 61).

For further information about the example and instructions on how to run it, please see the `README.md` in `examples/dynamiccolumns`.

Example ‘stringdistr’

The example **stringdistr** shows how to set up a Cisco ParStream cluster with an initial distribution (see section 6.3.1, page 57) for a specific string column.

For further information about the example and instructions on how to run it, please see the `README.md` in `examples/stringdistr`.

Example ‘xUDTO’ Defining External User-Defined Table Operators

The example `xUDTO` shows how to use Cisco ParStream’s external User-Defined Table Operators (xUDTO). It contains an R script that is used by SQL queries to fit linear models. For a detailed description about the xUDTO API see section 20, page 232.

For further information about the example and instructions on how to run it, please see the `README.md` in `examples/xUDTO`.

API Examples

This chapter gives an overview of some example demonstrating the abilities and usage of the Cisco ParStream API's. See chapter A, page 392 for some examples for the general usage of Cisco ParStream.

The examples are:

- The example **importapi_java** (see section B.1, page 394) demonstrates the usage of Cisco ParStream's Streaming Import Interface for streaming imports by a JAVA client.
- The example **jdbc** (see section B.2, page 394) demonstrates the usage of the JDBC driver.

The examples can be installed with the provided 'parstream-client-examples' package. All examples will then be located in the `examples` directory of the installed release in a subdirectory with the corresponding name.

Example 'importapi_java' Using the Streaming Import Interface by a JAVA Client

The example **importapi_java** shows how to use Cisco ParStream's Java Streaming Import Interface (see chapter 19, page 216) by a JAVA client.

It consists of the following directories and files:

- A `README.txt` file, explaining how to build the example, start the server, and run the example.
- A `conf` directory, holding the corresponding server configurations for a database with a schema the example import expects.
- A `example/src` subdirectory, containing the example JAVA client that demonstrates how to use the Cisco ParStream's Streaming Import Interface.
- A `example/pom.xml` file, which can be used to build the example client with Maven.

See section 19.4, page 218 for a detailed description of the function calls the example performs.

Example 'jdbc' Using the JDBC Driver

The example **jdbc** shows how to use Cisco ParStream's JDBC Client Interface (see chapter 18, page 214). It contains a simple Java client that sends SQL queries to the database via JDBC and prints the results to the console.

For further information about the example and instructions on how to run it, please see the `README.md` in `examples/jdbc`.

Glossary

C

cmake

– Tool for creating makefiles and different IDE-Projects from a single definition file. Used internally and for all provided examples. See `http://www.cmake.org`.

connection set

– A set of connections so that each node in a cluster can communicate with each other node.

D

description tree

– For each query, the description tree consists of a number of DescriptionNodes defining the logical stages of a query. Every description tree is translated into one or more execution trees performing the real processing part of every query.

E

execution tree

– The execution tree is the real processing part of every query. It is generated out of the description tree.

G

conf directory

– The directory with database configuration files (such as INI files). Usually `.conf` or a directory passed with `--confdir`.

P

logical partition

– A logical group of data separated from other data by some specific criteria (often distinct values of partitioning columns). Logical partitions can consist out of multiple physical partitions.

N

node

– A logical element of a cluster. It can be a query or an import node. Multiple nodes can run on the same system.

O

option

– The name of a parameter to configure the behavior of Cisco ParStream. Your own values can be set in INI files or via command-line argument.

P

partition

– A logical or physical part of the database. Logical partitions can consist of multiple physical partitions.

physical partition

- A directory with of data of a logical partition imported or merged together by the same process.

Q

query master

- In a multi-node cluster, the node which received the request from the client. It might change for each query. This node is responsible for distributing the query to the query-slaves and consolidate the sub results.

query slave

- In a multi-node cluster, all nodes which did not receive the request from the client (i.e. are no query-master for this request). They calculate the sub results which are consolidated by the query-master.

Index