

RapidsDB

RapidsDB User Guide

Release 4.3.3



1	Overview	10
1.1	Changes	10
1.1.1	Changes from 4.3	10
1.1.2	Changes from 4.2.3.2	10
1.1.3	Changes from 4.2.3.1	10
1.1.4	Changes from 4.2.3	10
1.1.5	Changes from 4.2.2	10
1.1.6	Changes from 4.2.1	10
1.1.7	Changes from 4.2	11
1.1.8	Changes from 4.1	11
1.1.9	Changes from 4.03	11
1.1.10	Changes from R3.6	11
1.1.11	Changes from 3.4.2	11
1.1.12	Changes from Release 3.4.1.....	11
1.1.13	Changes from Release 3.4.....	11
1.1.14	Changes from Release 3.3.2.....	11
1.1.15	Changes from Release 3.3.....	12
1.1.16	Changes from Release 3.1.....	13
1.2	What is RapidsDB?	14
1.3	RapidsDB Components.....	15
1.3.1	RapidsDB Plex.....	15
1.3.2	SQL Compiler and Optimizer.....	15
1.3.3	Massively Parallel Processing (MPP) Execution Engine	15
1.3.4	Federated Connectors.....	15
1.3.5	MOXE	16
1.3.6	Client API.....	16
1.3.6.1	RapidsDB Manager.....	16
1.3.6.2	rapids-shell	16
1.3.6.3	JBDC.....	16
1.3.6.4	Wireline Protocol	16
1.3.7	Zookeeper	16
1.4	RapidsDB Cluster Topology	17

2	Federations, Connectors and Naming	18
2.1	Overview	18
2.2	Connectors	18
2.3	Table Naming	19
2.4	Retrieval and Storage of Schema Metadata	20
2.5	Object Name Resolution and Case Sensitivity	22
2.6	Connector Lookup of Object Names (Default).....	23
2.7	Case-sensitive Lookups	26
2.8	Handling Table Metadata.....	28
2.9	Mapping Catalog, Schema and Table Names.....	28
3	Operational Considerations for Connectors	29
3.1	MOXE Connector.....	29
3.2	RDBMS Connectors	31
3.3	Generic JDBC Connector	32
3.4	Hadoop Connector	32
3.4.1	Partitioning.....	32
3.4.1.1	Delimited Files.....	33
3.4.1.2	ORC and Parquet Files.....	34
3.4.2	Hive-style Partitioning.....	35
3.4.3	Writing Data to HDFS	36
3.5	IMPEX Connector	36
4	Query Interfaces	37
4.1	RapidsDB Command Line Interface (rapids-shell).....	37
4.1.1	Running the rapids-shell Locally	38
4.1.2	Running the rapids-shell Remotely	38
4.1.3	Authentication of the rapids-shell	39
4.2	Programmatic Interfaces	40
4.2.1	JDBC	40
4.2.2	Invoking the rapids-shell Programmatically.....	41
5	SQL Syntax.....	41
5.1	Lexical Structure.....	41
5.1.1	Identifiers and Keywords	41

5.1.2	Constants	42
5.1.2.1	String Constants	42
5.1.2.2	Boolean Constants	42
5.1.2.3	Numeric Constants.....	42
5.1.3	Operators	43
5.1.4	Special Characters	43
5.1.5	Comments	44
5.1.6	Operator Precedence.....	44
5.2	Data Types and Type Specifiers	45
5.2.1	Data Types.....	45
5.2.2	Type Specifiers	46
5.2.3	Use in CAST	46
5.2.4	Use in Column Definitions.....	46
5.2.5	System Metadata	47
5.2.6	Internal Precision	47
5.3	Value Expressions.....	47
5.3.1	Column References	48
5.3.2	Operator Invocation.....	48
5.3.3	Function Call.....	48
5.3.4	Aggregate Expression.....	48
5.3.5	Type Cast.....	49
5.3.6	Decimal Expressions and Precision	49
5.3.7	Scalar Subquery.....	50
5.3.8	Expression Evaluation Rules.....	51
6	Queries.....	51
6.1	Overview	51
6.2	Table Expressions.....	52
6.2.1	The FROM Clause	52
6.2.1.1	Joined Tables.....	53
6.2.1.1.1	CROSS JOIN	53
6.2.1.1.2	INNER JOIN.....	53
6.2.1.1.3	LEFT OUTER JOIN	53

6.2.1.1.4	RIGHT OUTER JOIN.....	53
6.2.1.1.5	ON Clause.....	53
6.2.1.1.6	USING Clause	54
6.2.1.2	Table and Column Aliases	55
6.2.1.3	Subqueries	57
6.2.2	WHERE Clause.....	57
6.2.3	GROUP BY and HAVING Clause	58
6.3	SELECT Lists	59
6.3.1	SELECT List Items.....	59
6.3.2	Column Labels.....	59
6.3.3	DISTINCT.....	60
6.4	Combining Queries (UNION, INTERSECT, EXCEPT).....	60
6.4.1	UNION	60
6.4.2	INTERSECT	61
6.4.3	EXCEPT	62
6.5	ORDER BY	63
6.6	LIMIT and OFFSET.....	64
6.7	WITH (Common Table Expressions).....	64
7	Functions and Operators	66
7.1	Logical Operators	66
7.2	Comparison Operators and BETWEEN.....	66
7.3	Mathematical Operators and Functions.....	67
7.4	String Functions and Operators	69
7.5	Pattern Matching – LIKE.....	72
7.6	Date/Time Functions.....	73
7.6.1	EXTRACT(from timestamp)	73
7.6.2	CURRENT_TIMESTAMP	74
7.6.3	NOW()	74
7.6.4	Interval Arithmetic.....	75
7.6.4.1	Interval Types.....	75
7.6.4.2	YEAR-MONTH interval:.....	76
7.6.4.3	DAY-TIME interval:.....	76

7.6.4.4	Support for Interval Arithmetic:	77
7.6.4.5	EXTRACT(from interval)	78
7.6.4.6	BETWEEN Operator:	79
7.7	CONDITIONAL EXPRESSIONS	79
7.7.1	CASE	79
7.7.2	COALESCE	80
7.7.3	IF	80
7.7.4	IFNULL	81
7.7.5	NULLIF	81
7.8	AGGREGATE FUNCTIONS	81
7.9	SUB-QUERY EXPRESSIONS	82
7.9.1	IN	82
7.9.2	NOT IN	83
7.9.3	EXISTS	83
7.10	Session Functions	84
7.10.1	CURRENT_USER	84
7.10.2	CURRENT_CATALOG	84
7.10.3	CURRENT_SCHEMA	85
7.11	VERSION()	85
8	QUERY EXECUTION	86
8.1	RapidsDB SQL Statement Execution	86
8.2	Partitioned Query Plans	86
8.3	Non-Partitioned Query Plans	88
8.4	Combination of Partitioned and Non-Partitioned Plans	90
8.5	RapidsDB Join Algorithms	92
9	INSERT	92
10	DDL	94
10.1	CREATE TABLE	94
10.2	Creating MOXE Tables	99
10.2.1	Partitioned Tables	99
10.2.2	Reference Tables	101
10.3	CREATE TABLE [AS] SELECT	101

10.3.1	Examples	102
10.3.2	Semantics	104
10.3.3	Exclusions	106
10.3.4	Error Conditions	106
10.4	CREATE INDEX	107
10.5	DROP TABLE	107
10.6	TRUNCATE TABLE	108
11	IMPORT/EXPORT Using IMPEX Connector	108
11.1	Overview	108
11.2	IMPEX Connector Type	110
11.3	Creating an IMPEX Connector	110
11.4	IMPEX Connector Properties	110
11.5	CSV (Delimited) File Formatting	115
11.5.1	Text Handling	115
11.5.1.1	ESCAPE SEQUENCES	115
11.5.1.2	Handling of Leading and Trailing Blanks	116
11.5.1.3	Empty Strings	117
11.5.2	Dates and Timestamps	118
11.5.3	Booleans	118
11.5.4	NULL Values	119
11.5.5	DELIMITER='<char> \t'	120
11.5.6	ENCLOSED_BY='<char> ' ""	121
11.5.7	ESCAPE_CHAR='<char>'	123
11.5.8	HEADER	123
11.5.9	CHARSET	124
11.5.10	TRAILING	124
11.6	IMPORT References	125
11.7	EXPORT References	128
11.8	Default IMPORT and EXPORT Connectors	130
11.8.1	Usage	130
11.8.2	Default Properties	130
11.8.3	Changing the IMPEX Properties for the “IMPORT” and “EXPORT” Connectors	130

11.9	IMPORT using SELECT and INSERT	131
11.9.1	IMPORT Table Expressions.....	131
11.9.2	IMPORT using a SELECT statement	132
11.9.2.1	Overview	132
11.9.2.2	Column Naming Using Default Column Names	133
11.9.2.3	Column Naming Using AS clause.....	133
11.9.2.4	Column Naming Using HEADER option	133
11.9.2.5	Column Data Typing Using GUESS Property	134
11.9.2.6	Column Data Typing Using AS clause	136
11.9.2.7	Column Skipping/Pruning Using AS Clause	137
11.9.2.8	Column Naming and Data Typing Using LIKE clause	137
11.9.2.9	RAW Data Format	138
11.9.2.10	SELECT FROM FILE.....	138
11.9.2.11	SELECT FROM FOLDER.....	142
11.9.2.12	INSERT ... SELECT	144
11.9.2.13	CREATE AS SELECT.....	146
11.10	Bulk IMPORT	149
11.10.1	Bulk IMPORT Using FILES Option	150
11.10.2	Bulk IMPORT Using FILES Option With FILTER	159
11.10.3	Bulk IMPORT Using FOLDERS Option	162
11.11	EXPORT Using SELECT	166
11.11.1	EXPORT Using SELECT TO a File.....	167
11.11.2	EXPORT Using SELECT TO a Folder	169
11.12	Bulk EXPORT	172
11.12.1	Backing Up Files/Sub-Folders When Doing a REPLACE	173
11.12.1.1	Backup for FILES option	173
11.12.1.2	Backup for FOLDERS option	174
11.12.2	Bulk EXPORT Using FILES Option.....	175
11.12.3	Bulk EXPORT Using FOLDERS Option	179
11.13	Error Handling.....	182
11.13.1	ERROR_PATH.....	182
11.13.2	ERROR_LIMIT	185

11.13.3	Data Conversion Errors	185
11.13.4	Mismatched Number of Fields and Columns on INSERT	188
11.13.5	Wildcard import to multiple connectors	189
12	REFRESH Command	190
13	SYSTEM METADATA TABLES	190
13.1	OVERVIEW	190
13.2	NODES Table	191
13.3	FEDERATIONS Table	192
13.4	CONNECTORS Table	193
13.5	CATALOGS Table	193
13.6	SCHEMAS Table	194
13.7	TABLES Table	195
13.8	INDEXES Table	195
13.9	COLUMNS Table	196
13.10	TABLE_PROVIDERS Table	198
13.11	AUTHENTICATORS Table	199
13.12	AUTHENTICATOR_CONFIG Table	200
13.13	USERS Table	200
13.14	USER_CONFIG Table	200
13.15	SESSIONS Table	201
13.16	USERNAME_MAPS Table	201
13.17	PATTERN_MAPS Table	202
13.18	QUERIES Table	202
14	Cancelling a Query	203
14.1	rapids-shell	203
14.2	JDBC	204
14.3	CANCEL QUERY command	204
15	Performance Tuning	206
15.1	EXPLAIN	206
15.2	JOIN Order	206
15.3	Restrict Amount of Data	207
16	Error Messages	208

16.1	RapidsDB shell Messages	208
16.2	Query Rejection Messages.....	208
16.3	Data Store-Related Messages	210
Appendix A	SQL Grammar	0

1 Overview

1.1 Changes

1.1.1 Changes from 4.3

- New IMPEX options have been added to allow for reading text files line by line, handling files with a trailing field separator and selective guessing of data types.
- The default setting for the IMPEX Connector GUESS property has been changed to false.
- Section 11 (IMPORT/EXPORT Using IMPEX Connector) has been rewritten to provide better explanations and examples
- The MOXE Connector now supports comments for columns and tables (introduced for other Connectors in release 4.3).
- The *VERSION()* function reports version information for RapidsDB.

1.1.2 Changes from 4.2.3.2

- Added support for IMPORT and EXPORT using the IMPEX Connector
- Minor corrections to some descriptions for string functions
- The MemSQL, MySQL, Oracle, Postgres, Greenplum and generic JDBC Connectors now support comments for columns and tables.
- The COLUMNS and TABLES system metadata tables have been updated to include a COMMENT column.

1.1.3 Changes from 4.2.3.1

- Added the REPEAT function to the section on String Operators and Functions
- Removed the character_length function from the section on String Operators and Functions, use the char_length function instead
- Added the MOD function to the section on Mathematical Operators and Functions.

1.1.4 Changes from 4.2.3

- The POW function was removed from the section on Mathematical Operators and Functions. Use the POWER function instead.
- The following changes were made to the section on String Operators and Functions:
 - The concat function was added
 - The “+” operator was added for string concatenation
 - The description for the “||” concatenation operator was corrected
 - The descriptions for the left and right functions were corrected

1.1.5 Changes from 4.2.2

- This is the GA release. There are some minor documentation updates but no new features

1.1.6 Changes from 4.2.1

- Updated the messages returned when cancelling a query
- Removed the DECODE() function which is not supported

1.1.7 Changes from 4.2

- The user can now create multiple MOXE Connectors and thereby have multiple MOXE schemas in the same RapidsDB Cluster.
- A MOXE Connector can now be configured to run on a subset of the nodes in a RapidsDB Cluster

1.1.8 Changes from 4.1

- The RapidsSE Connector is not supported for this release
- The DATE datatype is now supported
- A new MySQL Connector has been added
- The Hadoop Connector now supports the ORC format
- UNION, INTERSECT and EXCEPT are supported
- The user can cancel a running query from any of the supported interfaces
- Two new system metadata tables, QUERIES and QUERY_STATS have been added to track actively running queries

1.1.9 Changes from 4.03

- Licensing has been enabled
- User authentication has been enabled

1.1.10 Changes from R3.6

- Addition of a new, internal, integrated memory storage engine for RapidsDB named MOXE
- The UNION clause is not supported for this release
- The STATS command is not fully supported for this release

1.1.11 Changes from 3.4.2

- RapidsDB now requires users to be authenticated when connecting to the system.
- Added CREATE/DROP/ALTER commands for USERS and AUTHENTICATORS.
- RapidsDB supports user authentication via passwords and Kerberos.
- Added metadata tables to manage users and authenticators.
- Updated JDBC driver to support user authentication.
- A custom SSH port number can now be specified in the cluster configuration.

1.1.12 Changes from Release 3.4.1

- Updates to the description and examples for CREATE TABLE

1.1.13 Changes from Release 3.4

- Added support for TRUNCATE TABLE

1.1.14 Changes from Release 3.3.2

The following lists the major changes from Release 3.3.2:

- The MemSQL, Oracle, Postgres and Generic JDBC Connectors now support the SCHEMA_METADATA option which allows the user to dynamically update the set of tables being accessed by the Connector
- The user can now refresh the metadata for a single Connector using the “refresh connector” command
- The handling of the following options has been changed in the RapidsSE Loader and the Hadoop Connector:
 - ENCLOSED_BY:
 - The default has been changed to no enclosing characters
 - All data types can now be optionally enclosed
 - ENCLOSED_BY has been extended to allow for two characters, where the first character will define the start of the enclosing, and the second character will define the end of the enclosing
 - DATE_FORMAT – this is a new option to configure how the date portion of timestamps are formatted
 - The handling of NULL values has been updated
- A new version of the rapids-shell is now supported that uses the RapidsDB JDBC Driver for communicating with the RapidsDB Cluster. This new version can also be installed on any platform that supports Java
- CREATE AS SELECT – support has been added to allow the user to automatically create a table based on the result set of a query and to have that result set inserted into the table
- The performance of RapidsSE has been significantly improved for the both indexed and non-indexed access
- The performance of INSERT ... SELECT has been improved for the MemSQL, Oracle, Postgres and Generic JDBC Connectors

1.1.15 Changes from Release 3.3

The following lists the major changes from Release 3.3:

- Support for the following options when loading delimited data into RapidsSE:
 - DELIMITER
 - TERMINATOR
 - ENCLOSED_BY
 - ESCAPE_CHAR
- Support for the following options for the Hadoop Connector:
 - DELIMITER
 - TERMINATOR
 - ENCLOSED_BY
 - ESCAPE_CHAR
 - IGNORE_HEADER
- Support for Hive-style partitioning in the Hadoop Connector

- Changed the name for Hadoop Connector to Hadoop Connector to reflect the fact that the Connector will support other formats in the future.
- Added support for disabling and enabling Connectors

1.1.16 Changes from Release 3.1

The following list summarizes the major changes from Release 3.1:

- Addition of Connectors for Greenplum, Oracle and Postgres (refer to Section 3.2 for more details).
- JDBC Connector – the JDBC Connector is a new Connector that supports access to data sources that provide a JDBC Driver (refer to Sections 3.3 for more details).
- Case-insensitive naming - RapidsDB will now support case-insensitive naming across all of the Connectors (refer to Section 2 for more details).
- VIEWS supported – the Connectors for MemSQL, Greenplum, Oracle, Postgres, VoltDB, and the JDBC Connector, will include views along with tables as part of the metadata collected from the source system.
- RapidsSE:
 - Support for multi-column partitioning keys
 - Support for multi-column, non-unique indexes
 - Support for up to 8 indexes per table
 - Support for up to 8 columns per index
 - Support for range queries (lower and upper bound). The RapidsDB Execution Engine will now be able to request a range of column values from a RapidsSE table instead of having to request all of the rows and then filter the data in the RapidsDB Execution Engine. RapidsSE will make use of an index to satisfy the range query when there is an index defined on the column referenced in the range query (refer to section 3.5 for more details).
 - Support for exact key queries. This is a special case of range queries where only a single value is being requested from the RapidsDB Execution Engine (refer to section 3.5 for more details).
 - Column validation - RapidsSE will check that the data for each column matches the data type and precision specified for that column.
 - Support for LOAD and EXPORT commands from the rapids-shell or JDBC as ESCAPE commands.
- Hadoop Connector – added the following new features:
 - Multi-file Support – The user can use wildcard characters in the file name component for the file location such that when multiple files match the wildcard specification, all of the matching files will be accessed as a single table.
 - Dynamic Schema Metadata – Allows the user to dynamically change the schema being used by the Connector so that new tables can be defined, existing table definitions

updated, or existing table definitions can be removed, without requiring the user to drop and recreate the Connector with the new schema

- Index metadata - Connectors will now return information about indexes if they are supported by the underlying data store, such as MemSQL

1.2 What is RapidsDB?

RapidsDB is a fully parallel, distributed, in-memory federated query system that is designed to support complex analytical SQL queries running against a set of different data stores. Figure 1 below shows the major components of the RapidsDB system:

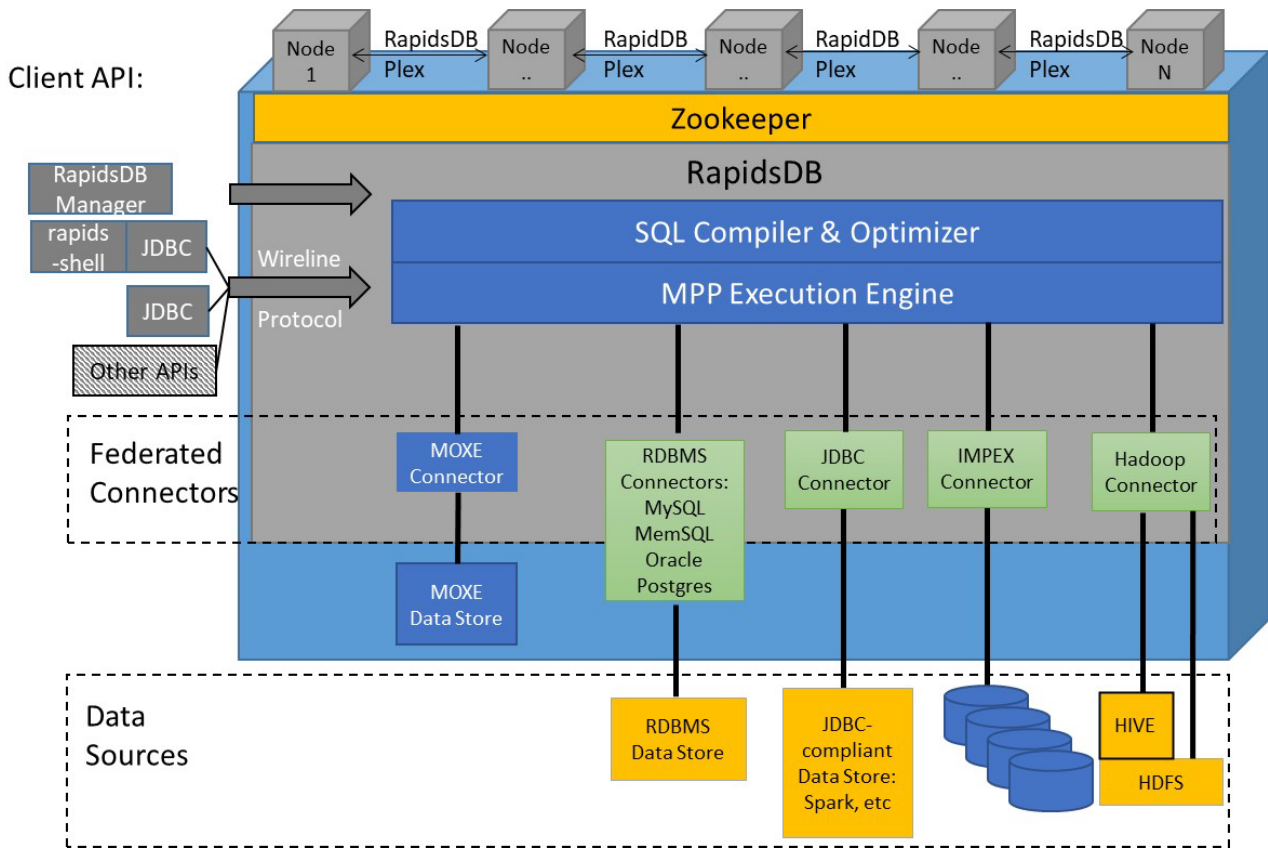


Figure 1. RapidsDB Architecture

RapidsDB provides unified SQL access to a wide variety of data sources, which can include relational and non-relational data sources. Data can be joined across all of the data sources. For this release, RapidsDB supports access to the following data sources:

- MOXE – internal in-memory data store for RapidsDB
- The following RDBMS data sources:
 - MySQL
 - MemSQL
 - Oracle
 - Postgres

- Greenplum
- Any data source that supports access via JDBC not including the RDBMS data sources above, for example, Spark
- IMPEX – provides access to csv formatted disk files
- Hadoop/HDFS delimited, ORC and Parquet files either through the Hive Metastore or directly against the files in HDFS

1.3 RapidsDB Components

1.3.1 RapidsDB Plex

The RapidsDB Plex is the internal communication fabric that is used between nodes in a RapidsDB cluster.

1.3.2 SQL Compiler and Optimizer

RapidsDB has an advanced, cost-based, SQL Compiler & Optimizer that is responsible for taking a user's SQL query and building the optimum query execution plan for that query. The query plan will then get passed to the Massively Parallel Processing (MPP) Execution Engine for execution of the query.

1.3.3 Massively Parallel Processing (MPP) Execution Engine

RapidsDB has its own fully parallel, MPP Execution Engine that is responsible for execution of the query plan generated by the RapidsDB SQL Compiler & Optimizer. The MPP Execution Engine is responsible for the execution of the query plans in concert with the Federated Connectors that provide the access to the underlying data sources. The Execution Engine is responsible for executing those parts of the query execution plan that cannot get pushed down to the underlying data source (see Federated Connectors below for details on query pushdown), and for delivering the final results of the query to user. For example, if the query involves a JOIN between a MOXE table and a Hive table, then the Execution Engine will perform the JOIN by getting the data from the MOXE and Hive tables using the MOXE and Hive Connectors.

1.3.4 Federated Connectors

The RapidsDB Federated Connectors are a set of dynamically, pluggable Connectors that control access to the underlying data stores that make up the federated database. The Connectors manage the metadata for the objects (typically tables or files) in the remote data store and present that metadata to the RapidsDB query execution engine as an ANSI-based SQL schema, thereby allowing the user to view the objects from the entire set of data sources as a single, federated SQL database.

The Connectors are responsible for managing data type conversion between the native data store and the RapidsDB query system which allows for the uniform handling of data types across all of the data stores. The Connectors present the data to the RapidsDB Execution Engine as rows and columns, which allows the data to be queried using standard ANSI SQL, and provides a uniform query interface regardless of the data source.

In order to optimize performance we generally want to have the underlying data source perform as much of the query as possible to reduce the amount of data that has to be processed by the RapidsDB

Execution Engine, rather than simply pulling all of the data from the data source and processing it within RapidsDB. For example, when the data source is a relational database such as Oracle, that data source is typically capable of executing a join on the tables in that data source, or it can filter the data based on predicates in the query.

RapidsDB supports an optimization feature called “Adaptive Query Pushdown” to deal with this. With Adaptive Query Pushdown, each Connector involved in the execution of a query plan analyzes the query plan and decides which parts of the query plan it can push down to the data source and for the remaining parts of the query the Connector will determine the optimum way to retrieve the data required to complete those parts of the query plan.

1.3.5 MOXE

MOXE (in-Memory Operational eXtreme Engine) is a parallel, fully distributed, internal in-memory data store that is co-resident with RapidsDB, sharing the same process space as the other RapidsDB components. MOXE uses hash partitioning to distribute the data across multiple partitions, and each partition operates in parallel when delivering data to the Execution Engine. The system can support multiple MOXE Connectors, each of which manages a single schema.

1.3.6 Client API

RapidsDB provides the following interfaces for accessing RapidsDB:

1.3.6.1 *RapidsDB Manager*

The RapidsDB Manager is a web-based management console for configuring and managing the RapidsDB cluster.

1.3.6.2 *rapids-shell*

The rapids-shell is a command line interface for configuring connectors and submitting queries. The rapids-shell uses the RapidsDB Unified JDBC Driver to communicate with the RapidsDB Cluster.

1.3.6.3 *JDBC*

RapidsDB supports the JDBC programmatic interface via the RapidsDB Unified JDBC Driver. The Unified RapidsDB JDBC Driver uses the RapidsDB Wireline Protocol (see below). Refer to the Unified JDBC Driver manual for more information.

1.3.6.4 *Wireline Protocol*

The RapidsDB Wireline Protocol is a platform-independent, Thrift-based, protocol that can be used for programmatically accessing RapidsDB from many different programming languages. It is a specification of the messages that flow between the client and server and sequencing of those messages. The Thrift-based API is a low-level interface that enables higher-level APIs to be developed to support almost any programming language. Refer to the RapidsDB Wireline Protocol Specification for more information.

1.3.7 Zookeeper

RapidsDB uses Zookeeper (version 3.4.6 or later) for configuration management across the RapidsDB cluster.

1.4 RapidsDB Cluster Topology

Figure 2 below shows the topology of a RapidsDB Cluster, in this example there are 5 nodes in the RapidsDB Cluster.:

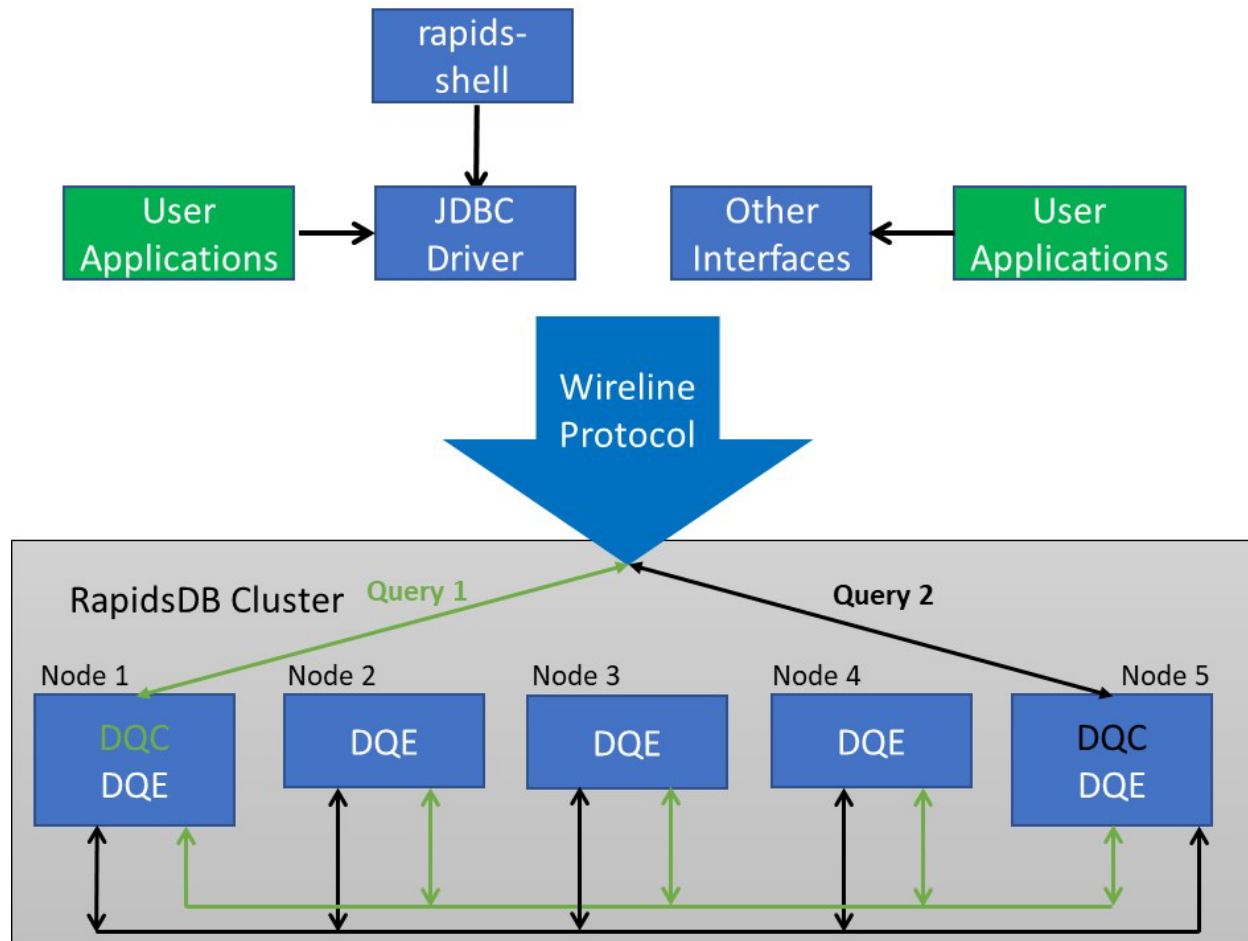


Figure 2. RapidsDB Cluster Topology

- rapids-shell – provides the command line interface for sending queries to the RapidsDB Cluster. The rapids-shell uses the RapidsDB JDBC Driver to communicate with the RapidsDB Cluster.
- User Applications can communicate with the RapidsDB Cluster using either the RapidsDB JDBC Driver or any other interfaces built on top of the RapidsDB Wireline Protocol.
- DQC (Distributed Query Coordinator) – queries can be submitted for execution over the Wireline Protocol to any node in the RapidsDB Cluster. The node where the query is submitted is called the DQC node, and this node is responsible for coordinating the execution of the query.

In the example above, two queries are submitted to the RapidsDB Cluster, Query 1 is submitted to Node 1, which becomes the DQC for that query, and Query 2 is submitted to Node 5, which becomes the DQC for that query.

In addition, when configuring the RapidsDB Cluster, one node must be configured as a DQC node, and this is the node where the RapidsDB cluster is installed from and where the RapidsDB cluster is stopped and started.

- DQE (Distributed Query Executor) – the nodes where the query execution is performed are called the DQE nodes. A DQE node will have a Connector running on it that is participating in the query. For example, if the query includes a MOXE table, then each node in the RapidsDB cluster where the MOXE Connector responsible for that table is running will participate in the query.

2 Federations, Connectors and Naming

2.1 Overview

In RapidsDB, a Federation is a logical grouping of a set of zero or more Connectors. Federations are named, and RapidsDB has a default Federation named “DEFAULTFED”. At this time, the DEFAULTFED Federation is the only Federation available. In a future release support will be provided for using multiple Federations.

2.2 Connectors

The Connectors manage the metadata for the objects (typically tables or files) in the remote data store and present that metadata to the RapidsDB query execution engine as an ANSI-based SQL schema, thereby allowing the user to view the objects from the entire set of data sources as a single, federated SQL database.

The Connectors are responsible for managing data type conversion between the native data store and the RapidsDB query system which allows for the uniform handling of data types across all of the data stores. The Connectors present the data to the RapidsDB Execution Engine as rows and columns, which allows the data to be queried using standard ANSI SQL, and provides a uniform query interface regardless of the data source.

In order to optimize performance we generally want to have the underlying data source perform as much of the query as possible to reduce the amount of data that has to be processed by the RapidsDB Execution Engine, rather than simply pulling all of the data from the data source and processing it within RapidsDB. For example, when the data source is a relational database such as Oracle, that data source is typically capable of executing a join on the tables in that data source, or it can filter the data based on predicates in the query.

RapidsDB supports an optimization feature called “Adaptive Query Pushdown” to deal with this. With Adaptive Query Pushdown, each Connector involved in the execution of a query plan analyzes the query plan and decides which parts of the query plan it can push down to the data source, and then only the query results from the pushed down query need to be streamed to the Execution Engine to complete the query execution. For the remaining parts of the query the Connector will determine the optimum

way to retrieve the data required to complete those parts of the query plan, and then stream that data to the Execution Engine.

2.3 Table Naming

All tables (and views) in RapidsDB are identified using an ANSI standard 3-part name of <catalog>.<schema>.<table>. The 3-part name has to be unique within a Federation, if the name is ambiguous an error will be returned when the name is referenced in a query. Each Connector is responsible for managing its own 3-part name space. The metadata for catalogs, schemas and tables can be stored in RapidsDB under an alternate name to prevent name conflicts (see section 2.9 below). The user can map any part of the 3-part name to a user-specified name using the “AS” clause as part of the Connector definition. Refer to section 2.9 for more information on mapping names. Where the underlying data source supports 3-part names, the Connector will use that 3-part name by default to identify each table. Where the underlying data source does not support a catalog and/or schema name, the Connector will provide the missing part(s) of the name. The table below shows the default assignment of catalog names and schema names for the different Connectors supported in this release:

Connector Type	Catalog Name	Schema Name
MOXE	Connector name	Connector name
MemSQL	Connector name	MemSQL database name (specified using the “DATABASE” option as part of the Connector definition)
MySQL	Connector name	MySQL database name
Oracle	Connector name	Oracle schema name
Postgres	Database name (specified using the “DATABASE” option as part of the Connector definition)	Postgres schema name
Greenplum	Database name (specified in JDBC connection url as part of the Connector definition)	Greenplum schema name
Hadoop	Connector name	PUBLIC
Hadoop with Hive Metastore	Connector name	Hive database name
JDBC	If the data source supports 3-part naming then this will be the data source catalog name,	If the data source supports 3-part naming then this will be the data source schema name, otherwise it will be the catalog or schema name

	otherwise it will be the Connector name	returned by the JDBC Driver for that data source, whichever is not NULL.
--	---	--

The table below shows some examples for a Postgres and Oracle Connector:

```
CREATE CONNECTOR PG1 TYPE POSTGRES WITH DATABASE= 'dw1', USER='adm', PASSWORD='admpsw'
NODE NODE1 SCHEMA PRODUCTION;
```

Catalog Name	Schema Name	Table Name
dw1	production	history
dw1	production	items
dw1	production	parts

```
CREATE CONNECTOR ORA TYPE ORACLE WITH USER='dba', PASSWORD='dba123', HOST='apollo',
SID='orcl' NODE NODE5;
```

Catalog Name	Schema Name	Table Name
orcl	customers	ORDERS
orcl	customers	CUSTOMER
orcl	customers	HISTORY

2.4 Retrieval and Storage of Schema Metadata

A Connector is responsible for retrieving the schema metadata (catalogs, schemas, tables, views) from the underlying data store that the Connector connects to, and for subsequently making that information available for use by the RapidsDB Query Planner. A Connector will save the schema metadata using the character case provided by the underlying data store. For example, by default Postgres returns all of the schema metadata information in lower case, whereas MemSQL will return the metadata for the table name in the case that was used when the table was created. The next section will describe how this schema metadata information is used.

The following table shows the schema metadata as stored by MemSQL and Postgres and the metadata stored by RapidsDB:

MEMSQL	RAPIDSDB
--------	----------

	DATABASE	TABLE	CATALOG	SCHEMA	TABLE
	WEST	customer_west	MEM1	WEST	customer_west
	WEST	ORDERS	MEM1	WEST	ORDERS
POSTGRES			RAPIDSDB		
DATABASE	SCHEMA	TABLE	CATALOG	SCHEMA	TABLE
sales	east	customer_east	sales	east	customer_east
sales	east	orders	sales	east	orders

As can be seen from the table above, the schema information is stored in RapidsDB in exactly the same case as the underlying data store, with “MEM1” being used as the Catalog name for MemSQL because MemSQL does not support catalogs, and that is the name of the MemSQL Connector in this example.

The user can check on the schema metadata by querying the RapidsDB System Metadata tables in the RAPIDS.SYSTEM schema:

- CATALOGS
- SCHEMAS
- TABLES
- COLUMNS

Below are some sample queries against the System Metadata tables for tables managed by MOXE:

```

rapids > select * from tables where schema_name='MOXE';
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME  IS_PARTITIONED  PROPERTIES
-----
MOXE          MOXE         SUPPLIER     true
MOXE          MOXE         ORDERS       true
MOXE          MOXE         LINEITEM     true
MOXE          MOXE         PARTSUPP     true
MOXE          MOXE         PART         true
MOXE          MOXE         REGION       false
MOXE          MOXE         CUSTOMER     true
MOXE          MOXE         NATION       false

8 row(s) returned (0.12 sec)
rapids > select column_name, data_type, is_partition_key, is_nullable from columns
> where schema_name='MOXE' and table_name='LINEITEM';
COLUMN_NAME  DATA_TYPE  IS_PARTITION_KEY  IS_NULLABLE
-----
L_ORDERKEY   INTEGER     true              false
L_PARTKEY    INTEGER     false             false
L_SUPPKEY    INTEGER     false             false
L_LINENUMBER INTEGER     false             false
L_QUANTITY   DECIMAL     false             false
L_EXTENDEDPRICE DECIMAL     false             false
L_DISCOUNT DECIMAL     false             false
L_TAX        DECIMAL     false             false
L_RETURNFLAG VARCHAR      false             false
L_LINESTATUS VARCHAR      false             false
L_SHIPDATE   TIMESTAMP   false             false
L_COMMITDATE TIMESTAMP     false             false
L_RECEIPTDATE TIMESTAMP     false             false
L_SHIPINSTRUCT VARCHAR      false             false
L_SHIPMODE   VARCHAR      false             false
L_COMMENT    VARCHAR      false             false

16 row(s) returned (0.12 sec)

```

2.5 Object Name Resolution and Case Sensitivity

When the user submits a query, the RapidsDB Query Planner will need to determine which Connector is responsible for each of the tables referenced in a SQL query. Each table name can optionally be qualified with a catalog and/or schema name. In order to determine which Connector is responsible for a table, the RapidsDB Query Planner will send a request to all of the Connectors with the name of the table, optionally qualified with the catalog and/or schema name, and each Connector will then look up the table name in its schema metadata information. By default the RapidsDB Query Planner follows SQL conventions, converting all table names to upper case before sending the request to the Connectors, unless any part of the name (catalog, schema, or table) is enclosed in identifier delimiters (back-ticks or double quotes), in which case that portion of the name will be sent using the case specified in the query. The table below shows some examples:

Original Query	Object Name Sent to Connectors for Resolution
Select * from customer;	CUSTOMER
Select * from west.customer;	WEST.CUSTOMER
Select * from "customer";	customer
Select * from "west"."customer";	west.customer

2.6 Connector Lookup of Object Names (Default)

The following describes the default way that Connectors will do lookup of the object names sent from the Query Planner. Each Connector maintains its own metadata information and controls the matching of names used in queries to names in the underlying data store. What the following will show is that **the user can specify the object names as case-insensitive names, even when the underlying data store, such as MemSQL, is case-sensitive.**

By default, each active Connector will do a case-insensitive lookup of the object name provided by the Query Planner, informing the Query Planner if it has a match for that object name. In the event that there are two or more matches for the object name, then an error will be returned to the user indicating that the object name is ambiguous. Assuming that the object name is unique, when the Connector builds the query to be submitted to the back-end data store, it will use the case as seen in the object metadata retrieved by the Connector (see Retrieval and Storage of Schema Metadata above). This means that for the vast majority of queries, the RapidsDB user can specify object names without regard for case. The Connector will ensure that the case used for the object names will match what the underlying data store expects. The only time that this would be a problem is when the underlying data store uses case-sensitive names (eg MemSQL) and the user has used the same object name but with different cases (eg customer and CUSTOMER). For this situation an option, IGNORE_CASE, is provided which when set to FALSE will instruct the Connector to perform a case-sensitive lookup. Refer to section 2.7 below for more details on IGNORE_CASE.

The following examples show how this name resolution can be applied to two Connectors, MemSQL (which is case-sensitive), and Postgres (by default, the object metadata information is stored in lower case).

MEMSQL		RAPIDSDB	
DATABASE	TABLE	SCHEMA	TABLE
WEST	customer_west	WEST	customer_west
WEST	ORDERS	WEST	ORDERS
POSTGRES		RAPIDSDB	
SCHEMA	TABLE	SCHEMA	TABLE
east	customer_east	east	customer_east
east	orders	east	orders

As described in section 2.4 above, Connectors preserve the case of names retrieved from the underlying data store but allow case-insensitive matching of names.

The following examples will go through the name resolution process using the metadata above:

1. Select * from customer_west;

The Query Planner would ask the Connectors for the object name "CUSTOMER_WEST", and the Connectors would do a case-insensitive match on that name, and the MemSQL Connector would have a match.

The MemSQL Connector would then build (condense) the following query to be sent to MemSQL:

```
select * from WEST.customer_west;
```

2. Select * from CUSTOMER_WEST;

The Query Planner would ask the Connectors for the object name "CUSTOMER_WEST", and the Connectors would do a case-insensitive match on that name, and the MemSQL Connector would have a match.

The MemSQL Connector would then build (condense) the following query to be sent to MemSQL:

```
select * from WEST.customer_west;
```

Note that in this case the query as specified by the user had the table name in upper case, but when the query was sent to MemSQL the table name was converted to lower case to match what was provided by MemSQL when the schema metadata was retrieved.

3. Select * from WEST.CUSTOMER_WEST;

The Query Planner would ask the Connectors for the object name "WEST.CUSTOMER_WEST", and the Connectors would do a case-insensitive match on that name, and the MemSQL Connector would have a match.

The MemSQL Connector would then construct (condense) the following query to be sent to MemSQL using the correct case for any table names and doing any necessary conversion for SQL syntax differences:

```
select * from WEST.customer_west;
```

Note that in this case the query as specified by the user had the schema and table names in upper case, but when the query was sent to MemSQL the table name was converted to lower case to match what was provided by MemSQL when the schema metadata was retrieved.

4. `select * from orders;`

The Query Planner would ask the Connectors for the object name “ORDERS”, and the Connectors would do a case-insensitive match on that name, and both the MemSQL and Postgres Connectors would have a match, and so the query would be rejected with an ambiguous name error. The user would have to specify the schema name to disambiguate the query as shown in example 5 below.

5. `select * from west.orders;`

The Query Planner would ask the Connectors for the object name “WEST.ORDERS”, and the Connectors would do a case-insensitive match on that name, and the MemSQL Connector would have a match.

The MemSQL Connector would then construct (condense) the following query to be sent to MemSQL using the correct case for any table names and doing any necessary conversion for SQL syntax differences:

```
select * from WEST.orders;
```

Note that in this case the query as specified by the user had the schema and table names in lower case, but when the query was sent to MemSQL the schema name was converted to upper case to match what was provided by MemSQL when the schema metadata was retrieved.

What is important to note is that the user can specify the object names as **case-insensitive** names, even when the underlying data store, such as MemSQL, is case-sensitive.

The following table shows another set of table names as stored by MemSQL and the names as stored by MemSQL Connector:

MEMSQL		RAPIDSDB	
DATABASE	TABLE	SCHEMA	TABLE
WEST	customer_west	WEST	customer_west
WEST	CUSTOMER_WEST	WEST	CUSTOMER_WEST

In this example, we have the same table name but they are specified using different cases.

The following examples will go through the name resolution process using the metadata above:

1. Select * from CUSTOMER_WEST;

The Query Planner would ask the Connectors for the object name “CUSTOMER_WEST”, and the Connectors would do a case-insensitive match on that name, and the MemSQL Connector would now have two matches, and so the query would be rejected with an ambiguous name error.

2. Select * from “customer_west”;

The Query Planner would ask the Connectors for the object name “customer_west”, and the Connectors would do a case-insensitive match on that name, and the MemSQL Connector would still have two matches, and so the query would be rejected with an ambiguous name error. Even when the user specified a quoted name, because the Connector did a case-insensitive lookup, both names matched.

The MemSQL and JDBC Connectors provide an option to handle this rare situation, IGNORE_CASE, which is described in section 2.7 below.

2.7 Case-sensitive Lookups

In the very rare case that the underlying data store uses case-sensitive naming (eg MemSQL) **AND** the user has used the same name but with different cases for two tables in the same schema, then the IGNORE_CASE option can be set to FALSE to instruct the Connector to use **case-sensitive** matching of names provided by the Query Planner.

The following table shows the database/schema and table names as stored by MemSQL and Postgres, and the names as stored by RapidsDB:

MEMSQL		RAPIDSDB	
DATABASE	TABLE	SCHEMA	TABLE
WEST	customer_west	WEST	customer_west
WEST	CUSTOMER_WEST	WEST	CUSTOMER_WEST
WEST	orders	WEST	orders
POSTGRES		RAPIDSDB	
SCHEMA	TABLE	SCHEMA	TABLE
east	customer_east	east	customer_east

east	orders	east	orders
------	--------	------	--------

The following examples will go through the name resolution process using the metadata above, with the IGNORE_CASE option set to 'FALSE' for the MemSQL Connector:

1. Select * from customer_west;

In this case this might not result in what the user expected because the original query had the table name in lower case, but because the table name was not enclosed within double quotes, the Query Planner will convert the name to upper case. In order to access the table name "customer_west" the table name must be quoted and specified in lower case as in the example below.

In this example, the Query Planner would ask the Connectors for the object name "CUSTOMER_WEST", and the MemSQL Connector would do a case-sensitive match on that name, and the MemSQL Connector would match the MemSQL table named "CUSTOMER_WEST".

The MemSQL Connector would then construct (condense) the following query to be sent to MemSQL:

```
select * from WEST.CUSTOMER_WEST;
```

2. Select * from "customer_west";

The Query Planner would ask the Connectors for the object name "customer_west", and the MemSQL Connector would do a case-sensitive match on that name, and the MemSQL Connector would match the MemSQL table named "customer_west".

The MemSQL Connector would then build (condense) the following query to be sent to MemSQL:

```
select * from WEST.customer_west;
```

3. Select * from west.orders;

The Query Planner would ask the Connectors for the object name "WEST.ORDERS", and the MemSQL Connector would do a case-sensitive match on that name, and the MemSQL Connector would not find a match because both the schema name and table do no match (due to the case).

4. Select * from "WEST"."orders";

The Query Planner would ask the Connectors for the object name “WEST.orders”, and the MemSQL Connector would do a case-sensitive match on that name, and the MemSQL Connector would find a match for the table named “orders”.

The MemSQL Connector would then build (condense) the following query to be sent to MemSQL:

```
select * from WEST.orders;
```

It is highly recommended that all object names managed by a Connector with the IGNORE_CASE option set to false are specified as quoted names with the case of the object names set to match the case used by the underlying data store.

2.8 Handling Table Metadata

When configuring Connectors (refer to the Installation and Management Guide for more information), the user can specify which tables from the remote data source should be included for querying. By default, a Connector will include the metadata for all of the tables that can be accessed from the underlying data source. The user can restrict which tables will be included when configuring a Connector by explicitly listing the catalogs, schemas and/or table names to be made available (see Installation and Management Guide for more information).

2.9 Mapping Catalog, Schema and Table Names

When configuring a Connector, the user can map one or more parts of the 3-part name to local names that are used when querying through RapidsDB. The Connector maps these local names back to the original names used by the underlying data source. Figure 3 below shows two MemSQL Connectors, MEM1 and MEM2, each with the table “CUSTOMERS”, but in different schemas named “EAST” and “WEST”:

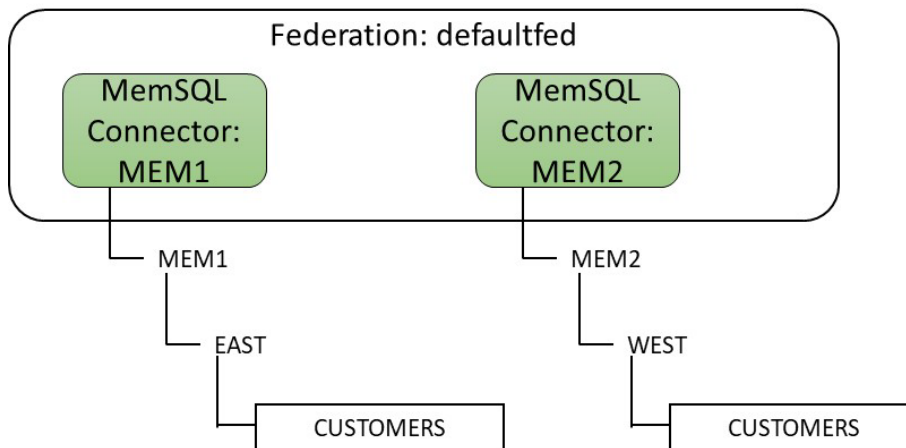


Figure 3. Mapping Catalog, Schema and Table Names

The query ‘SELECT * FROM customers’ would fail with an error indicating that the table name was ambiguous. To disambiguate the name the query would have to be changed to ‘SELECT * FROM east.customers’ to access the CUSTOMERS table managed by Connector MEM1.

When configuring the MemSQL Connectors the user could map the table named “EAST.CUSTOMERS” to “EAST_CUSTOMERS” and the table named “WEST.CUSTOMERS” to “WEST_CUSTOMERS”, and then the table names would be unique (for each Connector) and would not have to be qualified using the schema name, eg.

```
CREATE CONNECTOR MEMSQL1
  TYPE MEMSQL WITH host='192.168.1.1', port='3306', user='user1', database='EAST'
  NODE DB1
  CATALOG *
  SCHEMA *
  TABLE CUSTOMERS as EAST_CUSTOMERS;
```

```
SELECT * FROM east_customers WHERE ...
```

Refer to the Installation and Management Guide for details on how to do name mapping.

3 Operational Considerations for Connectors

Each of the RapidsDB Connectors exhibits different operational aspects which are described in the following sections.

3.1 MOXE Connector

The MOXE Connector operates as a multi-partitioned fully distributed Connector. When creating a MOXE Connector (see Installation and Management Guide for more information) the user can specify which nodes in the RapidsDB Cluster that Connector is to run on, the number of partitions used by MOXE on each node in the RapidsDB cluster, and the maximum amount of memory to be used on each node. MOXE supports both distributed and replicated tables. For distributed tables, each table managed by MOXE will be partitioned across all of the nodes on which the Connector is active and then distributed across all of the partitions on each node. The distribution of data across the partitions will be achieved by hashing the value of the partitioning column(s) specified for the table (as part of the CREATE TABLE – see section 10.1).

The user can create multiple MOXE Connectors on a system, each Connector will have its own catalog and schema which will match the Connector name.

When providing the data to Execution Engine for query processing, MOXE only passes pointers to the data to the Execution Engine, no physical copying of data occurs. The diagram below shows a two node RapidsDB cluster with a query being sent to Node 1 which acts as the query coordinator (DQC). The query results in all partitions of the table being scanned in parallel on both nodes and a set of row pointers (called row references) will be streamed to the Execution Engine on each node, and for each row reference the Execution Engine will ask for the value for the column O_ORDERPRIORITY, which can be accessed using the row reference, and the predicate filter on O_ORDERPRIORITY will be applied. If the O_ORDERPRIORITY value satisfies the predicate, then the Execution Engine will request the values

for the O_ORDERKEY and O_ORDERSTATUS columns for that row and then the results from Node 2 will be pipelined to Node 1 where the data from Node 2 will be merged with the data from Node 1 and the final result set will be returned to the client:

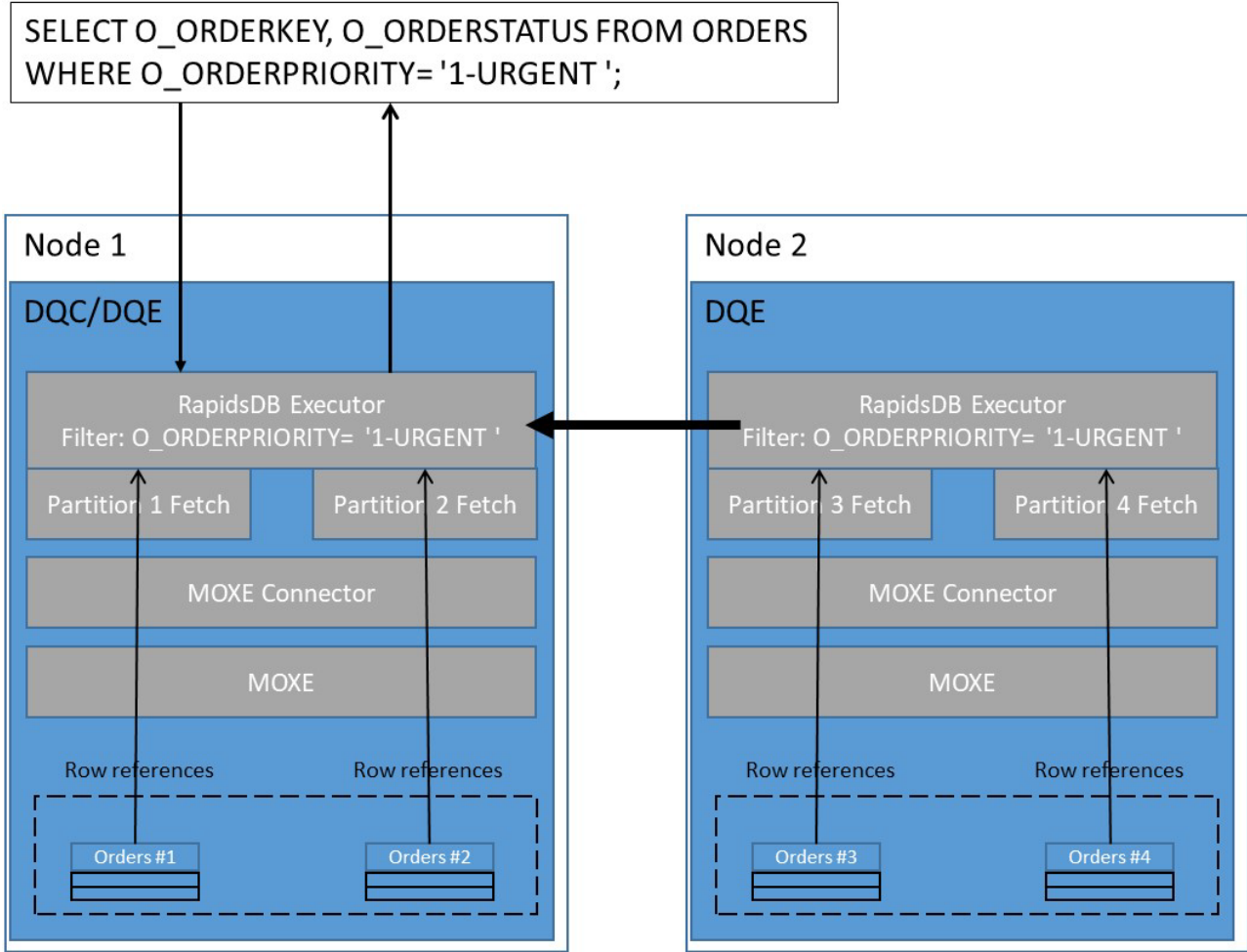


Figure 4. SELECT query processing with MOXE

For replicated tables, a full copy of each table will be replicated to each node in the RapidsDB cluster. Replicated Tables are typically used for storing smaller dimension tables so that when doing a JOIN with a larger table, the JOIN can be completed on each node without having to send any data from the replicated table across the network. In the example below, the NATION table is replicated and so the join between the NATION and SUPPLIER tables will be executed in parallel on both nodes, and the results from Node 2 will be pipelined to Node 1 where the data from Node 2 will be merged with the data from Node 1 and the final result set will be returned to the client:

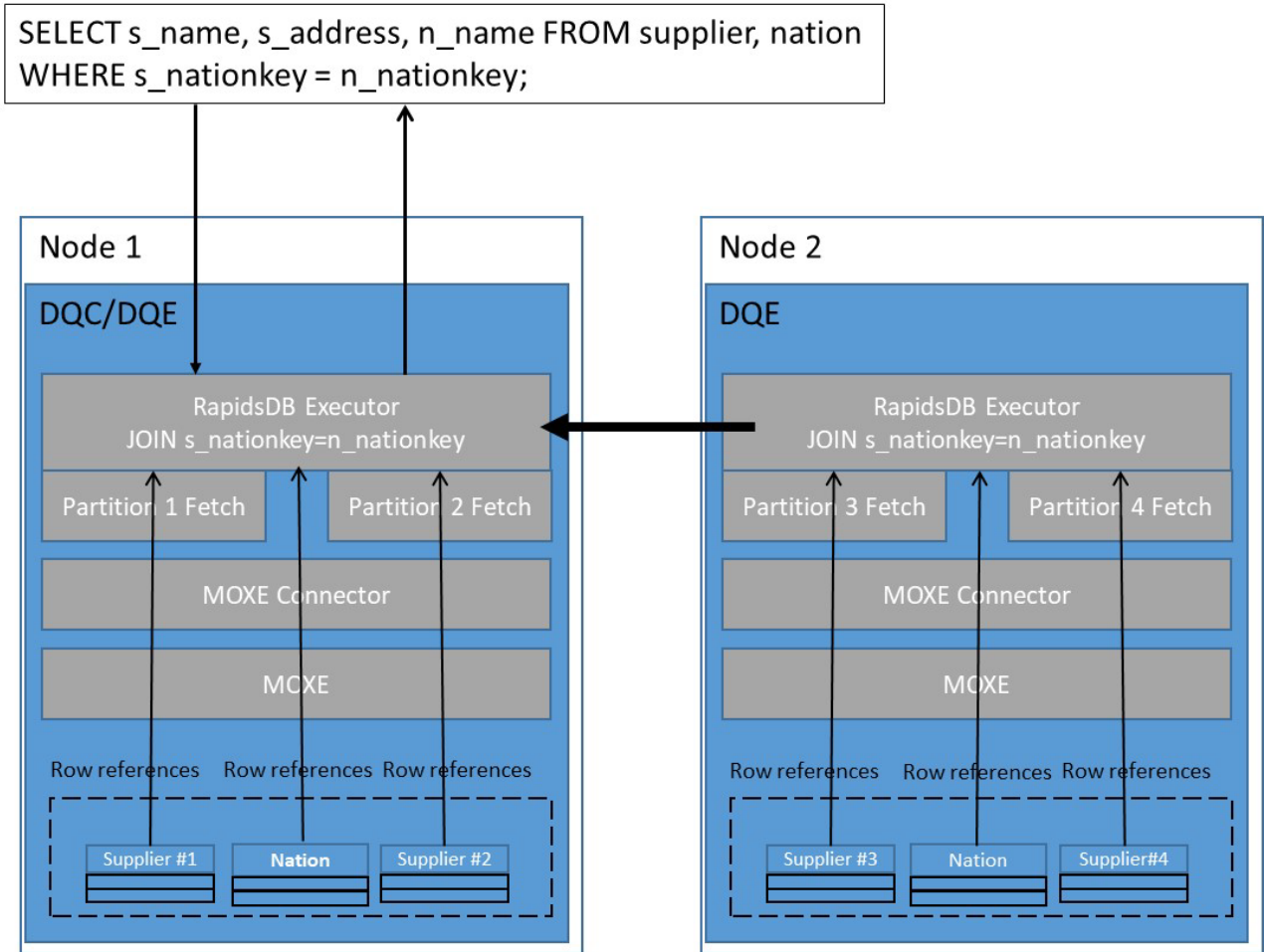


Figure 5. SELECT query processing with MOXE Replicated Table

MOXE also provides support for backing up either a complete database or individual tables to disk and then the user can subsequently restore the backups. (refer to the Installation and Management Guide for more information on configuring MOXE).

3.2 RDBMS Connectors

The RDBMS (MySQL, MemSQL, Oracle, Postgres and Greenplum) Connectors operate as non-partitioned, single node Connectors. This means that access to the underlying RDBMS will be from a single node in the RapidsDB Cluster. The user can specify multiple nodes for an RDBMS Connector when configuring the Connector, and RapidsDB will then use the node to access the underlying RDBMS database that would minimize movement of data over the network. Where possible, the user's query will be pushed down to the underlying RDBMS database and only the result of the query will get returned, and in this case the query execution will take advantage of any parallelism in the underlying RDBMS database. In the event that all or part of a query cannot be pushed down (for example, a federated query involving two or more data sources), then the data will be fetched from the RDBMS database by the Connector, and pipelined to the RapidsDB Execution Engine for further processing. Even in this case, any predicates will get applied to the data being fetched, to minimize the amount of

data being retrieved. The data pipelining allows the query to proceed as soon as the data starts to arrive from the data source, and in conjunction with the buffering performed by the Connector when retrieving the data, the net result is that only a small subset of the entire dataset being retrieved will be in memory at any one time. Figure 6 below shows a MemSQL Connector running on the same node as the MemSQL Aggregator that the MemSQL Connector will be using.

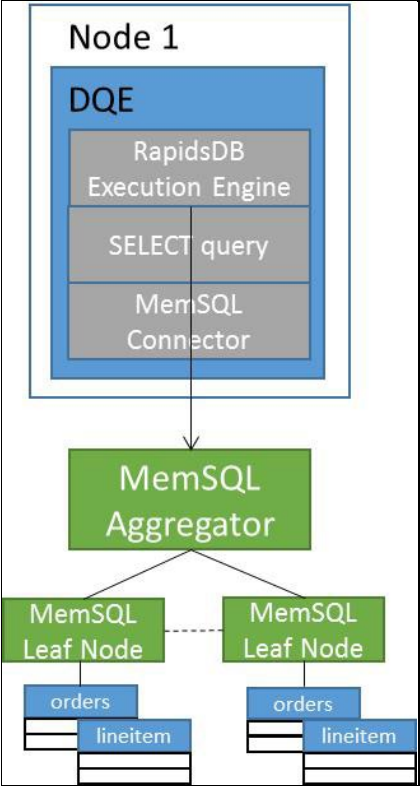


Figure 6. SELECT query processing with MemSQL Connector

3.3 Generic JDBC Connector

The JDBC Connector operates as a non-partitioned, single node Connector, in exactly the same way as the RDBMS Connectors (see 3.2). Refer to section Installation and Management Guide for more information on configuring a JDBC Connector.

3.4 Hadoop Connector

3.4.1 Partitioning

The Hadoop Connector operates as a multi-partitioned, fully distributed Connector. The user can specify which nodes in the RapidsDB cluster the Hadoop Connector is to run on, and how many partitions are supported per node. The partitioning with the Hadoop Connector is logical partitioning with all of the

partitioning being done within the Connector, not at the physical file level. This means that the user can change the partitioning, such as changing the number of partitions per node, without having to change the physical data in HDFS.

3.4.1.1 Delimited Files

When reading data from an HDFS delimited file, the Hadoop Connector will allocate a portion of the HDFS file to each node assigned to that Connector, and then on each node where Hadoop Connector is running the data will be split evenly across a set of partition readers so that the data will be read in parallel across the RapidsDB cluster. If there are n nodes assigned to a Hadoop Connector and m partitions allocated per node, then there will be n*m parallel reads against the HDFS file. The data being read is also buffered, with the records being pipelined to the RapidsDB Execution Engine for filtering and other processing. This means that it is possible for the Hadoop Connector to read files that are larger than the available memory on the nodes in the RapidsDB Cluster. Figure 7 below shows a two-node Hadoop Connector with m partitions per node:

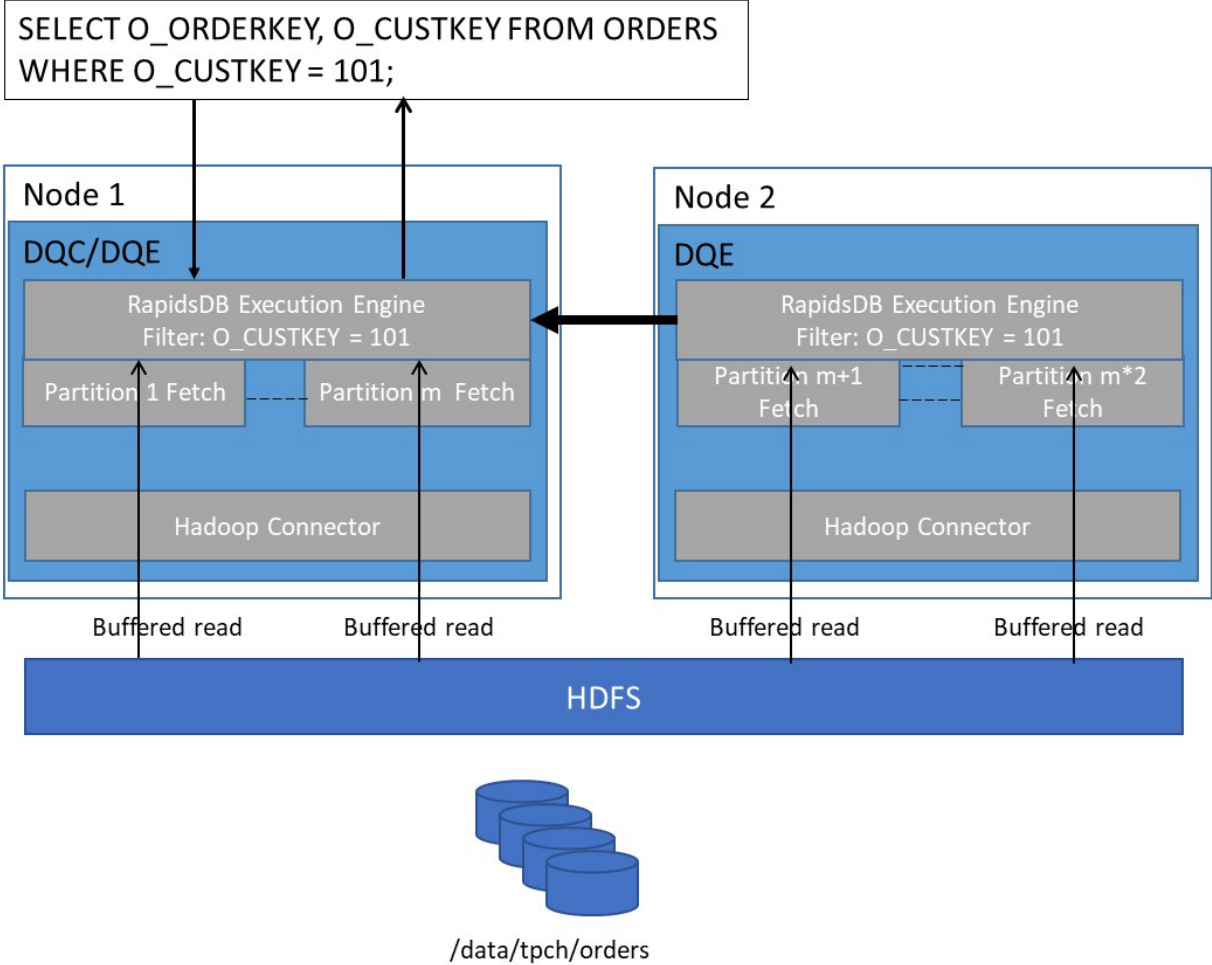


Figure 7. SELECT query processing with Hadoop Connector with delimited files

In this example, all of the columns from the data files in `/data/orders/tpch` directory will be returned to the Execution Engine, which will then apply the predicate `O_CUSTKEY=101` to filter the data and then only return the requested columns to the client.

When writing data to delimited files, if the data is being written in parallel, then the data being written will be distributed round-robin over all of the partitions and then written out to the target files (refer to the Installation and Management Guide for more information on configuring the Hadoop Connector).

3.4.1.2 ORC and Parquet Files

When reading ORC and Parquet files, the Hadoop Connector will calculate the number of HDFS blocks across all of the files to be read, and then divide up the blocks across the partition readers on each node where the Hadoop Connector is running. For example, if there are 10 files to be read and each file has 3 HDFS blocks, then there are 30 blocks to be read and if the Hadoop Connector was running on 6 nodes with 4 partitions per node, then the 30 blocks would be split across the 24 partition readers.

When reading ORC and Parquet files only the columns required to process the query from the associated table will be read. In addition, any column predicates will get used to restrict the data being read from the Parquet files. The example below illustrates this:

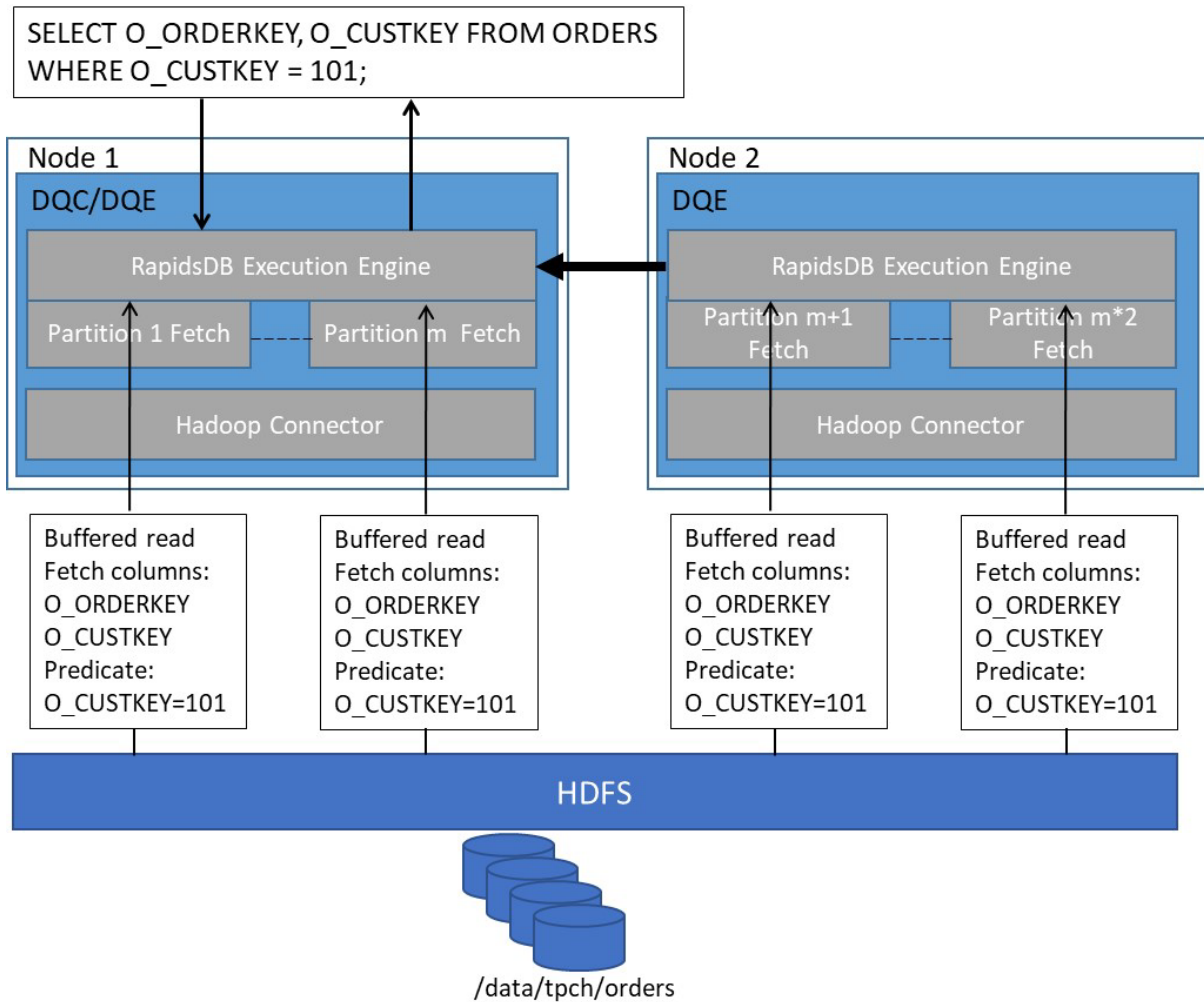


Figure 8. SELECT query processing with Hadoop Connector with Parquet files

In this example, only the data for the columns O_ORDERKEY and O_CUSTKEY will be read from the Parquet file(s) in the directory /data/tpch/orders, and only for those rows where the predicate O_CUSTKEY=101 is satisfied.

3.4.2 Hive-style Partitioning

The Hadoop Connector also supports Hive-style partitioning where the data stored in HDFS is arranged in directories where the directory names match the values for columns in the table. For example, in the following HDFS file structure below, the data is partitioned over the columns “region” and “country”, and so the files under /data/user/region=North America/country=US would match with a region of “North America” and country of “US”.

```
/data/user/region=North America/country=US
/data/user/region=North America/country=CA
/data/user/region=South America/country=BR
/data/user/region=South America/country=ME
```

When a query of the form `SELECT <column list> FROM <table> WHERE REGION='North America' AND COUNTRY='US'`; is submitted, the Hadoop Connector will use the predicate `"REGION='North America' AND COUNTRY='US' "` to restrict the files to be read to those files in the directory `/data/user/region=North America/country=US`. Refer to the Installation and Management Guide for more information.

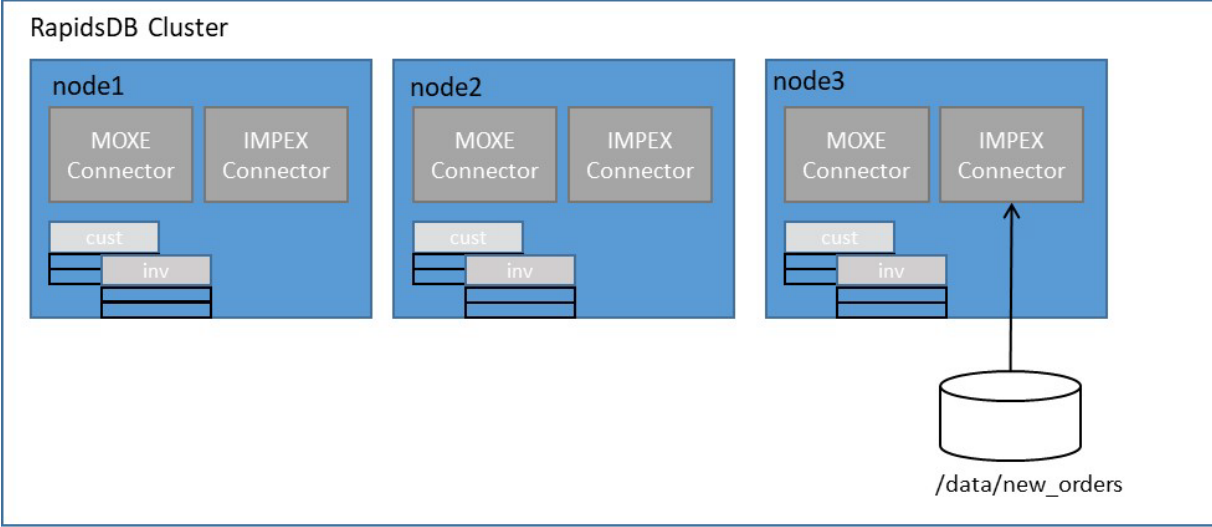
3.4.3 Writing Data to HDFS

The Hadoop Connector also supports writing the results of an `INSERT` or `INSERT ... SELECT` statement to HDFS. Refer to the Installation and Management Guide for more information.

3.5 IMPEX Connector

The IMPEX Connector is a new style of Connector that was introduced in Release 4.3. An IMPEX Connector can import and export data directly to and from disk files and also supports the ability to treat disk files as regular tables which can participate in federated queries (ie. in `SELECT` or `INSERT` queries). The implication of this is that the user does not need to go through an ETL process in order to load the data from the files into regular tables, such as MOXE tables. Instead, the files can be queried directly from the disk. For Release 4.3, an IMPEX Connector can read csv (delimited) files from any node in the RapidsDB Cluster, in future releases other file systems such as Amazon S3, Google Cloud and HDFS will be supported along with other file formats such as Parquet and ORC. After any data has been written to disk (in a supported format, ie csv for Release 4.3) it is available for querying. If needed, the user can also use an IMPEX Connector to load all or a subset of the data into regular tables, such as MOXE tables or other federated data sources such as Oracle, Postgres or MySQL. When reading the data from disk, an IMPEX Connector supports both column pruning and predicate pushdown so that only the data that is needed for the query is passed to the RapidsDB Execution Engine thereby allowing very large data files to be processed by the RapidsDB Engine where the size of the data can exceed the memory of the system. When reading the disk files the user does not need to define a schema for the table, an IMPEX Connector can estimate the data type for each field in the data by reading a sample of the data and the imputing the data type based on the actual data. This means that users can do fast exploration of data files without having to first assign a schema for the table. For example, by using a `LIMIT` clause the user can quickly look at a subset of the data and then can use other SQL predicates to do more sophisticated analysis of the data. If the schema for a file (or set of files) is known, then the user can provide that schema to the IMPEX Connector as part of the query.

An IMPEX Connector also supports the capability to write query results to files. Finally, an IMPEX Connector supports bulk import to allow for the rapid loading of data from disk files into any federated tables, and bulk export to allow for the rapid writing of the contents of any federated tables to disk files. Bulk `EXPORT` provides the ability for the user to take a snapshot of the federated database, and bulk `IMPORT` provides the ability to reload that snapshot.



```
SELECT ... FROM (FOLDER 'node://node3/data/new_orders'), MOXE.CUST, MOXE.INV ....;
```

In this example the IMPEX Connector is reading data from folder “/data/new_orders” on RapidsDB node “node3” and that data is then getting joined with data from two MOXE tables, “cust” and “inv”.

4 Query Interfaces

4.1 RapidsDB Command Line Interface (rapids-shell)

The rapids-shell is the command line interface to RapidDB and is started up using the rapids-shell.sh shell script file on Linux or the rapids-shell.bat file on Windows.

The user will be able to provide the following set of optional startup options:

Option	Default	Description
-h <ip address>	localhost	The ip address for the node in the RapidsDB Cluster to be used when sending commands to the RapidsDB Cluster
-p <port number>	4333	The port number for the RapidsDB JDBC Driver on the node to be used when sending commands to the RapidsDB Cluster
-s		Runs the rapids-shell in script mode
-t		Specifies that the data types for each column should be included in the csv result set

-k		Use Kerberos authentication instead of username and password.
----	--	---

4.1.1 Running the rapids-shell Locally

The rapids-shell can be run from any node in the RapidsDB Cluster, and is run from the RapidsDB cluster installation directory (eg /opt/rdp/current):

1. cd /opt/rdp/current
2. ./rapids-shell.sh

You will be prompted for a username and password before seeing the rapids prompt (see 4.1.3 below for more information on user authentication):

```
[rapids@boray05 current]$ ./rapids-shell.sh
Please enter a username > rapids
Please enter the password for user 'RAPIDS' >
rapids >
```

3. You should then be able to execute SQL queries or any of the supported rapids-shell command (refer to the Rapids-shell User Guide for more details). For example:

```
rapids > select * from catalogs;
CATALOG_NAME
-----
HADOOP
RAPIDS
rapidsse

3 row(s) returned (0.01 sec)
```

4.1.2 Running the rapids-shell Remotely

The rapids-shell can also be run remotely from any node that has TCP/IP connectivity to the RapidsDB Cluster. To install the rapids-shell on a remote system use the following steps:

1. Copy the rapids-shell-<version>.zip file from the 'shell' directory located in the RapidsDB installation directory on any node in the RapidsDB Cluster to the target system
2. Unzip the rapids-shell-<version>.zip file
3. The rapids-shell can then be started by running either the rapids-shell.sh file on Linux or the rapids-shell.bat file on Windows.
4. Refer to the Rapids Shell User Guide for more information.

4.1.3 Authentication of the rapids-shell

When the rapids-shell is started normally it will interactively ask for the username and password to be used for authentication. When entering the username interactively, rapids-shell will treat the name like a SQL object identifier by folding the name to uppercase unless it is surrounded by double quotes, in which case the case will be preserved. Care must be taken if case-sensitive usernames are used.

The password being entered will be treated as case sensitive and does not require any quoting.

To avoid being prompted to enter a username and password when invoking rapids-shell, simply define the following shell or environment variables when starting rapids-shell:

- RDP_USERNAME
- RDP_PASSWORD

Shell variables can be defined on the same line used to invoke the rapids-shell. They will only exist for the rapids-shell process.

Example 1:

```
$ RDP_USERNAME=rapids RDP_PASSWORD=rapids ./rapids-shell.sh
rapids > select current_user from tables limit 1;
?1
--
RAPIDS

1 row(s) returned (0.02 sec)
```

Alternatively, the variables can be exported to be environment variables before the rapids-shell is invoked.

Example 2:

```
$ export RDP_USERNAME=rapids
$ export RDP_PASSWORD=rapids
$ ./rapids-shell.sh
rapids > select current_user from tables limit 1;
?1
```



```
--
```

```
RAPIDS
```

```
1 row(s) returned (0.02 sec)
```

When defining the shell or environment variables for the username, please note that it will be treated the same as if it was entered interactively. i.e., unquoted usernames will be folded to uppercase. However, entering double quotes around a shell/environment variable is not as straight forward as it seems because the unix shell will first interpret the quotes and remove them before they are seen by the rapids-shell. As a result, the double quotes need to be escaped by single quotes that will be removed by the unix shell.

Example 3:

```
$ export RDP_USERNAME='"john"'      ← note outer single quotes and inner  
double quotes.
```

```
$ export RDP_PASSWORD=john
```

```
$ ./rapids-shell.sh
```

```
rapids > select current_user from tables limit 1;
```

```
?1
```

```
--
```

```
john                                ← note case sensitive name.
```

```
1 row(s) returned (0.02 sec)
```

Please refer to the Rapids-shell User Guide for more details.

To use Kerberos authentication with the RapidsDB shell, please refer to the Rapids-shell User Guide for details.

4.2 Programmatic Interfaces

4.2.1 JDBC

Refer to the RapidDB JDBC Driver manual for details on using the JDBC interface.

4.2.2 Invoking the rapids-shell Programmatically

The RapidsDB shell provides a scripting option to facilitate the submission of queries programmatically. Refer to the Rapids Shell User Guide for more information.

5 SQL Syntax

This section describes the SQL syntax supported by RapidsDB.

5.1 Lexical Structure

A SQL command is composed of a sequence of tokens, terminated by a semicolon (";"). Which tokens are valid depends on the syntax of the particular command.

A token can be a keyword, an identifier, a quoted identifier, a literal (or constant), or a special character symbol. Additionally, comments can occur in SQL input. Comments are not tokens, they are effectively equivalent to whitespace.

Here is an example of syntactically valid SQL command:

```
SELECT * FROM CUSTOMER WHERE CUSTOMER_NAME = 'Smith';
```

Refer to Appendix B for details of the SQL grammar supported by RapidsDB.

5.1.1 Identifiers and Keywords

Tokens such as SELECT or WHERE are examples of keywords, that is, words that have a fixed meaning in the SQL language. In the example query above, the tokens CUSTOMER and CUSTOMER_NAME are examples of identifiers. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called "names". Keywords and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a keyword without knowing the language.

SQL identifiers and keywords must begin with a letter (a-z, but also letters with diacritical marks and non-Latin letters) or an underscore (_). Subsequent characters in an identifier or keyword can be letters, underscores, or digits (0-9).

RapidsDB supports identifiers up to a maximum length of 32,000 characters, but underlying data systems may reject very long identifiers in CREATE TABLE statements.

By default, SQL keywords and identifiers are converted internally to uppercase. Therefore:

```
SELECT * FROM CUSTOMER WHERE CUSTOMER_NAME = 'Smith';
```

can equivalently be written as:

```
Select * FroM customer Where Customer_name = 'Smith';
```

A convention often used is to write keywords in upper case and names in lower case, e.g.:

```
SELECT * FROM customer WHERE customer_name = 'Smith';
```

There is a second kind of identifier: the delimited identifier or quoted identifier. It is formed by enclosing an arbitrary sequence of characters in back-ticks (`) or double-quotes ("). A delimited identifier is always recognized as an identifier, never a keyword. So "select" could be used to refer to a column or table named "select", whereas an unquoted select would be taken as a keyword and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

```
SELECT * FROM "CUSTOMER" WHERE "CUSTOMER_NAME" = 'Smith';
```

Quoted identifiers can contain any character, except the character with code zero. (To include a double quote, write two double quotes.) This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. Quoted identifiers are case sensitive.

5.1.2 Constants

There are three kinds of implicitly-typed constants: strings, booleans and numbers. Constants can also be specified with explicit types. These alternatives are discussed in the following subsections.

5.1.2.1 String Constants

A string constant is an arbitrary sequence of characters bounded by single quotes ('), for example 'This is a string'. To include a single-quote character within a string constant, write two adjacent single quotes, e.g., 'Dianne''s horse'. Note that this is not the same as a double-quote character (").

5.1.2.2 Boolean Constants

A boolean constant can either be the 4 character string true, with no enclosing quotes, or the 5 character string false, with no enclosing quotes.

e.g.

```
rapids > select true=true;
[1]
---
true
```

5.1.2.3 Numeric Constants

Numeric constants are accepted in these general forms:

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

where digits is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (e), if one is present.

There cannot be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

42
3.5
4.
.001
5e2
1.925e-3

A numeric constant that contains neither a decimal point nor an exponent has the data type INTEGER. A numeric constant containing a decimal point but no exponent has the data type DECIMAL. A numeric constant containing an exponent has the data type FLOAT. For more information on data types, see 5.2 below.

5.1.3 Operators

The following operators are supported:

'+'
'-'
'*'
'/'
'%'
'||'
'<'
'<='
'='
'!='
'<>'
'>='
'>'

5.1.4 Special Characters

Some characters that are not alphanumeric have a special meaning that is different from being an operator. Details on the usage can be found at the location where the respective syntax element is described. This section only exists to advise the existence and summarize the purposes of these characters.

Parentheses (()) have their usual meaning to group expressions and enforce precedence. In some cases parentheses are required as part of the fixed syntax of a particular SQL command.

Commas (,) are used in some syntactical constructs to separate the elements of a list.

The semicolon (;) terminates an SQL command. It cannot appear anywhere within a command, except within a string constant or delimited identifier.

The asterisk (*) is used in some contexts to denote all the columns of a table.

The period (.) is used in numeric constants, and to separate schema, table, and column names.

5.1.5 Comments

C-style block comments can be used where the comment begins with /* and extends to the matching occurrence of */. These block comments cannot nest.

A comment is removed from the input stream before further syntax analysis and is effectively replaced by whitespace.

Example:

```
SELECT /*Customer query */ * FROM customer /* Source table */ WHERE customer_name = 'Smith';
```

5.1.6 Operator Precedence

The table below shows the operator precedence rules:

Operator	Associativity	Description
.	Left	Table/column name separator
+ -	Right	Unary plus, unary minus
* / %	Left	Multiplication, division, modulo
+ -	Left	Addition, subtraction
BETWEEN IN LIKE		
< > = <= >= <>		Comparison operators
IS NULL, IS NOT NULL		
NOT	Right	
AND	Left	
OR	left	

5.2 Data Types and Type Specifiers

5.2.1 Data Types

Because the RapidsDB execution engine is implemented in the Java language and uses the Java Virtual Machine (JVM) for runtime support, RapidsDB data types are implemented internally as Java classes. Every data value handled by the system is an “instance” of a Java class. The Java class implements the behaviors and functionality for values of that class.

For simplicity and to facilitate type harmonization across different data systems, RapidsDB organizes these underlying Java classes into abstract “SQL types” that are similar in concept to the data types used in most SQL-based data management systems. A RapidsDB user will generally specify data types in terms of the SQL types, leaving the choice of Java class to the Connector and the execution engine. Connectors translate the user’s requested SQL types to equivalent types in the associated DBMS or data store. When presenting data to the execution engine, the Connector will select a suitable Java class (unless the user has explicitly specified a class to be used).

The RapidsDB runtime includes a library of Java classes that implement the standard behaviors and capabilities for SQL types, including RapidsDB-SQL standard functions (see section 7) and type conversions. The system can also be extended with other Java classes (“User Defined Types”) which offer extended or specialized capabilities. A User Defined Type may or may not correspond to any standard SQL type.

The table below shows the RapidsDB SQL types, along with the underlying Java class used by default in the execution engine and most Connectors. Note, however, that a given Connector may select a different Java class, for example to provide extended numeric precision or to handle data that doesn’t correspond to a SQL type.

SQL Type	Default internal Java class	Description
INTEGER	com.rapidsdata.stdlib.FastInteger	64-bit signed integer, nullable ¹
DECIMAL	com.rapidsdata.stdlib.FastDecimal ²	64-bit decimal (17 digits precision), nullable
FLOAT	com.rapidsdata.stdlib.FastFloat	64-bit IEEE floating point, nullable ¹
DATE	com.rapidsdata.stdlib.FastDate	64-bit date, range 0000-01-01 to 9999-12-31, nullable
TIMESTAMP	com.rapidsdata.stdlib.FastTimestamp	64-bit microsecond timestamp, nullable
BOOLEAN	com.rapidsdata.stdlib.FastBoolean	Boolean, nullable
VARCHAR	com.rapidsdata.stdlib.FastString ³	Up to 32k UTF-16 characters, nullable

Notes:

1. The FastInteger and FastFloat types reserve the lowest possible numeric value to represent NULL
2. Some Connectors may use java.math.BigDecimal
3. Some Connectors may use java.lang.String

5.2.2 Type Specifiers

A RapidsDB type specifier specifies a data type and desired precision. Type specifiers are used in the CAST operator (see 5.2.3) and also the column definitions in CREATE TABLE statements (see 10.1) and the USING clause of the CREATE CONNECTOR statement (see Installation and Management Guide).

The interpretation of size, precision and scale values in a RapidsDB type specifier depends on the data type. In this release the interpretations are as follows:

Type	Default interpretation of size / scale / precision
INTEGER	Precision in decimal digits (if unspecified: 17)
DECIMAL	Precision in decimal digits, scale in decimal digits (if unspecified: 17, 2)
FLOAT	Size of mantissa in binary digits (if unspecified: 53)
VARCHAR	Maximum length in characters (if unspecified: limited by Java class)

NOTES:

1. The value for precision of a FLOAT is interpreted as the number of binary digits in the mantissa. This is per ANSI SQL. A 53-bit mantissa corresponds to a standard 64-bit IEEE double precision floating point value. Expressed in decimal, the precision is 22 digits.
2. The type specifier may optionally be followed by a USING clause to explicitly specify a Java class to be used internally.

5.2.3 Use in CAST

In principle, CAST should precisely convert a value to the specified data type and precision or generate an error if this is not possible. So, for example, CAST(myColumn AS INTEGER(3)) should produce a value between -999 and +999 or generate an overflow exception if the value of myColumn lies outside that range. In practice, the behavior may deviate in certain cases, depending on whether the CAST operation is pushed down to the underlying data store (in particular, some data stores fail to generate overflow exceptions where expected).

5.2.4 Use in Column Definitions

When used in a column definition (CREATE TABLE or CREATE CONNECTOR WITH TABLE USING), a type specifier specifies the minimum requirement for a column. The underlying system may optionally substitute a definition of greater size or precision. It may not, however, substitute a definition of lesser size or precision. So, for example, INTEGER(4) specifies a column that can store values in the range -9999 to +9999 (and obeys the rules for integer math). An underlying data store may create the corresponding physical database column as, for example, a 16-bit integer.

If a column definition specifies a capability that exceeds the maximum capability of either the RapidsDB Execution Engine or an underlying Data Store, the definition will be rejected. For example, INTEGER(22) will be rejected by RapidsDB because the maximum precision for integers in RapidsDB is 19 decimal digits.

5.2.5 System Metadata

Column information in the COLUMNS table in the RapidsDB System Metadata (see 13.9) reflects the type, size, precision and scale of columns as reported by the underlying system and interpreted by RapidsDB. The information may differ from the definitions used to create the tables. The column data types are shown as standardized RapidsDB types. Size, precision and scale may exceed the values specified in a RapidsDB CREATE TABLE statement as noted above.

5.2.6 Internal Precision

The type specifiers affect storage, reading, writing and conversion of data values but do not control the precision of calculations on those values during query execution. Query calculations are performed by the RapidsDB Execution Engine (or the query engines of underlying Data Stores) with precision no less than the following:

- For integer values, 64-bit signed integers.
- For floating point values, 64-bit double precision (Java IEEE 754).
- For character values, Java String values of up to 2GB (using UTF-16 encoding).
- For binary values, Java byte arrays of up to 2GB in size.

5.3 Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the SELECT command or in search conditions. The result of a value expression is sometimes called a scalar, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called scalar expressions (or even simply expressions). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

A value expression is one of the following:

- A constant or literal value
- A column reference
- An operator invocation
- A function call
- An aggregate expression
- A type cast
- A scalar subquery
- Another value expression in parentheses (used to group subexpressions and override precedence)

In addition to this list, there are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in the appropriate location in Chapter 5. An example is the IS NULL clause.

We have already discussed constants in Section 4.1.2. The following sections discuss the remaining options.

5.3.1 Column References

A column can be referenced in the form:

```
qualifier.columnname
```

qualifier is the name of a table, or an alias for a table defined by means of a FROM clause. The qualifier and separating dot can be omitted if the column name is unique across all the tables being used in the current query.

5.3.2 Operator Invocation

There are three possible syntaxes for an operator invocation:

- expression operator expression (binary infix operator)
- operator expression (unary prefix operator)
- expression operator (unary postfix operator) where the operator token follows the syntax rules of Section 5.1.3, or is one of the keywords AND, OR, and NOT

5.3.3 Function Call

The syntax for a function call is the name of a function followed by its argument list enclosed in parentheses:

```
function_name ([expression [, expression ... ]])
```

For example, the following computes maximum value of column C1:

```
max(c1)
```

The list of built-in functions is in Section 5.

5.3.4 Aggregate Expression

An aggregate expression represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

```
aggregate_name (expression [, ... ] )  
aggregate_name (ALL expression [, ... ] )  
aggregate_name (DISTINCT expression [, ... ] )
```

where aggregate_name is a system-defined aggregate and expression is any value expression that does not itself contain an aggregate .

The first form of aggregate expression invokes the aggregate once for each input row. The second form is the same as the first, since ALL is the default. The third form invokes the aggregate once for each distinct value of the expression (or distinct set of values, for multiple expressions) found in the input rows.

Most aggregate functions ignore null inputs, so that rows in which one or more of the expression(s) yield null are discarded.

For example, count(*) yields the total number of input rows; count(f1) yields the number of input rows in which f1 is non-null, since count ignores nulls; and count(distinct f1) yields the number of distinct non-null values of f1.

5.3.5 Type Cast

A type cast specifies a conversion from one data type to another:

```
CAST ( expression AS type )
```

When a cast is applied to a value expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion operation has been defined. A cast applied to an unadorned string literal represents the initial assignment of a type to a literal constant value, and so it will succeed for any type (if the contents of the string literal are acceptable input syntax for the data type).

The table below shows the supported cast operations:

Source	Target							
	INTEGER	DECIMAL	FLOAT	STRING	TIMESTAMP	BOOLEAN	BINARY	NULL
INTEGER		Y	Y	Y	N	N	N	N
DECIMAL	Y	Y	Y	Y	N	N	N	N
FLOAT	Y	Y	Y	Y	N	N	N	N
STRING	Y	Y	Y	Y	Y	Y	N	N
TIMESTAMP	N	N	N	Y	Y	N	N	N
BOOLEAN	N	N	N	Y	N	Y	N	N
BINARY	N	N	N	N	N	N	Y	N
NULL	Y	Y	Y	Y	Y	Y	Y	Y

5.3.6 Decimal Expressions and Precision

Decimal expressions are mathematical expressions with a data type DECIMAL. Decimal values occur either because a column is defined as type DECIMAL or because a value is converted to DECIMAL using the CAST operator.

A decimal value has a precision, which is the total number of significant digits in the value, and a scale, which is the number of digits to the right of the decimal point.

When the RapidsDB Query Planner analyzes a decimal expression, it assumes a "canonical" precision and scale for each operator in the expression. The actual precision and scale depend on the Java class(es) involved in the calculations. The canonical precision and scale rules are designed to preserve sufficient decimal places to fully represent the possible result of the calculation. The following table summarizes the canonical precision assumptions. p_1 and s_1 represent the precision and scale of the first operand of a math operator; p_2 and s_2 represent the precision and scale of the second operand.

Operation	Canonical Precision and Scale
+ or -	Scale = $\max(s_1, s_2)$ Precision = $\max(p_1 - s_1, p_2 - s_2) + 1 + \text{scale}$
*	Scale = $s_1 + s_2$ Precision = $p_1 + p_2 + 1$
/	Scale = $\max(4, s_1 + p_2 + 1)$ Precision = $p_1 - s_1 + s_2 + \text{scale}$

Note that in all cases, the actual maximum precision of a decimal calculation depends on the underlying Java class. During execution of a query, if a calculation produces a result whose precision would exceed the maximum, the scale is typically reduced to preserve the integral part of the result.

The precision and scale of a decimal result can be specified explicitly using the CAST operator (see 5.3.5).

5.3.7 Scalar Subquery

A scalar subquery is an ordinary SELECT query in parentheses that returns exactly one row with one column. (See Chapter 6 for information about writing queries.) The SELECT query is executed and the single returned value is used as the expression result. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. (But if, during a particular execution, the subquery returns no rows, there is no error; the scalar result is taken to be NULL.) The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

For example, the following finds the largest city population in each state:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name) FROM states;
```

5.3.8 Expression Evaluation Rules

The query optimizer may significantly reorganize a query to improve performance. As a result, the order of evaluation of query expressions is not defined. Subqueries may be executed in any order or in parallel. Notably, the arguments of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

Furthermore, if the result of an expression can be determined by evaluating only some parts of it, then other subexpressions might not be evaluated at all. For instance, if one wrote:

```
SELECT true OR somefunc();
```

then `somefunc()` would (probably) not be called at all.

As a consequence, it is unwise to use functions with side effects as part of complex expressions. It is particularly dangerous to rely on side effects or evaluation order in `WHERE` and `HAVING` clauses, since those clauses are extensively reprocessed as part of developing an execution plan.

A common situation is trying to avoid division by zero in a `WHERE` clause. Attempting to check for a zero value first is not reliable:

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

A better solution is to use the `NULLIF` function (see section 7.7.5).

Note that `CASE` statements are also not guaranteed to execute in order. For example, a `CASE` cannot prevent evaluation of an aggregate expression contained within it, because aggregate expressions are computed before other expressions in a `SELECT` list or `HAVING` clause are considered. For example, the following query can cause a division-by-zero error despite seemingly having protected against it:

```
SELECT CASE WHEN min(employees) > 0
           THEN avg(expenses / employees)
           END
FROM departments;
```

The `min()` and `avg()` aggregates are computed concurrently over all the input rows, so if any row has `employees` equal to zero, the division-by-zero error will occur before there is any opportunity to test the result of `min()`. Instead, use a `WHERE` clause to prevent problematic input rows from reaching an aggregate function in the first place.

6 Queries

6.1 Overview

The general syntax of the `SELECT` command is

```
SELECT select_list FROM table_expression [sort_specification]
```

The following sections describe the details of the select list, the table expression, and the sort specification.

A simple kind of query has the form:

```
SELECT * FROM table1;
```

The select list specification `*` means all columns that the table expression happens to provide. A select list can also select a subset of the available columns or make calculations using the columns. For example, if `table1` has columns named `a`, `b`, and `c` you can make the following query:

```
SELECT a, b + c FROM table1;
```

(assuming that `b` and `c` are of a numerical data type). See Section 5.3 for more details.

`FROM table1` is a simple kind of table expression: it reads just one table. In general, table expressions can be complex constructs of base tables, joins, and subqueries. But you can also omit the table expression entirely and use the `SELECT` command as a calculator:

```
SELECT 3 * 4;
```

This is more useful if the expressions in the select list return varying results. For example, you could call a function this way:

```
SELECT round(123.99);
```

6.2 Table Expressions

A table expression computes a table. The table expression follows the `FROM` clause and is optionally followed by `WHERE`, `GROUP BY`, and `HAVING` clauses. Trivial table expressions simply refer to a table, a so-called base table, but more complex expressions can be used to modify or combine base tables in various ways.

The optional `WHERE`, `GROUP BY`, and `HAVING` clauses in the table expression specify a pipeline of successive transformations performed on the table derived in the `FROM` clause. All these transformations produce a virtual table that provides the rows that are passed to the select list to compute the output rows of the query.

6.2.1 The FROM Clause

The `FROM` Clause derives a table from one or more other tables given in a comma-separated table reference list.

```
FROM table_expression [, table_expression [, ...]]
```

A table expression can be a table name (optionally qualified by <catalog>.<schema> or <schema>), or a derived table such as a subquery, a JOIN construct, or complex combinations of these. If more than one table expression is listed in the FROM clause, the tables are cross-joined (that is, the Cartesian product of their rows is formed; see below). The result of the FROM list is an intermediate virtual table that can then be subject to transformations by the WHERE, GROUP BY, and HAVING clauses and is finally the result of the overall table expression.

6.2.1.1 Joined Tables

A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type. Inner, outer, and cross-joins are available. The general syntax of a joined table is

```
T1 join_type T2 [ join_condition ]
```

Joins of all types can be chained together, or nested: either or both T1 and T2 can be joined tables. Parentheses can be used around JOIN clauses to control the join order. In the absence of parentheses, JOIN clauses associate left-to-right.

The following describes the Join Types supported:

6.2.1.1.1 CROSS JOIN

T1 CROSS JOIN T2

For every possible combination of rows from T1 and T2 (i.e., a Cartesian product), the joined table will contain a row consisting of all columns in T1 followed by all columns in T2. If the tables have N and M rows respectively, the joined table will have N * M rows.

6.2.1.1.2 INNER JOIN

For each row R1 of T1, the joined table has a row for each row in T2 that satisfies the join condition with R1.

6.2.1.1.3 LEFT OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Thus, the joined table always has at least one row for each row in T1.

6.2.1.1.4 RIGHT OUTER JOIN

First, an inner join is performed. Then, for each row in T2 that does not satisfy the join condition with any row in T1, a joined row is added with null values in columns of T1. This is the converse of a left join: the result table will always have a row for each row in T2.

6.2.1.1.5 ON Clause

The ON clause is the most general kind of join condition: it takes a Boolean value expression of the same kind as is used in a WHERE clause. A pair of rows from T1 and T2 match if the ON expression evaluates to true.

6.2.1.1.6 USING Clause

The USING clause is a shorthand that allows you to take advantage of the specific situation where both sides of the join use the same name for the joining column(s). It takes a comma-separated list of the shared column names and forms a join condition that includes an equality comparison for each one. For example, joining T1 and T2 with USING (a, b) produces the join condition ON T1.a = T2.a AND T1.b = T2.b.

To put this together, assume we have tables t1, with columns num and name:

num	name
1	a
2	b
3	c

and t2 with columns num and value:

num	value
1	xxx
3	yyy
5	zzz

then we get the following results for the various joins:

```
rapids > select * from t1 inner join t2 using(num);
  NUM NAME      NUM VALUE
  ---  ---
   1 a          1 xxx
   3 c          3 yyy

2 row(s) returned (0.08 sec)
rapids > select * from t1 inner join t2 on t1.num=t2.num;
  NUM NAME      NUM VALUE
  ---  ---
   1 a          1 xxx
   3 c          3 yyy

2 row(s) returned (0.06 sec)
rapids > select * from t1 left join t2 on t1.num=t2.num;
  NUM NAME      NUM VALUE
  ---  ---
```

```

1 a          1 xxx
3 c          3 yyy
2 b          NULL NULL

3 row(s) returned (0.08 sec)
rapids > select * from t1 right join t2 on t1.num=t2.num;
  NUM NAME      NUM VALUE
  --- ----      -
  1 a          1 xxx
  3 c          3 yyy
  NULL NULL    5 zzz

3 row(s) returned (0.04 sec)

```

The join condition specified with ON can also contain conditions that do not relate directly to the join. This can prove useful for some queries but needs to be thought out carefully. For example:

```

rapids > select * from t1 left join t2 on t1.num = t2.num and t2.value = 'xxx';
  NUM NAME      NUM VALUE
  --- ----      -
  1 a          1 xxx
  2 b          NULL NULL
  3 c          NULL NULL

3 row(s) returned

```

Notice that placing the restriction in the WHERE clause produces a different result:

```

rapids > select * from t1 left join t2 on t1.num = t2.num where t2.value = 'xxx';
  NUM NAME      NUM VALUE
  --- ----      -
  1 a          1 xxx

1 row(s) returned

```

This is because a restriction placed in the ON clause is processed before the join, while a restriction placed in the WHERE clause is processed after the join. That does not matter with inner joins, but it matters a lot with outer joins.

6.2.1.2 Table and Column Aliases

A temporary name can be given to tables and complex table expressions to be used for references to the derived table in the rest of the query. This is called a table alias.

To create a table alias, write

```

FROM table_expression AS alias
Or
FROM table_expression alias

```

The AS keyword is optional. The alias can be any valid identifier.

A typical application of table aliases is to assign short identifiers to long table names to keep the join clauses readable. For example:

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num;
```

The alias becomes the new name of the table expression within the current query—the original name cannot be used elsewhere in the query. Thus, this is not valid:

```
SELECT * FROM my_table AS m WHERE my_table.a > 5; -- wrong
```

Table aliases are mainly for notational convenience, but it is necessary to use them when joining a table to itself, e.g.:

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
```

Parentheses are used to resolve ambiguities. In the following example, the first statement assigns the alias `b` to the second instance of `my_table`, but the second statement assigns the alias to the result of the join:

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
```

```
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

An extended form of table aliasing gives temporary names to the columns of the table, as well as the table itself:

```
FROM table_expression [AS] table_alias ( column_alias1 [, column_alias2 [, ...]] )
```

If fewer column aliases are specified than the number of columns in the table expression, the remaining columns are not renamed and will not participate in the query. This syntax is especially useful for self-joins or subqueries.

When an alias is applied to the output of a JOIN clause, the alias hides the original name(s) within the JOIN. For example:

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

is valid SQL, but:

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

is not valid; the table alias `a` is not visible outside the alias `c`.

NOTE: The alias name cannot be a reserved word unless it is enclosed in double quotes. For example, the following query will fail because the word “order” is a reserved word:

```
select cast(f_col1 as integer) as order from t1 where f_col2 > 480;
```

To use a reserved word as an identifier, enclose it in back-ticks:

```
select cast(f_col1 as integer) as `order` from t1 where f_col2 > 480;
```

6.2.1.3 Subqueries

Subqueries specifying a derived table must be enclosed in parentheses and may optionally be assigned a table alias name (as in Section 5.2.1.2). For example:

```
SELECT * FROM (SELECT * FROM table1) AS alias_name
```

This example is equivalent to `SELECT * FROM table1 AS alias_name`. More interesting cases, which cannot be reduced to a plain join, arise when the subquery involves grouping or aggregation.

For more information see Section 7.9.

6.2.2 WHERE Clause

The syntax of the WHERE Clause is

WHERE <search_condition>

where `search_condition` is any value expression (see Section 4.2) that returns a value of type boolean.

After the processing of the FROM clause is done, each row of the derived virtual table is checked against the search condition. If the result of the condition is true, the row is kept in the output table, otherwise (i.e., if the result is false or null) it is discarded.

Here are some examples of WHERE clauses:

```
SELECT ... FROM fdt WHERE c1 > 5
```

```
SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
```

```
SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100
```

```
SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

`fdt` is the table derived in the FROM clause. Rows that do not meet the search condition of the WHERE clause are eliminated from `fdt`. Notice the use of scalar subqueries as value expressions. Just like any other query, the subqueries can employ complex table expressions. Notice also how `fdt` is referenced in the subqueries. Qualifying `c1` as `fdt.c1` is only necessary if `c1` is also the name of a column in the derived

input table of the subquery. But qualifying the column name adds clarity even when it is not required. This example shows how the column naming scope of an outer query extends into its inner queries.

6.2.3 GROUP BY and HAVING Clause

After passing the WHERE filter, the derived input table might be subject to grouping, using the GROUP BY clause, and elimination of group rows using the HAVING clause.

```
SELECT select_list
FROM ...
[WHERE ...]
GROUP BY exprList
```

The GROUP BY Clause is used to group together those rows in a table that have the same values in all the columns listed. The order in which the columns are listed does not matter. The effect is to combine each set of rows having common values into one group row that represents all rows in the group. This is done to eliminate redundancy in the output and/or compute aggregates that apply to these groups.

If a table has been grouped using GROUP BY, but only certain groups are of interest, the HAVING clause can be used, much like a WHERE clause, to eliminate groups from the result. The syntax is:

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

Expressions in the HAVING clause can refer both to grouped expressions and to ungrouped expressions (which necessarily involve an aggregate function).

Examples:

```
SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
```

```
SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
```

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
FROM products p LEFT JOIN sales s USING (product_id)
WHERE s.date > '2015-06-01 00:00:00' AND s.date < '2015-07-01 00:00:00'
GROUP BY product_id, p.name, p.price, p.cost
HAVING sum(p.price * s.units) > 5000;
```

In the example above, the WHERE clause is selecting rows by a column that is not grouped (the expression is only true for sales during the month of June), while the HAVING clause restricts the output to groups with total gross sales over 5000. Note that the aggregate expressions do not necessarily need to be the same in all parts of the query.

6.3 SELECT Lists

As shown in the previous section, the table expression in the SELECT command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the select list. The select list determines which columns of the intermediate table are included in the result.

6.3.1 SELECT List Items

The simplest kind of select list is `*` which emits all columns that the table expression produces. Otherwise, a select list is a comma-separated list of value expressions (as defined in Section 4.2). For instance, it could be a list of column names:

```
SELECT a, b, c FROM ...
```

The column names `a`, `b`, and `c` are either the actual names of the columns of tables referenced in the FROM clause, or the aliases given to them as explained in Section 6.2.1.2. The name space available in the select list is the same as in the WHERE clause, unless grouping is used, in which case it is the same as in the HAVING clause.

If more than one table has a column of the same name, the table name must also be given, as in:

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

When working with multiple tables, it can also be useful to ask for all the columns of a particular table:

```
SELECT tbl1.*, tbl2.a FROM ...
```

(See also Section 5.2.2.)

If an arbitrary value expression is used in the select list, it conceptually adds a new virtual column to the returned table. The value expression is evaluated once for each result row, with the row's values substituted for any column references. But the expressions in the select list do not have to reference any columns in the table expression of the FROM clause; they can be constant arithmetic expressions, for instance.

6.3.2 Column Labels

The entries in the select list can be assigned names for subsequent processing, such as for use in an ORDER BY clause or for display by the client application. For example:

```
SELECT a AS value, b + c AS sum FROM ...
```

If no output column name is specified using AS, the system assigns a default column name. For simple column references, this is the name of the referenced column. For complex expressions, the system will generate a generic name.

```

rapids > select * from t1;
NUM NAME
---- ----
 1 a
 2 b
 3 c

3 row(s) returned (0.06 sec)
rapids > select num+100 as plus_100, name from t1;
PLUS_100 NAME
-----
 101 a
 102 b
 103 c

3 row(s) returned (0.06 sec)

```

6.3.3 DISTINCT

After the select list has been processed, the result table can optionally be subject to the elimination of duplicate rows. The DISTINCT keyword is written directly after SELECT to specify this:

```
SELECT DISTINCT select_list ...
```

(Instead of DISTINCT the keyword ALL can be used to specify the default behavior of retaining all rows.)

Obviously, two rows are considered distinct if they differ in at least one column value. Null values are considered equal in this comparison.

6.4 Combining Queries (UNION, INTERSECT, EXCEPT)

6.4.1 UNION

The results of two queries can be combined using the UNION set operation. The syntax is

```
query1 UNION [ALL] query2
```

query1 and query2 are queries that can use any of the features discussed up to this point. Set operations can also be nested and chained, for example

```
query1 UNION query2 UNION query3
```

which is executed as:

```
(query1 UNION query2) UNION query3
```

UNION effectively appends the result of query2 to the result of query1 (although there is no guarantee that this is the order in which the rows are actually returned). Furthermore, it eliminates duplicate rows from its result, in the same way as DISTINCT, unless UNION ALL is used.

In order to calculate the union of two queries, the two queries must be "union compatible", which means that they return the same number of columns and the corresponding columns have compatible data types. Also, any LIMIT or ORDER BY clause can only appear at the end of statement.

```
rapids > select * from t1;
NUM NAME
---- ----
 1 a
 2 b
 3 c

3 row(s) returned (0.05 sec)
rapids > select * from t3;
NUM VALUE
---- ----
 1 xxx
 2 yyy
 3 zzz
 1 xxx

4 row(s) returned (0.05 sec)
rapids > select * from t1 union select * from t3;
NUM NAME
---- ----
 1 a
 2 b
 3 c
 1 xxx
 2 yyy
 3 zzz

6 row(s) returned (0.13 sec)
```

```
rapids > select * from t1 union all select * from t3;
NUM NAME
---- ----
 1 a
 2 b
 3 c
 1 xxx
 2 yyy
 3 zzz
 1 xxx

7 row(s) returned (0.11 sec)
```

6.4.2 INTERSECT

INTERSECT returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operator.

The syntax is

```
query1 INTERSECT query2
```

query1 and query2 are queries that can use any of the features discussed up to this point. Set operations can also be nested and chained, for example

```
query1 INTERSECT query2 INTERSECT query3
```

which is executed as:

```
(query1 INTERSECT query2) INTERSECT query3
```

In order to calculate the intersect of two queries, the two queries must be "intersect compatible", which means that they return the same number of columns and the corresponding columns have compatible data types.

```
rapids > select * from t2;
NUM VALUE
----
1 xxx
2 yy
3 zz

3 row(s) returned (0.05 sec)
rapids > select * from t3;
NUM VALUE
----
1 xxx
2 yy
3 zz
1 xxx
4 aaa

5 row(s) returned (0.06 sec)
rapids > select * from t3 intersect select * from t2;
NUM VALUE
----
1 xxx
2 yy
3 zz

3 row(s) returned (0.07 sec)
```

6.4.3 EXCEPT

EXCEPT returns any distinct values from the query left of the EXCEPT operator. Those values return as long the right query doesn't return those values as well.

The syntax is

```
query1 EXCEPT query2
```

query1 and query2 are queries that can use any of the features discussed up to this point. Set operations can also be nested and chained, for example

```
query1 EXCEPT query2 EXCEPT query3
```

which is executed as:

```
(query1 EXCEPT query2) EXCEPT query3
```

In order to calculate the except of two queries, the two queries must be "except compatible", which means that they return the same number of columns and the corresponding columns have compatible data types. Also, any LIMIT or ORDER BY clause can only appear at the end of statement.

```
rapids > select * from t2;
NUM VALUE
---- -----
 1 xxx
 2 yy
 3 zz

3 row(s) returned (0.05 sec)
rapids > select * from t3;
NUM VALUE
---- -----
 1 xxx
 2 yy
 3 zz
 1 xxx
 4 aa

5 row(s) returned (0.06 sec)
rapids > select * from t3 except select * from t2;
NUM VALUE
---- -----
 4 aa

1 row(s) returned (0.08 sec)
```

6.5 ORDER BY

After a query has produced an output table (after the select list has been processed) it can optionally be sorted. If sorting is not chosen, the rows will be returned in an unspecified order. The actual order in that case will depend on the scan and join plan types, but it must not be relied on. A particular output ordering can only be guaranteed if the sort step is explicitly chosen.

The ORDER BY clause specifies the sort order:

```
SELECT select_list
  FROM table_expression
  ORDER BY orderByList [ASC | DESC]
```

The orderByList can be any expression that would be valid in the query's select list. An example is:

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

When more than one expression is specified, the later values are used to sort rows that are equal according to the earlier values. Each expression can be followed by an optional ASC or DESC keyword to

set the sort direction to ascending or descending. ASC order is the default. Ascending order puts smaller values first, where "smaller" is defined in terms of the < operator. Similarly, descending order is determined with the > operator.

Note that the ordering options are considered independently for each sort column. For example ORDER BY x, y DESC means ORDER BY x ASC, y DESC, which is not the same as ORDER BY x DESC, y DESC.

A sort_expression can also be the column label or number of an output column, as in:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
```

```
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

both of which sort by the first output column.

ORDER BY can be applied to the result of a UNION, INTERSECT, or EXCEPT combination, but in this case it is only permitted to sort by output column names or numbers, not by expressions.

6.6 LIMIT and OFFSET

LIMIT and OFFSET allow you to retrieve just a portion of the rows that are generated by the rest of the query:

```
SELECT select_list
  FROM table_expression
  [ ORDER BY ... ]
  [ LIMIT { number } ] [ OFFSET number ]
```

If a limit count is given, no more than that many rows will be returned (but possibly less, if the query itself yields less rows).

OFFSET says to skip that many rows before beginning to return rows. OFFSET 0 is the same as omitting the OFFSET clause. If both OFFSET and LIMIT appear, then OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

When using LIMIT or OFFSET, it is important to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows. You might be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? The ordering is unknown, unless you specified ORDER BY.

The rows skipped by an OFFSET clause still have to be computed inside the server; therefore a large OFFSET might be inefficient.

6.7 WITH (Common Table Expressions)

WITH provides a way to write auxiliary statements for use in a larger query. These statements are often referred to as Common Table Expressions or CTEs.

```
WITH common_table_expression [, common_table_expression ...] SELECT query;
common_table_expression:: table_name [(column list)] AS (SELECT query)
column_list:: column_name [,column_name ...]
```

Key characteristics of CTEs:

- RapidsDB supports only non-recursive CTEs.
- The column_list is optional, and when specified, the columns of CTE will be known by the names specified and the column names or aliases of the underlying query are not visible when referring to the CTE. In the example below, the column names in the CTE are x, y and z whereas the column names in the underlying query are a,b and c. (Note that the original names are still used normally within the underlying query itself, e.g. in the WHERE clause in the example below):

```
WITH cte_1(x, y, z) AS (SELECT a, b, c FROM t WHERE a < 5) SELECT x FROM cte_1;
```
- When the CTE is referenced in a SELECT statement, the CTE will be merged into the query and executed as part of the query. If there are multiple references to the same CTE, each reference to the CTE will be executed. In a future release, references to the CTE will be optimized to avoid unnecessary computation.
- If a CTE defined in the WITH clause is not referenced in the SELECT statement, it has no effect on the execution of the query.

Example:

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
    product,
    SUM(quantity) AS product_units,
    SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

7 Functions and Operators

7.1 Logical Operators

The usual logical operators are available:

AND

OR

NOT

SQL uses a three-valued logic system with true, false, and null, which represents "unknown". Observe the following truth tables:

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.

7.2 Comparison Operators and BETWEEN

The usual comparison operators are available:

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal

<> or !=	not equal
----------	-----------

Comparison operators are available for all relevant data types. All comparison operators are binary operators that return values of type Boolean.

In addition to the comparison operators, the special BETWEEN construct is available:

a BETWEEN x AND y

is equivalent to

a >= x AND a <= y

Notice that BETWEEN treats the endpoint values as included in the range. NOT BETWEEN does the opposite comparison:

a NOT BETWEEN x AND y

is equivalent to

a < x OR a > y

To check whether a value is or is not null, use the constructs:

expression IS NULL

expression IS NOT NULL

Do not write expression = NULL because NULL is not "equal to" NULL. (The null value represents an unknown value, and it is not known whether two unknown values are equal.) This behavior conforms to the SQL standard.

7.3 Mathematical Operators and Functions

Mathematical Operators

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division (integer division truncates the result)	4 / 2	2

%	modulo (remainder)	5 % 4	1
---	--------------------	-------	---

Mathematical Functions

Function	Return Type	Description	Example	Result
abs(x)	(same as input)	absolute value	abs(-17.4)	17.4
ceil(numeric)	integer	smallest integer not less than argument	ceil(-42.8)	-42
ceiling(numeric)	integer	smallest integer not less than argument (alias for ceil)	ceiling(-95.3)	-95
floor(numeric)	integer	largest integer not greater than argument	floor(-42.8)	-43
mod(numeric, numeric)	float	returns the remainder of the first argument divided by the second argument	mod(1.25, 0.5)	0.25
power(numeric, numeric)	float	raise first argument to the power of the second argument	power(9, 2)	81.0
round(numeric)	integer	round to nearest integer	round(123.99)	123

round(numeric, int)	float	round to int number of decimal places	round((123.999,2)	123.99
sqrt(numeric)	float	square root of argument	sqrt(10)	3.1622776985168457
stddev(expression)	float	historical alias for stddev_samp		
stddev_pop(expression)	float	population standard deviation of the input values		
stddev_samp(expression)	float	sample standard deviation of the input values		
variance(expression)	float	historical alias for var_samp		
var_pop(expression)	float	population variance of the input values (square of the population standard deviation)		
var_samp(expression)	float	sample variance of the input values (square of the sample standard deviation)		

7.4 String Functions and Operators

Function	Return Type	Description	Example	Result
concat(string, string)	text	String concatenation	'Post' 'greSQL'	'PostgreSQL'
concat(string, numeric) concat (numeric,string)	text	String concatenation	concat('Value: ', 3.1) concat(3.1, ' times')	'Value: 3.1' '3.1 times'
string string	text	String concatenation	'Post' 'greSQL'	'PostgreSQL'
string numeric	text	String concatenation	'Value: ' 3.1	'Value: 3.1'
string + string	text	String concatenation	'Post' + 'greSQL'	'PostgreSQL'
string + numeric	text	String concatenation	'Value: ' + 3.1	'Value: 3.1'
char_length(string)	int	Number of characters in string	char_length('jose')	4
lower(string)	text	Convert string to lower case	lower('TOM')	'tom'
position(substring in string)	int	Location of specified substring	position('om' in 'Thomas')	3
repeat(string, int)	text	Repeat the specified string for the specified number of times.	repeat('Post',2)	'PostPost'
substring(string from int [for int])	text	Extract substring starting at the "from" position, for	substring('Thomas' from 2 for 3)	'hom'

		the length specified by the "for" (defaults to rest of string)	substring('Thomas' from 2)	'homas'
substring(string from negative int [for int])	text	Extract substring starting at the "from" position counting backwards from the right of the string for the length specified by the "for" (defaults to rest of string)	Substring('Thomas' from -3 for 3)	'mas'
trim([leading trailing both] [character] from string)	text	Remove the longest string containing only the specified character (a space by default) from the start/end/both ends of the string	trim(both 'x' from 'xTomxx')	'Tom'
ltrim(string [,character])	text	Remove the longest string containing only the specified character (a space by default) from the start of the string	ltrim(' Tom') ltrim('aaTom','a')	'Tom' 'Tom'

<code>rtrim(string [,character])</code>	text	Remove the longest string containing only the specified character (a space by default) from the end of the string	<code>rtrim('Tom ')</code> <code>rtrim('Tomaa','a')</code>	'Tom' 'Tom'
<code>upper(string)</code>	text	Convert string to upper case	<code>upper('tom')</code>	'TOM'
<code>left(str text, n int)</code>	text	Return first n characters in the string. When n is negative an empty string will be returned.	<code>left('Tomas',2)</code>	'To'
<code>right(str text, n int)</code>	text	Return last n characters in the string. When n is negative an empty string will be returned.	<code>right('Tomas',2)</code>	'as'

7.5 Pattern Matching - LIKE

```
{string-expression} LIKE '{pattern}' [ESCAPE 'escape-character']
```

The LIKE expression returns true if the string-expression matches the supplied pattern. (As expected, the NOT LIKE expression returns false if LIKE returns true, and vice versa.)

If pattern does not contain percent signs or underscores, then the pattern only represents the string itself; in that case LIKE acts like the equals operator. An underscore (`_`) in pattern stands for (matches) any single character; a percent sign (`%`) matches any sequence of zero or more characters.

Some examples:

```
'abc' LIKE 'abc' true
'abc' LIKE 'a%' true
'abc' LIKE '_b_' true
'abc' LIKE 'c' false
```

LIKE pattern matching always covers the entire string. Therefore, if it's desired to match a sequence anywhere within a string, the pattern must start and end with a percent sign.

To match a literal underscore or percent sign the user must specify an escape character to be used as part of the pattern string by adding the ESCAPE clause after the pattern string. The respective character in the pattern must be preceded then by the escape character.

Some examples:

```
'a_b' LIKE 'a\_%' ESCAPE '\' true
'abc' LIKE 'a\_%' ESCAPE '\' false
```

7.6 Date/Time Functions

7.6.1 EXTRACT(from timestamp)

EXTRACT(selection-keyword FROM timestamp-expression)

The EXTRACT() function returns the value of the selected portion of a timestamp. The table below lists the supported keywords, the datatype of the value returned by the function, and a description of its contents.

Keyword	Datatype	Description
YEAR	INTEGER	The year as a numeric value.
QUARTER	INTEGER	The quarter of the year as a single numeric value between 1 and 4.
MONTH	INTEGER	The month of the year as a numeric value between 1 and 12.
DAY	INTEGER	The day of the month as a numeric value between 1 and 31.
WEEK	INTEGER	The week of the year as a numeric value between 1 and 52.
HOUR	INTEGER	The hour of the day as a numeric value between 0 and 23.

Keyword	Datatype	Description
MINUTE	INTEGER	The minute of the hour as a numeric value between 0 and 59.
SECOND	INTEGER	The whole part of the number of seconds within the minute as a value between 0 and 59.

```

rapids > select * from t3;
C1
--
2021-08-19 09:01:02.12

1 row(s) returned (0.06 sec)
rapids > select extract(second from c1) from t3;
[1]
---
2

1 row(s) returned (0.02 sec)

```

7.6.2 CURRENT_TIMESTAMP

CURRENT_TIMESTAMP

Returns the current date and time as a timestamp

```

rapids > select current_timestamp;
[1]
---
2020-12-19 19:10:21.746

1 row(s) returned (0.07 sec)

```

7.6.3 NOW()

NOW()

Returns the current date and time as a timestamp. Equivalent to CURRENT_TIMESTAMP.

```

rapids > select now();
[1]
---
2020-12-19 19:11:04.705

1 row(s) returned (0.06 sec)

```

7.6.4 Interval Arithmetic

7.6.4.1 Interval Types

RapidsDB provides support for INTERVAL arithmetic as defined by the SQL-99 standard. There are two types of intervals:

YEAR-MONTH INTERVAL

Examples:

- INTERVAL '1' YEAR
- INTERVAL '2' MONTH
- INTERVAL '1-2' YEAR TO MONTH

DAY_TIME INTERVAL

Examples:

- INTERVAL '5' DAY
 - INTERVAL '5 10:10' DAY TO MINUTE
 - INTERVAL '1 2:10:10.234' DAY TO SECOND
- and so on...

Precision:

The user can specify a leading precision for any of the intervals. The default precision for all of DAY, HOUR, MINUTE, SECOND, YEAR, MONTH is 2. The maximum precision allowed is 9. The default fractional second precision is 6, and the maximum is 9. You can specify precision for the leading field and also for the SECOND field, the remaining fields will follow the default precision.

The following are valid intervals:

- INTERVAL '1-2' YEAR TO MONTH
- INTERVAL '13-2' YEAR TO MONTH
- INTERVAL '199-2' YEAR(3) TO MONTH
- INTERVAL '199' MONTH(3) ---- NOTE: we have to specify the precision of three because the value 199 is greater than the default precision
- INTERVAL '1 10:10:10.234' DAY TO SECOND
- INTERVAL '123 10:10:10.234' DAY(3) TO SECOND ---- NOTE: we have to specify the precision of three because the value 123 is greater than the default precision
- INTERVAL '123 10:10:10.12345678' DAY(3) TO SECOND(8) ---- NOTE: we have to specify the precision of three because the value 123 for the DAY is greater than the default precision, and the precision for SECOND is also greater than the default
- INTERVAL '123 10:10:10.12345678' DAY(3) TO SECOND : the fractional second will round off to the default 6 digits precision, and you will get back: +123 10:10:10.123457 NOTE: we have to

specify the precision of three because the value 123 for the DAY is greater than the default precision

Range:

Can be negative or positive.

7.6.4.2 YEAR-MONTH interval:

Can be negative or positive:

year - constrained by precision. Hence with a precision of 9 the maximum value can be 999999999
month - 0 to 11 (But if leading then constrained by precision).

The following example is valid:

INTERVAL '10-10' YEAR TO MONTH.

but the following is invalid:

INTERVAL '10-13' YEAR TO MONTH.

The following is also valid:

INTERVAL '13' MONTH

This is valid because MONTH is the leading number, and so it is constrained by precision, and the leading default precision is 2. So you can have a max value of 99 in month. But if you specify precision of more than 2 it can be higher.

For example, you can have:

INTERVAL '999' MONTH(3)

7.6.4.3 DAY-TIME interval:

Can be negative or positive:

day - constrained by precision. Hence with a precision of 9, the maximum value can be 999999999
hour - 0 to 23
minute - 0 to 59
second - 0 to 59.999999999

Note that if hour, minute or second are leading then we can specify a precision other than the default for them.

e.g INTERVAL '999' HOUR(3)

Also note that we can give a fractional second precision:

e.g INTERVAL '10:20.30.888' HOUR TO SECOND(3).

We can also have:

INTERVAL '10.89' SECOND(2,2)

Also note that with the fractional second, if the number does not fit the precision, it will get rounded.

e.g INTERVAL '10.23456' SECOND(2,4)
will become '+10.2346'

Support for interval comparisons:

We can compare DATE-TIME intervals with DATE-TIME intervals.
We can compare YEAR-MONTH intervals with YEAR-MONTH intervals.

7.6.4.4 Support for Interval Arithmetic:

Operand	Operator	Operand	Result Type
Timestamp	-	Timestamp	Interval
Timestamp	+	Interval	Timestamp
Timestamp	-	Interval	Timestamp
Interval	+	Timestamp	Timestamp
Interval	+	Interval	Interval
Interval	-	Interval	Interval
Interval	*	Numeric	Interval
Numeric	*	Interval	Interval
Interval	/	Numeric	Interval

Notes:

1. When you do arithmetic on intervals, the resulting interval has a precision of the maximum allowed (see examples below).
2. When doing interval arithmetic with a timestamp literal, the timestamp literal must be specified using the timestamp keyword (see examples below)

Examples:

In the following examples, the table t1 has column c2 defined as a timestamp column:

```

rapids > create table moxe.t1(c1 integer,c2 timestamp);
0 row(s) returned (0.10 sec)
rapids > insert into t1 values(1,'2018-11-12 10:12:13'),(2,'2019-12-08
09:11:30'),(3,'2017-01-01 06:12:50');
0 row(s) returned (0.09 sec)
rapids > select c2-interval '100' HOUR(3) from t1;

```

```

[1]
---
2018-11-08 06:12:13.0
2019-12-04 05:11:30.0
2016-12-28 02:12:50.0

3 row(s) returned (0.05 sec)
rapids > select c2-'2018-01-01' from t1;
[1]
---
315
706
-365

3 row(s) returned (0.05 sec)
rapids > select timestamp '2018-01-01 01:01:01' -interval '100'
      HOUR(3) from t1 limit 1;
[1]
---
2017-12-27 21:01:01.0

1 row(s) returned (0.05 sec)

```

7.6.4.5 *EXTRACT(from interval)*

EXTRACT(selection-keyword FROM interval-expression)

The EXTRACT() function returns the value of the selected portion of a timestamp. The table below lists the supported keywords, the datatype of the value returned by the function, and a description of its contents.

Keyword	From Interval	Datatype	Description
YEAR	Year-month	INTEGER	The year as a numeric value.
QUARTER	Year-month	INTEGER	The quarter as a numeric value.
MONTH	Year-month	INTEGER	The month as a numeric value
DAY	Day-time	INTEGER	The day as a numeric value
HOUR	Day-time	INTEGER	The hour as a numeric value.

Keyword	From Interval	Datatype	Description
MINUTE	Day-time	INTEGER	The minute as a numeric value.
SECOND	Day-time	INTEGER	The second as a numeric value.

```

rapids > select * from t4;
C1
--
2020-12-25 09:00:00.0

1 row(s) returned (0.06 sec)
rapids > select extract(year from c1) from t4;
 [1]
 ---
 2020

1 row(s) returned (0.06 sec)
rapids > select extract(second from c1) from t4;
 [1]
 ---
 0

1 row(s) returned (0.06 sec)
rapids > select extract(hour from c1) from t4;
 [1]
 ---
 9

1 row(s) returned (0.05 sec)

```

7.6.4.6 BETWEEN Operator:

BETWEEN interval1 AND interval2

The BETWEEN operator can return the value between two day-time or two year-month intervals. For example:

```

SELECT .... WHERE TS_INTERVAL BETWEEN INTERVAL '100 10:00:00.000 DAY TO SECOND AND INTERVAL
'299 10:00:00.000' DAY TO SECOND ...

```

7.7 CONDITIONAL EXPRESSIONS

7.7.1 CASE

The CASE expression is a generic conditional expression, similar to if/else statements in other programming languages:

CASE WHEN condition THEN result

```
[WHEN ...]
[ELSE result]
END
```

CASE clauses can be used wherever an expression is valid. Each condition is an expression that returns a boolean result. If the condition's result is true, the value of the CASE expression is the result that follows the condition, and the remainder of the CASE expression is not processed. If the condition's result is not true, any subsequent WHEN clauses are examined in the same manner. If no WHEN condition yields true, the value of the CASE expression is the result of the ELSE clause. If the ELSE clause is omitted and no condition is true, the result is null.

An example:

```
rapids > select num from t1;
NUM
---
  1
  2
  3

3 row(s) returned
```

```
rapids > select num, CASE WHEN num=1 THEN 'ONE' WHEN num=2 THEN 'TWO' ELSE 'OTHER' END AS
TEXT from t1;
NUM TEXT
---- ----
  1 ONE
  2 TWO
  3 OTHER

3 row(s) returned
```

7.7.2 COALESCE

```
COALESCE(value [, ...])
```

The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null. It is often used to substitute a default value for null values when data is retrieved for display, for example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

This returns description if it is not null, otherwise short_description if it is not null, otherwise the text '(none)'.

7.7.3 IF

```
IF( boolean_expression, true_result_expression, false_result_expression )
```

IF evaluates the `boolean_expression`, and then evaluates one of the other two expressions to produce a result. If the `boolean_expression` is true, then the `true_result_expression` is evaluated and returned as the result; otherwise the `false_result_expression` is evaluated and returned as the result.

`true_result_expression` and `false_result_expression` may be of any type but the two must match or be implicitly convertible to a common type.

Example:

```
SELECT IF(1<2, 2, 3) ...
```

This returns the value 3.

7.7.4 IFNULL

IFNULL(value1, value2)

The IFNULL function returns value2 if value1 is null; otherwise it returns value1, for example.

```
SELECT IFNULL(description, '(none)') ...
```

This returns the string '(none)' if the value for the description column is null, otherwise it returns the value for the column .

7.7.5 NULLIF

NULLIF(value1, value2)

The NULLIF function returns a null value if value1 equals value2; otherwise it returns value1, for example.

```
SELECT NULLIF(description, '(none)') ...
```

This returns a null value if the value for the description column equals '(none)' otherwise it returns the value for the description column.

7.8 AGGREGATE FUNCTIONS

Aggregate functions compute a single result from a set of input values.

Function	Argument Type(s)	Return Type	Description
avg(expression)	integer, decimal or float	double precision for a floating-point argument, otherwise same as the argument data type	the average (arithmetic mean) of all input values
count(*)		integer	number of input rows

count(expression)	any	integer	number of input rows for which the value of expression is not null
max(expression)	any numeric, string, or date/time types	same as argument type	maximum value of expression across all input values
min(expression)	any numeric, string, or date/time types	same as argument type	minimum value of expression across all input values
sum(expression)	Integer, decimal or float types	same as the argument data type	sum of expression across all input values

It should be noted that except for count, these functions return a null value when no rows are selected. In particular, sum of no rows returns null, not zero as one might expect. The COALESCE function can be used to substitute zero or an empty array for null when necessary.

7.9 SUB-QUERY EXPRESSIONS

7.9.1 IN

expression IN (subquery)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of IN is "true" if any equal subquery row is found. The result is "false" if no equal row is found (including the case where the subquery returns no rows).

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the IN construct will be null, not false.

```

rapids > select * from t1;
NUM NAME
-----
 1 a
 2 b
 3 c

3 row(s) returned (0.06 sec)
rapids > select * from t1 where num in (1,3);
NUM NAME
-----
 1 a
 3 c

2 row(s) returned (0.06 sec)

```

7.9.2 NOT IN

expression NOT IN (subquery)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of NOT IN is "true" if only unequal subquery rows are found (including the case where the subquery returns no rows). The result is "false" if any equal row is found.

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the NOT IN construct will be null, not true.

```
rapids > select * from t1;
NUM NAME
---- ----
 1 a
 2 b
 3 c

3 row(s) returned (0.06 sec)
rapids > select * from t1 where num not in (1,3);
NUM NAME
---- ----
 2 b

1 row(s) returned (0.05 sec)
```

7.9.3 EXISTS

EXISTS (subquery)

The argument of EXISTS is an arbitrary SELECT statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is "true"; if the subquery returns no rows, the result of EXISTS is "false".

```

rapids > select * from t1;
NUM NAME
----
 1 a
 2 b
 3 c

3 row(s) returned (0.05 sec)
rapids > select * from t2;
NUM VALUE
-----
 1 xxx
 2 yy
 3 zz

3 row(s) returned (0.06 sec)
rapids > select name from t1 where exists(select value from t2 where t1.num=t2.num and
value='xxx');
NAME
----
a

1 row(s) returned (0.15 sec)

```

7.10 Session Functions

7.10.1 CURRENT_USER

The CURRENT_USER keyword can be used in a SELECT list to return the username of the current session.

```

rapids > select current_user from rapids.system.tables limit 1;
[1]
---
RAPIDS

1 row(s) returned (0.05 sec)

```

7.10.2 CURRENT_CATALOG

The CURRENT_CATALOG keyword can be used in a SELECT list to return the name of the catalog that will be used for resolving table or object names that are not fully qualified. If a default catalog has not been set then this keyword will return NULL.

```

rapids > set catalog moxe;
rapids > select current_catalog from rapids.system.tables limit 1;
[1]
---
MOXE

1 row(s) returned (0.05 sec)

```

7.10.3 CURRENT_SCHEMA

The CURRENT_SCHEMA keyword can be used in a SELECT list to return the name of the schema that will be used for resolving table or object names that are not fully qualified. If a default schema has not been set then this keyword will return NULL.

```
rapids > set schema moxe;
rapids > select current_schema from rapids.system.tables limit 1;
[1]
---
MOXE

1 row(s) returned (0.02 sec)
```

7.11 VERSION()

The VERSION function returns version information about the RapidsDB software. There are three supported variations of the VERSION function:

1. VERSION() – returns version number and build date for the RapidsDB Software, the version of Linux and the Java version:

```
rapids > select version();
[1]
---
RapidsDB 4.3.1 2022-01-20, Linux 5.4.0-74-generic, Java 1.8.0_292 (openj9-0.26.0)

1 row(s) returned (0.08 sec)
```

2. VERSION(1) – returns the version number for the RapidsD software:

```
rapids > select version(1);
[1]
---
4.3.1

1 row(s) returned (0.02 sec)
```

3. VERSION(2) – returns the internal software repository commit id for the RapidsDB software:

```
rapids > select version(2);
[1]
---
bbc91e81a65297e3b945484d3b669087cf11ef77

1 row(s) returned (0.06 sec)
```

8 QUERY EXECUTION

8.1 RapidsDB SQL Statement Execution

RapidsDB will parse a SQL statement and build a query execution plan to execute the SQL statement. When optimizing the execution plan the RapidsDB Optimizer attempts to “push down” as much of the query logic as possible to the underlying Data Store using a minimum number of operations. Those parts of the query logic that cannot be pushed down will be executed by the RapidsDB Execution Engine. For example, when executing a JOIN that includes tables from two different Connectors, the join will take place in the RapidsDB Execution Engine. **EXPLAIN** (see 15.1) can be used to see which parts of the query will be executed in RapidsDB and to inspect the SQL statements that will be sent to the underlying Data Store.

For those parts of a query that are executed by the RapidsDB Execution Engine, there are two types of query plans which can be generated:

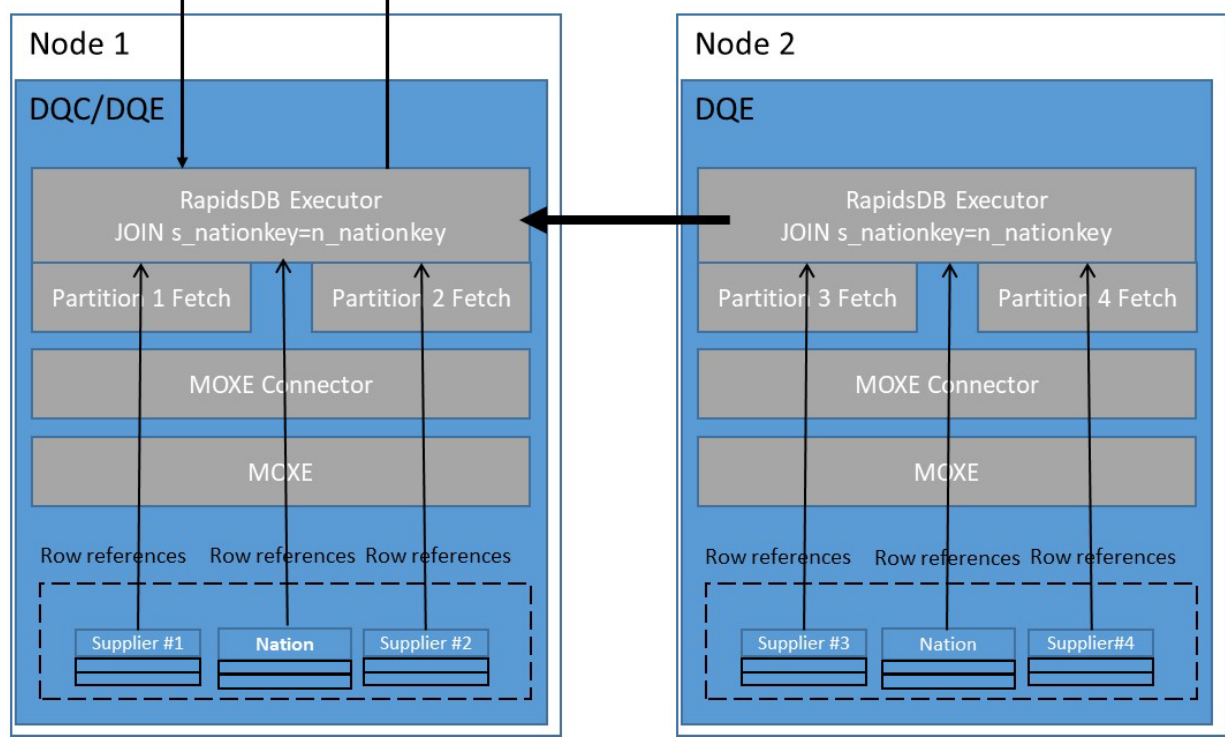
1. Partitioned Query Plans (see 8.2)
2. Non-Partitioned Query Plans (see 8.3)

NOTE: It is important to understand that Partitioned and Non-Partitioned Query Plans only apply to those parts of the query plan that cannot be pushed down to the underlying data stores, where the RapidsDB Execution Engine will be executing that part of the query plan. For queries that can be pushed down to the underlying data store, it is the responsibility of the underlying data store to parallelize the query execution where possible.

8.2 Partitioned Query Plans

MOXE and the Hadoop Connector support partitioning of the data across nodes in the RapidsDB cluster, which allows a query to be executed in parallel against each of the partitions of the tables being queried. For MOXE and the Hadoop Connector, RapidsDB will generate a Partitioned Query Plan, where portions of the query plan will be executed in the RapidsDB Execution Engine in parallel against each partition of a table. Figure 10 below illustrates this:

```
SELECT s_name, s_address, n_name FROM supplier, nation
WHERE s_nationkey = n_nationkey;
```



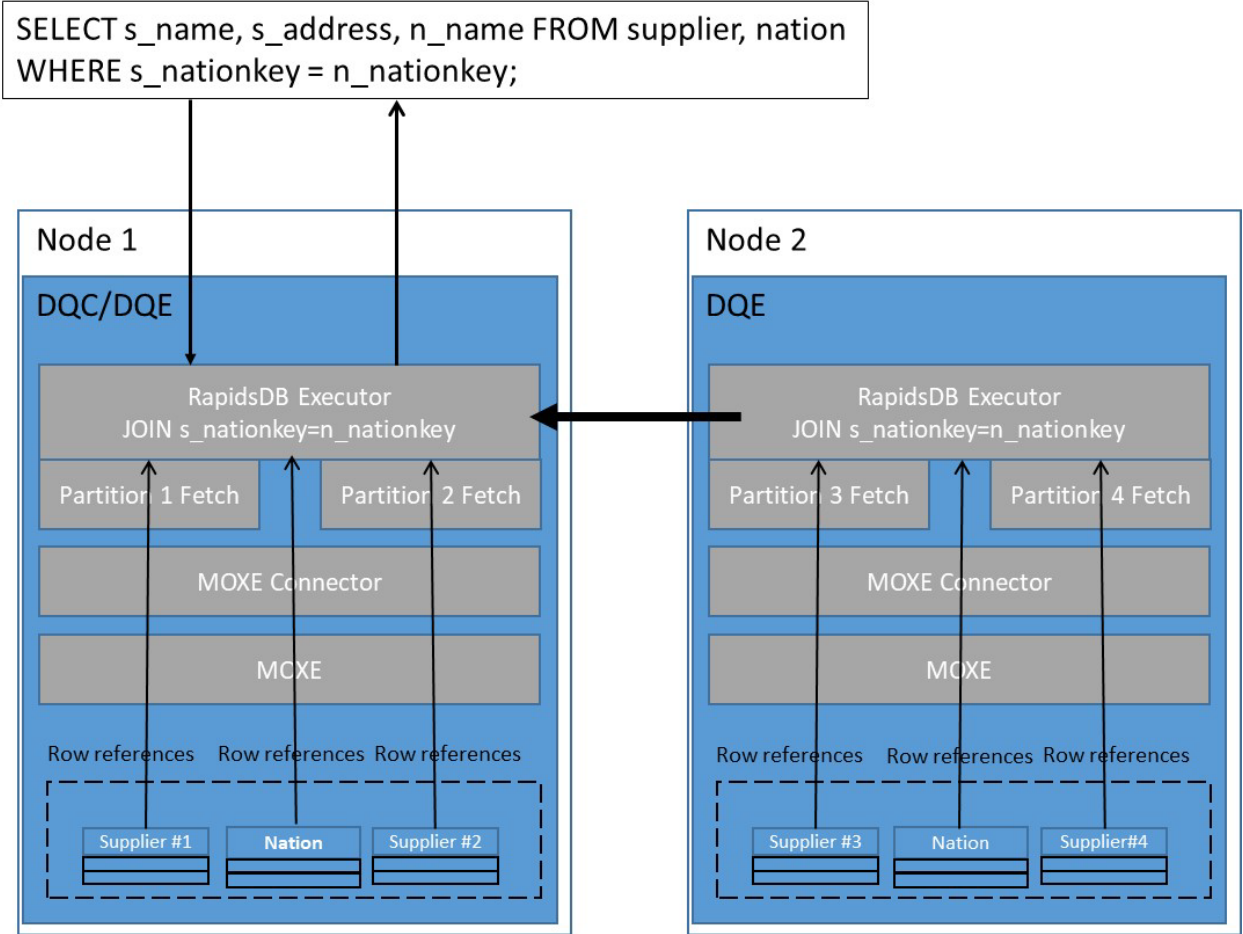


Figure 10. Partitioned Query Plan

In this example there is a join between the Supplier and Nation tables. The Supplier table is a partitioned table that is distributed across two nodes, and the Nation table is a replicated table with a copy of the table on each node. Each node will perform a parallel join between the partitions of the Supplier table and the Nation table, and then the results will be merged on the originating node for the query, which is Node1 in this example.

8.3 Non-Partitioned Query Plans

For other data stores where a single query cannot be split up and executed in parallel, RapidsDB will generate a Non-Partitioned Query plan, sending a single query to the underlying data store. It will be responsibility of the underlying data store to parallelize the execution of the query if possible. MemSQL is an example of a Data Store where the query cannot be split up by RapidsDB and executed in parallel, but MemSQL can the parallelize the execution of a SQL statement when it is pushed down to MemSQL. Figure 11 below illustrates this:

Example: `select l_orderkey, l_partkey, p_mfgr from part join lineitem on p_partkey = l_partkey and p_mfgr = 'LG';`

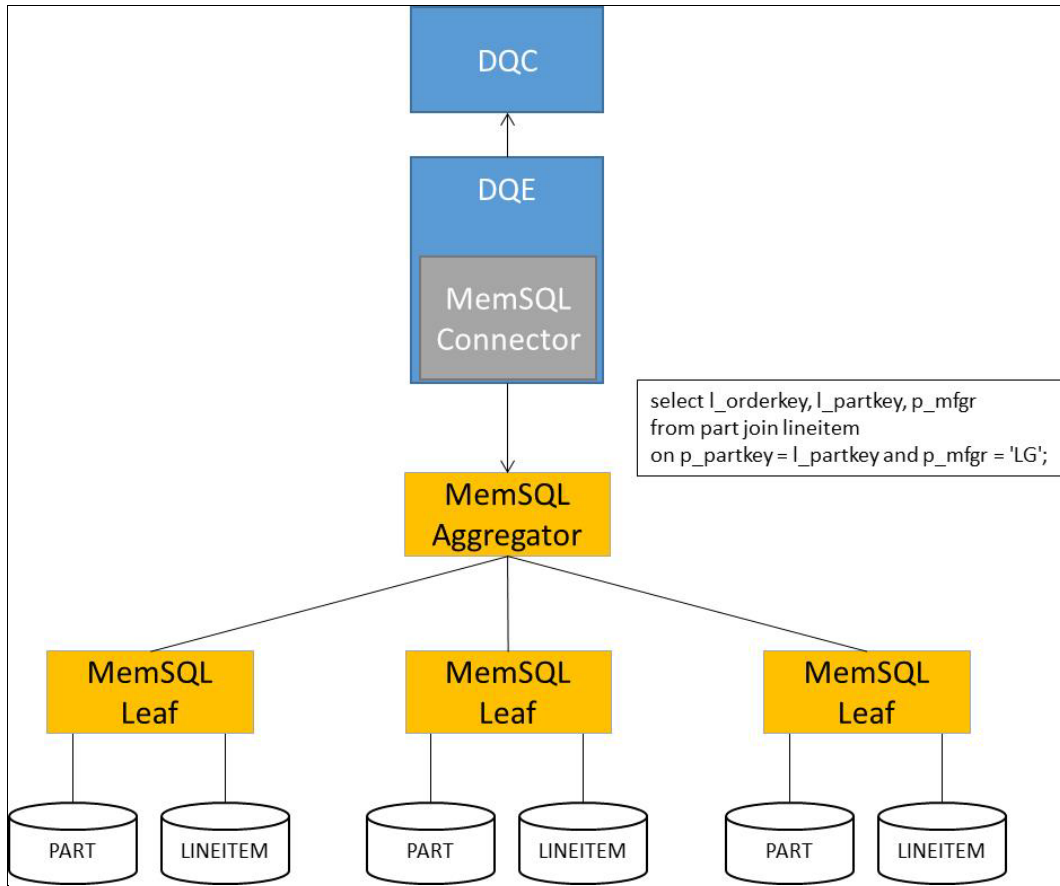


Figure 11. Non-partitioned Query Plan

Figure 11 shows the entire query getting pushed down to MemSQL (via the memSQL Aggregator) and then MemSQL parallelizing the execution across all of the MemSQL Leaf nodes in the MemSQL cluster.

Other data stores, such as Oracle, which are not distributed data stores, will execute the query on a single node. Figure 12 below shows the same query executed against Oracle:

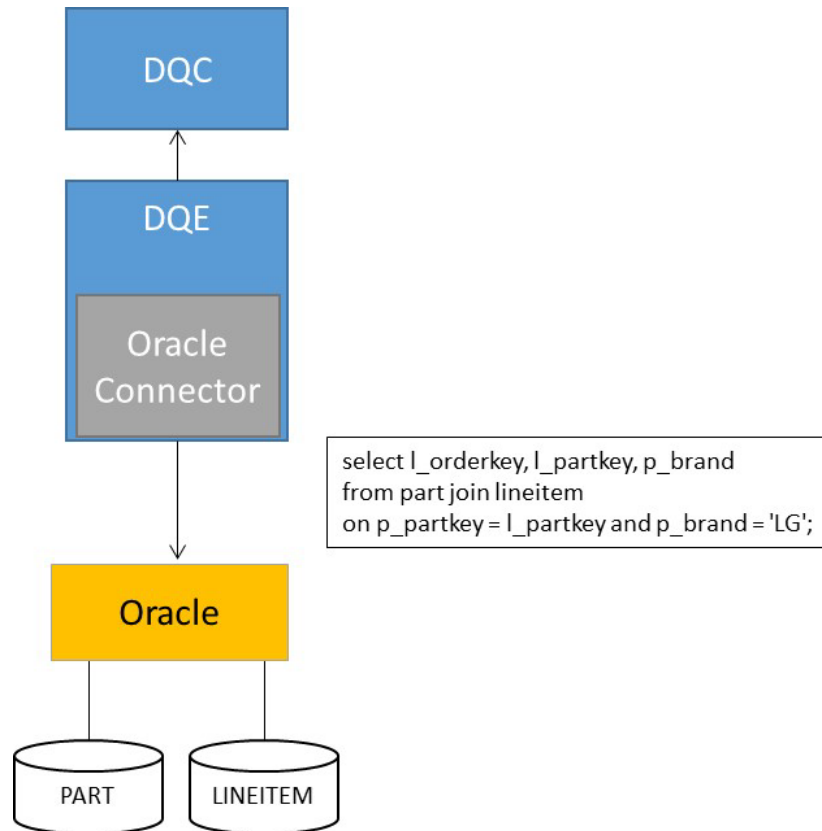


Figure 12 Non-partitioned query plan

8.4 Combination of Partitioned and Non-Partitioned Plans

When joining tables across Connectors where one Connector uses Partitioned plans and the other Connector uses Non-Partitioned plans (eg. MOXE and MemSQL), the join will be executed by RapidsDB, and RapidsDB will push down as much of the processing as possible before performing the join.

Example: `select l_orderkey, l_partkey, p_mfgr from part join lineitem on p_partkey = l_partkey and p_mfgr = 'LG' and l_shipdate <= '2020-01-01';`

This is the same query as the examples above, but in this example the PART table is managed by MOXE and the LINEITEM table is managed by Oracle.

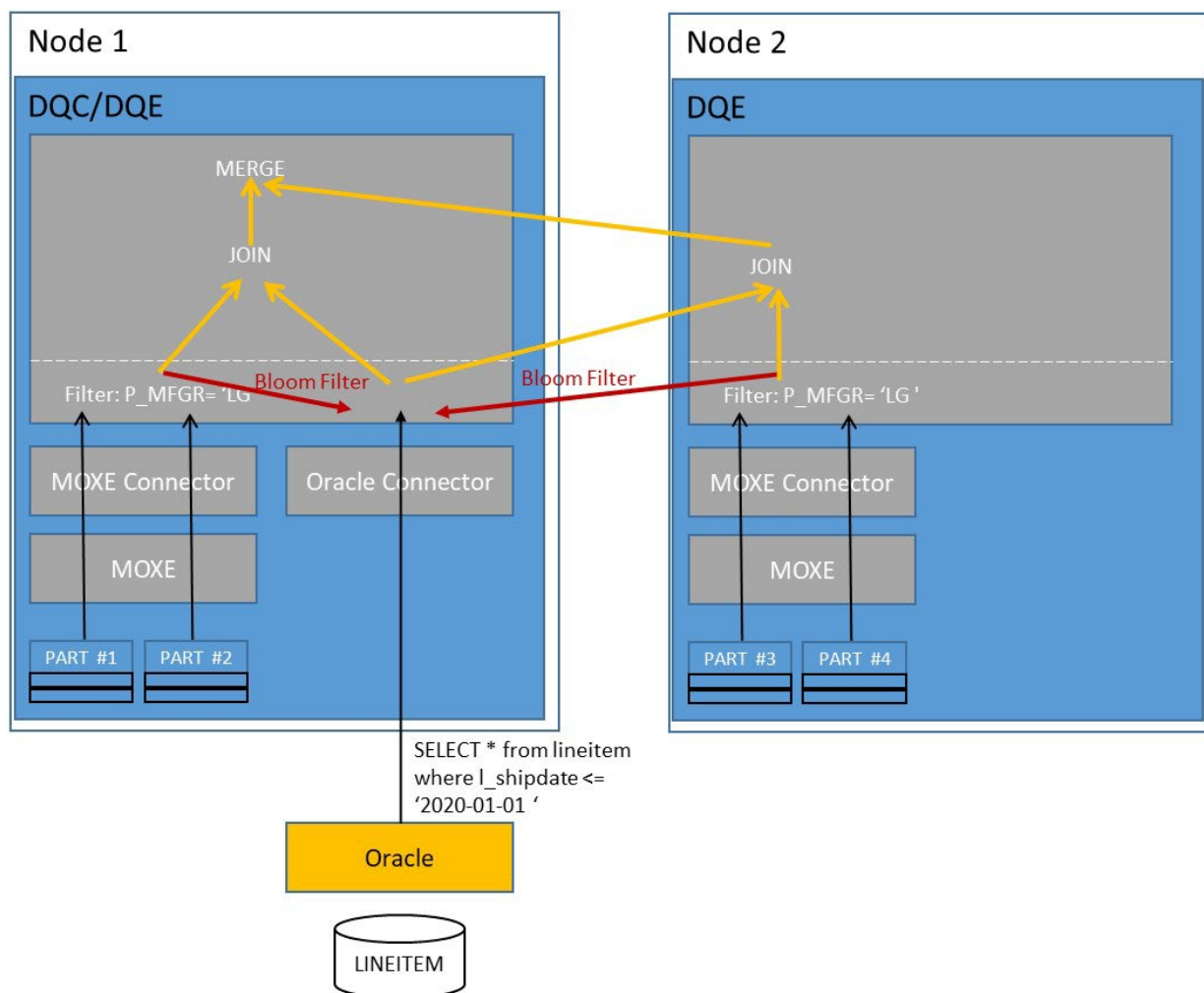


Figure 13. Combination of Partitioned and Non-Partitioned Plans

Figure 13 shows that the data from the `PART` table will be retrieved from MOXE in parallel using a Partitioned plan, and the data from the `LINEITEM` table will be retrieved from Oracle using a Non-Partitioned plan. The steps of the plan will be:

- 1 Execute a partitioned fetch from MOXE against the `PART` table, and apply a filter to the data (`p_mfgr='LG'`)
- 2 Build a set of Bloom filters using the data returned from the `PART` table for each partition
- 3 Execute a non-partitioned fetch from Oracle against the `LINEITEM` table with the predicate `l_shipdate<='2020-01-01'`
- 4 Send the bloom filters to the node with the `LINEITEM` data and join the resulting rows with the rows returned from the `PART` table
- 5 The DQC will merge the results from the two nodes doing the Bloom joins and then return the results to the user

8.5 RapidsDB Join Algorithms

To reduce network data movement, RapidsDB automatically distributes join operations based on the network location(s) of the data.

The RapidsDB join optimizer analyzes available information about the size and location of the join operands (the tables or subqueries participating in the join) and plans the join operation with the goal of minimizing network cost by executing it co-locally with the data. If one or both operands are distributed across multiple network nodes, RapidsDB partitions the join operation and executes join operators on multiple nodes in parallel.

For equi-joins (i.e. where the join predicate contains at least one equality condition between the two operands) RapidsDB uses a hash join algorithm. The join operator(s) ingest the rows of one operand and build a hash index of the join keys for those rows. The operator then streams the rows of the other operand, looking up each row's join key in the index to determine whether it can satisfy the join predicate.

If the operands of an equi-join operator are not co-located, data must be streamed over the network. In this case a *Bloom filter* is dynamically created and sent to the location(s) from which data will be streamed. A Bloom filter is effectively a much smaller (but also less accurate) hash index that yields a simple yes/no answer as to whether a given row can potentially participate in the join. Candidate rows are skipped if they don't satisfy the filter, eliminating the need to send those rows over the network to be tested against the join predicate. Although a Bloom filter is not perfectly accurate (it will allow a few unqualified rows to pass), it nonetheless reduces network transmission significantly in most cases, resulting in significantly higher join performance.

For non equi-joins RapidsDB performs a distributed cross-join. To reduce network cost, the optimizer tries to plan the query such that the join operator(s) will run co-locally with the larger operand. Rows of the other operand are broadcast (if they are not co-located) and the join operator(s) apply the join predicate to every possible row combination to find combinations that satisfy the predicate. Testing all combinations can be quite time consuming, so joins of this type are not advisable if the operands have a large number of rows.

9 INSERT

The user can insert data into tables in the schema managed by the following Connectors:

- MOXE
- MySQL
- MemSQL
- Oracle
- Postgres
- Greenplum
- Hadoop

- JDBC

The syntax for the INSERT command is:

```
INSERT INTO [catalog.][schema.]<table name> [(col_name,...)]
VALUES (expr,...),(...),...
```

```
INSERT INTO [catalog.][schema.]<table name> [(col_name,...)]
SELECT select-query
col_name = insert_expr
[, col_name = insert_expr] ... ]
```

select-query: any valid select query as defined by section 6

NOTES:

1. The catalog and schema names are used to identify which Connector the INSERT command should be sent to. The catalog name is only needed in the situation where the schema name is not unique.
2. For an INSERT ... SELECT, the data types in the result set from the SELECT must be compatible with the data types for the target insert table.
3. FOR INSERT ... SELECT the tables specified in the INSERT and SELECT clauses can be in different schema managed by different Connectors.

Example:

- INSERT INTO test.t1 VALUES (1,'test text', '2015-01-01 00:00:00');

The INSERT would be sent to the Connector managing the schema named test to insert data into table t1

- INSERT INTO test.t1 VALUES (1,'test text', '2016-01-01 00:00:00'),(2,'text', '2016-02-01 12:00:00');

The INSERT for two rows would be sent to the Connector managing the schema named test to insert data into table t1

- INSERT INTO mysql.test.t1 (c1,c2) VALUES (1,'test text');

The INSERT would be sent to the MySQL Connector managing the schema named test to insert data into table t1 for columns c1 and c2, with default values for any other columns in table t1.

- INSERT INTO moxe.t1 SELECT t1, t2, t3 FROM memsql.test.t2;

The INSERT would be sent to the MOXE Connector.

10 DDL

The user can create and drop tables in the schema managed by the following types of Connectors:

- MOXE
- MySQL
- MemSQL
- Oracle
- Postgres
- Greenplum
- JDBC
- Hadoop (when used with the Hive metastore)

10.1 CREATE TABLE

The syntax for the CREATE TABLE command is:

```
CREATE TABLE [IF NOT EXISTS] <tableReference>
(
  <columnDefinition>, ...
  <indexDefinition>, ...
)
[ PARTITION [BY] (<expr>, ...) ] [COMMENT <string>]

where:

<tableReference> is:
  [catalog.][schema.]<table name>

<column definition> is:
  <columnName> <type> [[NOT] NULL] [COMMENT <string>]

<type> is:
  INTEGER [(precision)] |
  DECIMAL [(scale[, precision])] |
  FLOAT |
  VARCHAR [(size)] |
  BOOLEAN |
  DATE |
  TIMESTAMP

<column name> is: <SQL identifier>

<indexDefinition> is:
```

```
INDEX <indexName> [ON] ( <columnName>, ... )
```

NOTES:

1. The catalog and schema names are used to identify which Connector the CREATE TABLE command should be sent to. The catalog name is only needed in the situation where the schema name is not unique.
2. The column name must be a valid SQL identifier (see 5.1.1). If the column name is a reserved word then it must be enclosed in double quotes, however, the target database may still reject a quoted identifier for some reserved words. For example, Postgres will not accept “select” as a quoted identifier for a column name.
3. After creating the table, the metadata for the associated Connector will be refreshed, and there is no need to manually run the REFRESH command.
4. For the Integer, Float, Decimal, and VARCHAR data types the actual size, precision and scale of the columns will be determined by the underlying data store and can be different from the value specified by the user. The DESCRIBE TABLE command can be used to see the column information, for example:

```
rapids > describe table region;
```

TABLE_NAME	COLUMN_NAME	DATA_TYPE	ORDINAL	IS_PARTITION_KEY	IS_NULLABLE	PRECISION	SCALE
REGION 0	R_REGIONKEY	INTEGER	1	false	false	10	
REGION NULL	R_NAME	VARCHAR	2	false	false	25	
REGION NULL	R_COMMENT	VARCHAR	3	false	false	152	

```
3 row(s) returned (0.04 sec)
```

(refer to the Rapids-shell User Guide for more information on describe table)

The column information in the COLUMNS table in the RapidsDB System Metadata (see 13.9) for the created table reflects the type, size, precision and scale of columns as reported by the underlying data store and interpreted by RapidsDB. The following query can be used to see the column information for the created table:

```
SELECT * FROM rapids.system.columns
```



```
WHERE catalog_name = '<catalog name for new table>' AND
schema_name = '<schema name for new table>' AND
table_name = '<name of new table>';
```

The tables below show how the data types are handled across MOXE, MySQL, MemSQL, Postgres (includes Greenplum) and Oracle when issuing a CREATE TABLE statement from the rapids-shell:

MOXE		
Data Type	MOXE Data Type	Comments
INTEGER	INTEGER	Precision=19
INTEGER(n)	INTEGER	Precision ignored and set to 19
FLOAT	FLOAT	64-bit double precision
FLOAT(n)	FLOAT	64-bit double precision
DECIMAL	DECIMAL(19,0)	
DECIMAL(p,s)	DECIMAL(p,s)	Maximum precision of 19
BOOLEAN	BOOLEAN	
DATE	DATE	
TIMESTAMP	TIMESTAMP	Maximum of 6 digits precision for seconds
VARCHAR	VARCHAR	Defaults to maximum size of 65,536 bytes
VARCHAR(n)	VARCHAR(n)	The maximum size is 65,536 bytes.

MEMSQL		
Data Type	MemSQL Data Type	Comments
INTEGER	BIGINT	Precision=19
INTEGER(n)	BIGINT	Precision=19
FLOAT	DOUBLE	Precision=22

FLOAT(n)	DOUBLE	Precision=22
DECIMAL	DECIMAL(19,0)	
DECIMAL(p,s)	DECIMAL(p,s)	Max precision is 65, max scale is 30
BOOLEAN	TINYINT(1)	
DATE	DATE	
TIMESTAMP	TIMESTAMP	
VARCHAR	VARCHAR	Defaults to 255 characters
VARCHAR(n)	VARCHAR(n)	Max of 21,845

MYSQL		
Data Type	MemSQL Data Type	Comments
INTEGER	BIGINT	Precision=19
INTEGER(n)	BIGINT	Precision=19
FLOAT	DOUBLE	Precision=22
FLOAT(n)	DOUBLE	Precision=22
DECIMAL	DECIMAL(19,0)	
DECIMAL(p,s)	DECIMAL(p,s)	Max precision is 65, max scale is 30
BOOLEAN	TINYINT(1)	
DATE	DATE	
TIMESTAMP	TIMESTAMP	
VARCHAR	VARCHAR	Defaults to 255 characters
VARCHAR(n)	VARCHAR(n)	Values in VARCHAR columns are variable-length strings. The length can be specified as a value from 0 to 65,535. The effective maximum length of a VARCHAR is subject to the maximum row size (65,535 bytes, which is shared among all columns)

Oracle		
Data Type	Oracle Data Type	Comments
INTEGER	INTEGER	Precision=19
INTEGER(n)	INTEGER	Precision ignored and set to 19
FLOAT	FLOAT	64-bit double precision
FLOAT(n)	FLOAT	64-bit double precision
DECIMAL	DECIMAL(19,0)	
DECIMAL(p,s)	DECIMAL(p,s)	Maximum precision of 19
BOOLEAN	BOOLEAN	
DATE	DATE	
TIMESTAMP	TIMESTAMP	Maximum of 6 digits precision for seconds
VARCHAR	VARCHAR	Defaults to maximum size of 65,536 bytes
VARCHAR(n)	VARCHAR(n)	The maximum size is 65,536 bytes.

Postgres		
Data Type	Postgres Data Type	Comments
INTEGER	BIGINT	Precision=19
INTEGER(n)	BIGINT	Precision=19
FLOAT	FLOAT	Precision=53
FLOAT(n)	FLOAT	Precision=53
DECIMAL	DECIMAL(38,12)	

DECIMAL(p,s)	DECIMAL(38,12)	Max precision is 65, max scale is 12. Precision and scale are ignored and will always be set to 38 and 12 respectively
BOOLEAN	Not supported	
DATE	DATE	
TIMESTAMP	TIMESTAMP	Maximum of 6 digits precision for seconds
VARCHAR	VARCHAR	Defaults to maximum size of 65,536 bytes
VARCHAR(n)	VARCHAR(n)	The maximum size is 65,536 bytes.

Examples:

```
CREATE TABLE test.t1 (c1 integer not null, c2 varchar(64), c3 timestamp);
```

This command would be sent to the Connector managing the schema named “test” to create the table “t1”

```
CREATE TABLE mysql.test.t1 (c1 integer not null comment 'first column', c2 varchar(64) comment 'second column', c3 timestamp comment 'third column') comment 'test table';
```

This command would be sent to the Connector named “mysql” that is managing the schema “test” to create the table “t1”, with comments on all of the columns as well as the table.

```
CREATE TABLE test.t1 (“YEAR” integer not null, c2 varchar(64), c3 timestamp);
```

This command would be sent to the Connector managing the schema named “test” to create the table “t1” with the first column being named “YEAR”. This is an example of using a quoted identifier for a column name that is a reserved word.

10.2 Creating MOXE Tables

MOXE supports two types of tables, partitioned tables (see 10.2.1) and reference tables (see 10.2.2) as described below.

10.2.1 Partitioned Tables

A partitioned table is a table where the data is distributed across all of the nodes in the RapidsDB cluster where the associated MOXE Connector is running, and the data is partitioned using the columns specified by the “PARTITION [BY]” clause.

The following example creates a partitioned MOXE table with the column s_suppkey as the partitioning column:

```
rapids > create table moxe.SUPPLIER (
  > s_suppkey integer NOT NULL comment 'Supplier key',
  > s_name varchar(25),
  > s_address varchar(40),
  > s_nationkey integer,
  > s_phone varchar(15),
  > s_acctbal decimal(17,2),
  > s_comment varchar(101)
  > ) PARTITION (s_suppkey) comment 'Supplier table';
0 row(s) returned (0.15 sec)
```

This table also has a comment on the column s_suppkey and a table level comment. The comments can be seen by querying the RapidsDB COMMENTS and TABLES tables as shown below:

```
rapids > select * from tables where table_name='SUPPLIER';
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME  IS_PARTITIONED COMMENT
PROPERTIES
-----
MOXE          MOXE          SUPPLIER     true Supplier
table        NULL

1 row(s) returned (0.07 sec)
rapids > select * from columns where table_name='SUPPLIER';
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME  COLUMN_NAME  DATA_TYPE
ORDINAL      IS_PARTITION_KEY  IS_NULLABLE  PRECISION  PRECISION_RADIX
SCALE CHARACTER_SET  COLLATION  COMMENT  PROPERTIES
-----
MOXE          MOXE          SUPPLIER     S_SUPPKEY    INTEGER
0            true          false        64           2
NULL NULL      NULL        Supplier key  NULL
MOXE          MOXE          SUPPLIER     S_NAME       VARCHAR
1            false         true         NULL         NULL
NULL UTF16      BINARY     NULL         NULL
MOXE          MOXE          SUPPLIER     S_ADDRESS    VARCHAR
2            false         true         NULL         NULL
NULL UTF16      BINARY     NULL         NULL
```

MOXE	MOXE	SUPPLIER	S_NATIONKEY	INTEGER
3	false	true	64	2
NULL NULL	NULL	NULL	NULL	
MOXE	MOXE	SUPPLIER	S_PHONE	VARCHAR
4	false	true	NULL	NULL
NULL UTF16	BINARY	NULL	NULL	
MOXE	MOXE	SUPPLIER	S_ACCTBAL	DECIMAL
5	false	true	17	10
2 NULL	NULL	NULL	NULL	
MOXE	MOXE	SUPPLIER	S_COMMENT	VARCHAR
6	false	true	NULL	NULL
NULL UTF16	BINARY	NULL	NULL	

7 row(s) returned (0.08 sec)

10.2.2 Reference Tables

Reference tables are tables that are replicated to each node in the RapidsDB cluster where the associated MOXE Connector is running. Reference tables are typically used for small dimension tables which can result in improved query performance when doing JOINS because the JOINS to the reference tables can be completed locally on each node in the RapidsDB cluster avoiding any network overhead.

The following example creates a replicated table that will be replicated to every RapidsDB node in the cluster:

```
rapids > create table MOXE.REGION (
  > r_regionkey integer not null,
  > r_name varchar(25) not null,
  > r_comment varchar(152)
  > );
0 row(s) returned (0.27 sec)
```

10.3 CREATE TABLE [AS] SELECT

Allows the user to create a table automatically from the results of a query and then insert the query results into the table. This command can be used from the rapids-shell or from JDBC.

CREATE TABLE AS SELECT is a simple way to create a copy of an existing table or to create a materialized copy of a result set. It is similar to the INSERT...SELECT statements except that the INSERT...SELECT statement appends rows to a table that already exists. As such, CREATE TABLE [AS] SELECT is a quick and easy way to take a copy of a result set and save it in a separate table.

The column names will default to the column names from the associated columns in the SELECT query, but the names can also be specified explicitly. If the SELECT query is providing literal values for the columns, then the column names will be "col1", "col2", etc.

The data types for the columns in the newly created table will default to the data types from the associated columns in the SELECT query. The user can also specify the data types to be used and in this case if the data types of the columns from the SELECT query do not match those specified for the table then the columns of the SELECT query will be cast to match. If this results in an incompatible data type cast then an error will be returned.

The addition of the column data types as well as the AS clause is a RapidsDB extension to the SQL standard.

Syntax:

```
statement := CREATE TABLE [ IF NOT EXISTS ] <tableName>
[ ( <tableDefinition> ) ]
[ <partitionInformation> ]1
[ <tableProperties> ] 2
[AS] <subquery> [ WITH [NO] DATA ];
tableDefinition := <objectDefinition> [ , <objectDefinition> [ , ... ] ]
objectDefinition := <columnDefinition> | <tableConstraint> | <indexDefinition>
columnDefinition := <columnName> [ <columnType> [ <columnConstraint> ] ]
columnConstraint := NOT NULL | PRIMARY KEY | UNIQUE KEY
tableConstraint := PRIMARY KEY ( <expression> )
indexDefinition := UNIQUE KEY ( <expression> ) | KEY ( <expression> )
subquery := <selectOrValuesQuery> | ( <selectOrValuesQuery> )
selectOrValuesQuery := <selectQuery> | <valuesQuery>
selectQuery := SELECT <selectQueryExpression>
valuesQuery := VALUES ( <expression> [ , <expression [ , ... ] ] ) [ , ( ... ) ]
```

10.3.1 Examples

```

rapids > describe table t1;
TABLE_NAME  COLUMN_NAME  DATA_TYPE  ORDINAL  IS_PARTITION_KEY  IS_NULLABLE  PRECISION  SCALE
-----
T1          NUM          INTEGER     0        false             true          64         NULL
T1          NAME         VARCHAR     1        false             true          NULL        NULL

2 row(s) returned (0.24 sec)
rapids > create table moxel.t6 select * from t1;
0 row(s) returned (0.20 sec)
rapids > describe table t6;
TABLE_NAME  COLUMN_NAME  DATA_TYPE  ORDINAL  IS_PARTITION_KEY  IS_NULLABLE  PRECISION  SCALE
-----
T6          NUM          INTEGER     0        false             true          64         NULL
T6          NAME         VARCHAR     1        false             true          NULL        NULL

2 row(s) returned (0.26 sec)
rapids > select * from t6;
  NUM NAME
  ----
   1  a
   2  b
   3  c

3 row(s) returned (0.05 sec)

```

Creates a table t with columns and data from t1. This statement is compliant with the SQL standard.

```

rapids > create table moxel.t7 select * from t1 WITH NO DATA;
0 row(s) returned (0.09 sec)
rapids > describe table t7;
TABLE_NAME  COLUMN_NAME  DATA_TYPE  ORDINAL  IS_PARTITION_KEY  IS_NULLABLE  PRECISION  SCALE
-----
T7          NUM          INTEGER     0        false             true          64         NULL
T7          NAME         VARCHAR     1        false             true          NULL        NULL

2 row(s) returned (0.23 sec)
rapids > select * from t7;
0 row(s) returned (0.05 sec)

```

Creates a table t7 which is a copy of the table t1 but without any data . This statement is compliant with the SQL standard.

```

rapids > create table moxel.t8 as select name from t1;
0 row(s) returned (0.20 sec)
rapids > describe table t8;
TABLE_NAME  COLUMN_NAME  DATA_TYPE  ORDINAL  IS_PARTITION_KEY  IS_NULLABLE  PRECISION  SCALE
-----
T8          NAME         VARCHAR     0        false             true          NULL        NULL

1 row(s) returned (0.20 sec)
rapids > select * from t8;
NAME
----
 a
 b
 c

3 row(s) returned (0.05 sec)

```

Creates a table t8 with column name from t1.


```

rapids > create table moxe1.t1 as values (1,'abc',12.1,1.0e0);
0 row(s) returned (0.20 sec)
rapids > describe table moxe1.t1;
TABLE_NAME  COLUMN_NAME  DATA_TYPE  ORDINAL  IS_PARTITION_KEY  IS_NULLABLE  PRECISION  SCALE
-----
T1          COL1         INTEGER     0         false             false        64         NULL
T1          COL2         VARCHAR     1         false             false        NULL       NULL
T1          COL3         DECIMAL     2         false             false        3          1
T1          COL4         FLOAT       3         false             false        53         NULL

4 row(s) returned (0.20 sec)
rapids > select * from t1;
COL1 COL2  COL3  COL4
-----
1 abc  12.1  1.0

1 row(s) returned (0.07 sec)

```

Creates a table t1 with automatically named columns (“col1”, “col2”, “col3” and “col4”) and one row of data from the VALUES clause. The data types of the columns are determined by how the literals are expressed in the VALUES clause according to the SQL standard (e.g., 12.1 is a decimal while 1.0e0 is a float).

```

rapids > create table moxe1.t2(id, name, price, disc) as select * from t1;
0 row(s) returned (0.19 sec)
rapids > select * from t2;
ID NAME  PRICE  DISC
-----
1 abc   12.1  1.0

1 row(s) returned (0.05 sec)

```

Creates a table t with columns named “id”, “name”, “price” and “disc” and filled with data from columns a, b and c from table u. Apart from the AS clause this statement is compliant with the SQL standard.

```

rapids > describe table t1;
TABLE_NAME  COLUMN_NAME  DATA_TYPE  ORDINAL  IS_PARTITION_KEY  IS_NULLABLE  PRECISION  SCALE
-----
T1          COL1         INTEGER     0         false             false        64         NULL
T1          COL2         VARCHAR     1         false             false        NULL       NULL
T1          COL3         DECIMAL     2         false             false        3          1
T1          COL4         FLOAT       3         false             false        53         NULL

4 row(s) returned (0.25 sec)
rapids > create table moxe1.t3(id integer, name varchar, price decimal(15,2), disc date) as select * from t1;
Error: Line 1 position 1: Type mismatch for column DISC: requires DATE using `com.rapidsdata.stdlib.FastDate` and there is no conversion from FLOAT(53 BITS) using `com.rapidsdata.stdlib.FastFloat`. locus=BORAY01

```

Attempts to create a table t3 where the column data type for the column “disc” is not compatible with the data type for the fourth column of the source table, “col4” and so an error is returned.

10.3.2 Semantics

1. In keeping with the regular CREATE TABLE statement, the catalog and schema names of the target table name will determine which data source will ultimately hold the table and data for this query.

2. If a table with the same name as the target table already exists in RapidsDB and IF NOT EXISTS is not specified, then the query will fail and no data from the subquery will be copied into the target table.
3. If a table with the same name as the target table already exists in RapidsDB, and IF NOT EXISTS is specified, then the query will return a success indicator to the user but no data from the subquery will be copied into the target table.
4. Specifying a <tableDefinition> allows the user to rename columns from the subquery. This end result can also be achieved by applying column aliases to the subquery instead. If the <tableDefinition> is not specified then the column names from the subquery will be used instead.
5. If the <tableDefinition> contains any duplicate column names then an error will be reported.
6. If a <tableDefinition> is specified and if the number of column names in <columnList> is not equal to the number of columns in the subquery then an error will be returned.
7. If a <tableDefinition> is specified with data types and the data types are incompatible with the column types of the SELECT statement then an error will be returned. An example of this would be specifying a column with a timestamp data type when the corresponding column in the SELECT query returns a boolean. The rules surrounding what data types can be cast are determined by the RapidsDB CAST operator.
8. When the target table is created there will be no indexes created on it unless a <tableDefinition> is provided and it contains index definitions. Creating a target table based on a query on a source table will not result in indexes from the source table being copied to the target table unless they are explicitly specified in a <tableDefinition> clause.
9. If a <tableDefinition> clause is not specified then the precision and scale of each column will be set according to the table below:

Datatype	Precision	Scale
Integer	19	0
Decimal	Will be preserved from result column up to a maximum of 19. If >19 or if the precision is unknown then it will be set to 19.	Will be preserved from the result column up to a maximum of 7. If the scale is >7 or the scale is unknown then it will be set to 7.
Float	Will not be specified. All floating point columns will be created with a datatype of FLOAT.	No scale. Scale has no meaning for an approximate data type.
Varchar	Will be preserved from the result column. If the result column has no precision then it will be set to 8000.	No scale.

All other types	No precision.	No scale.
-----------------	---------------	-----------

10. The nullability properties of columns in the SELECT query will be preserved in the target table being created. However no uniqueness constraints will be preserved as this implies automatic index creation.
11. If the WITH NO DATA clause is specified then the target table will be created according to the column definitions of the subquery however no data will be copied into the target table from the subquery.
12. If it is possible for RapidsDB to do so, the target table being created will be dropped if an error occurs while copying data into the table. Because RapidsDB is not transactional, there will be error scenarios where the query may fail and it is not possible for RapidsDB to drop the incomplete table automatically (e.g., if there is a problem with the connector, internal RapidsDB errors, etc).
13. By specifying table and column constraints in the CREATE TABLE AS SELECT statement (e.g., CREATE TABLE t (aa INTEGER PRIMARY KEY) AS SELECT a FROM u;), if the SELECT query retrieves a result set that does not match the column constraint (e.g., the values of column a in the above query are not unique) then the query will fail while copying the data. For a large data set, it could take a while before this constraint violation is detected and the failure status returned to the user. Examples of this would include violations of column uniqueness or nullability (e.g., CREATE TABLE t (col1 INTEGER NOT NULL) AS VALUES (NULL);)

10.3.3 Exclusions

1. The CREATE TABLE AS SELECT statement will not be executed transactionally since RapidsDB has no support for transactions. This means that it is possible for the table to be created successfully but an error occurs while copying the data such that the statement fails and the table is not able to be cleaned up (e.g., a communication problem with the underlying data source).
2. In this release CREATE TABLE AS SELECT statements will not support being pushed down directly to the underlying data source if the entire statement occurs directly in that data source. This is because the syntax of CREATE TABLE AS SELECT statements can vary significantly across data sources (and RapidsDB only supports a common subset that is defined in the standard).
3. RapidsDB will not support SELECT...INTO statements as a synonym of CREATE TABLE AS SELECT.
4. If a value from the subquery exceeds the precision or scale of the table definition then the underlying storage engine may return an error or it may silently truncate/round the data value when it is being inserted into the table.

10.3.4 Error Conditions

The following are a common set of conditions that will cause RapidsDB to generate an error:

1. Specifying a column name list where the number of column names does not match the number of columns in the subquery.

2. Specifying a column name list or table definition where a column name is not unique.
3. Specifying a full table definition but the data type of a column is not compatible with the data type of the corresponding column in the subquery and the subquery value cannot be cast.
4. Specifying a VALUES subquery where all values for a given column are NULL. In this case the data type of the column for the CREATE TABLE statement cannot be determined.
5. Specifying a VALUES subquery with multiple rows where the data type for a given column is not consistent across all rows.
6. Specifying a table definition with a column constraint (e.g., NOT NULL) where the subquery data does not conform to that constraint (e.g. contains NULL values).
7. Specifying a table definition with a table constraint (e.g., PRIMARY KEY) where the subquery data does not conform to that constraint (e.g., non-unique values across PK columns).
8. Specifying a column name and data type for some columns but not specifying a data type for all columns.

10.4 CREATE INDEX

The syntax for the CREATE INDEX command is:

```
CREATE [UNIQUE] INDEX [IF NOT EXISTS] <indexName> ON <tableReference> ( <columnName>, ... )
```

where:

<tableReference> is:

```
[[<catalog>.] [<schema>.]<tableName>
```

NOTE:

MOXE does not support creating indexes.

Example:

```
CREATE UNIQUE INDEX idx1 on memsql.dw.t1 (c1);
```

This command would be sent to the MemSQL Connector that is managing the schema “dw” to create a unique index on table “t1”.

10.5 DROP TABLE

The syntax for the DROP TABLE command is:

```
DROP TABLE [IF EXISTS] [[<catalog>.]<schema>.]<table name>;
```

NOTES:

1. The catalog and schema names are only needed when the <table name> is not unique.

2. After dropping the table, the metadata for the associated Connector will be refreshed, and there is no need to manually run the REFRESH command.

Examples:

```
DROP TABLE test.t1;
```

This command would be sent to the Connector managing the schema named “test” to drop the table “t1”

```
DROP TABLE memsql.test.t1;
```

This command would be sent to the MemSQL Connector that is managing the schema “test” to drop the table “t1”

10.6 TRUNCATE TABLE

The TRUNCATE TABLE deletes all of the data from a table and any associated indexes.

The syntax for the TRUNCATE TABLE command is:

```
TRUNCATE TABLE [[<catalog>.<schema>.<table name>];
```

NOTES:

1. The catalog and schema names are only needed when the <table name> is not unique.

Examples:

```
TRUNCATE TABLE rapidsse.public.t1
```

This command would be sent to the RapidsSE Connector to delete all of the data from table t1.

```
TRUNCATE TABLE memsql.test.t1;
```

This command would be sent to the MemSQL Connector that is managing the schema “test” to delete the data from table t1.

11 IMPORT/EXPORT Using IMPEX Connector

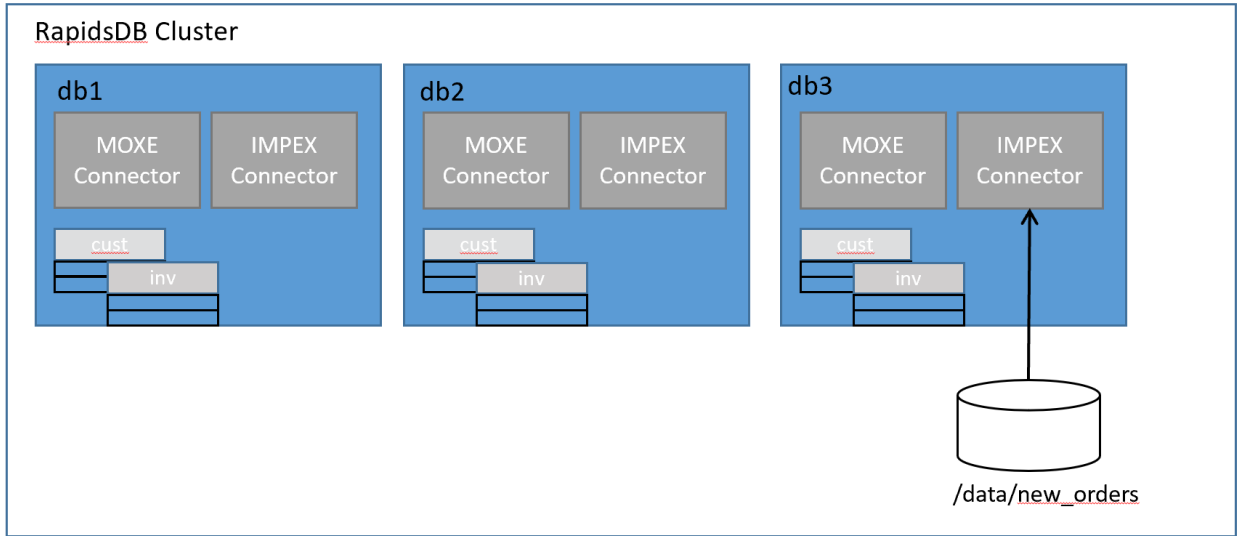
11.1 Overview

The IMPEX Connector is a new style of Connector that was introduced in Release 4.3. An IMPEX Connector supports the ability to treat disk files as regular tables which can participate in federated queries (ie. in SELECT or INSERT queries). The implication of this is that the user does not need to go through an ETL process in order to load the data from the files into regular tables, such as MOXE tables,

instead, the files can be queried directly from the disk. For Release 4.3, an IMPEX Connector can read csv (delimited) files from any node in the RapidsDB Cluster, in future releases other file systems such as Amazon S3, Google Cloud and HDFS will be supported along with other file formats such as Parquet and ORC. After any data has been written to disk (in a supported format, ie csv for Release 4.3) it is available for querying. If needed, the user can also use an IMPEX Connector to load all or a subset of the data into regular tables, such as MOXE tables or other federated data sources such as Oracle, Postgres or MySQL. When reading the data from disk, an IMPEX Connector supports both column pruning and predicate pushdown so that only the data that is needed for the query is passed to the RapidsDB Execution Engine thereby allowing very large data files to be processed by the RapidsDB Engine where the size of the data can exceed the memory of the system. When reading the disk files the user does not need to define a schema for the table, an IMPEX Connector can estimate the data type for each field in the data by reading a sample of the data and the imputing the data type based on the actual data. This means that users can do fast exploration of data files without having to first assign a schema for the table. For example, by using a LIMIT clause the user can quickly look at a subset of the data and then can use other SQL predicates to do more sophisticated analysis of the data. If the schema for a file (or set of files) is known, then the user can provide that schema to the IMPEX Connector as part of the query.

An IMPEX Connector also supports the capability to write query results to files. Finally, an IMPEX Connector supports bulk import to allow for the rapid loading of data from disk files into any federated tables, and bulk export to allow for the rapid writing of the contents of any federated tables to disk files. Bulk EXPORT provides the ability for the user to take a snapshot of the federated database, and bulk IMPORT provides the ability to reload that snapshot.

Example:



```
SELECT ... FROM (FOLDER 'node://db3/data/new_orders'), MOXE.CUST, MOXE.INV ....;
```

In the example above the IMPEX Connector is reading data from the folder “/data/new_orders” on RapidsDB node “db3” and that data is then getting joined with data from two MOXE tables, “cust” and “inv”.

11.2 IMPEX Connector Type

The IMPEX Connector type is used for creating Connectors that are used for doing import and export operations. The following sections provide more details on how to configure and use IMPEX Connectors.

11.3 Creating an IMPEX Connector

The user can create import and export Connectors using the IMPEX Connector type. To create an IMPEX Connector use the following command

```
CREATE CONNECTOR <name> TYPE IMPEX [WITH <key>=<value>' [,<key>=<value>']]
[<NODE * | NODE <node name> [NODE <node name>] [<further node names>]];
```

where <key> is one of the supported IMPEX Connector properties as defined in the next section.

Example:

```
CREATE CONNECTOR CSV TYPE IMPEX WITH DELIMITER='|', PATH='/';
```

Would create an IMPEX Connector named “CSV” that can run on any node in the RapidsDB cluster and where the delimiter character is '|', and the base path is the root directory ('/'). All other IMPEX properties would use default values as described below (see 11.4).

```
CREATE CONNECTOR CSV TYPE IMPEX WITH DELIMITER='|', PATH='/' NODE 'db1';
```

This would create the same Connector as the previous example with the one difference being that this Connector could only run on the RapidsDB node named “db1”.

11.4 IMPEX Connector Properties

The IMPEX Connector type supports the following properties which can be set either when creating the Connector using the CREATE CONNECTOR command (see examples below, also, refer to the Installation and Management Guide for more information on creating Connectors) or as part of an import reference (see 11.6) or export reference (see 11.7):

Key:	Default	Syntax	Description
FORMAT	'CSV'	'CSV' 'RAW'	Specifies the file format: <ul style="list-style-type: none"> • CSV: A delimited file (see section 11.5) • RAW: will produce a table with a single VARCHAR column containing the full text of each

			record in the imported file. See section 11.9.2.9 for examples)
PATH	'/var/tmp/rapids'	'<fully qualified path>'	Specifies the fully qualified path name to use as the base path name for all import references (see 11.6) or export references (see 11.7).
ERROR_PATH	'/var/tmp/rapids_errors'	'<fully qualified path>'	Specifies the fully qualified path name to use as the base path for the error files generated if an import operation fails (see 11.13.1 for more information).
ERROR_LIMIT	10	Integer, -1 0 >0	Specifies the maximum number of allowable errors on an import operation. Once the limit is reached the import will be terminated. The possible values are: -1 no limit 0 terminate on first error >0 terminate after specified number of errors See 11.13.2 for more information
BACKUP	false	[] true false	For EXPORT only. For bulk export operations (see 11.12), when the REPLACE option is specified, if BACKUP is “false”, then any existing files with a suffix of “.csv” in the specified folder or sub-folders prior to the export operation will get deleted and then new files created for the export. For bulk export operations (see 11.12), when the REPLACE option is specified, if BACKUP is “true”, then any existing files with a suffix of “.csv” in the specified folders or sub-folders prior to the export operation will be moved to a backup folder so that they can be recovered if needed and then new files created for the export. Note: if “true” or “false” are omitted and just the keyword “BACKUP” is specified, that is equivalent to “true”.

CHARSET	'UTF-8'	'<string>' as defined by the Java charset class https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html	Specifies the character set to be used. Some examples: 'GBK' 'GB2312' 'GB18030' 'Big5'
DELIMITER	','	'<char>' Non-empty, single character string	Specifies the field delimiter character. This can only be a single character.
ENCLOSED_BY	"" double quote	'<char>' Non-empty, single character string	Specifies whether a field is optionally enclosed by a specified character. This is commonly used to specify that string fields are optionally enclosed by either a single quote or double quote character and that character should not be included as part of the field data. If the same character is also included as part of the field data, then it must be escaped (see ESCAPE_CHAR below for more details).
ESCAPE_CHAR	'\'	'<char>' Non-empty, single character string	Specifies the character to be used as an escape character. This will allow the user to include embedded field delimiters and enclosed_by characters in the data .
FILTER	'*.*'	'<string>' Non-empty, character string using a REGEX format	For IMPORT only. The FILTER property allows the user to control which files are imported in a wildcard import operation and, optionally, how table names are created from the names of imported files. The FILTER value is a character string containing a Java regular expression (a "regex"). When performing a wildcard import, IMPEX examines each filename available from the import source. Only files whose names satisfy the FILTER regex are imported. (For a tutorial on Java regular expressions, see

			<p>https://www.oracle.com/technical-resources/articles/java/regex.html)</p> <p>A "capturing group" can be used in the regex to control how IMPEX creates a table name from the name of an imported file. The characters matched by the first group in the regex are used as the table name. If the regex contains no groups, then the table name will match the first part of the file name (before any dot suffixes).</p> <p>Note: for convenience, a FILTER value that starts with an asterisk is interpreted as a simple filename filter. For example FILTER='*.csv' will import all files with a ".csv" extension.</p>
GUESS	false	[] true false	<p>For IMPORT only</p> <p>When "false" specifies that the Connector should treat all columns as polymorphic strings, which will be automatically cast into the appropriate data types depending on the query (see 11.9.2.5 for more information).</p> <p>When "true" specifies that the Connector should derive the column data types for any columns whose data type has not been specified. The data types are derived by sampling the data being imported and then determining what the appropriate data type would be for each input field in the sampled data. For example, if the sampled data contained 100 records, and a given field contains alphanumeric characters for all 100 records, then it would be mapped to a VARCHAR column, if the field contained just integer characters then it would be mapped to an</p>

			<p>INTEGER, and so on (see 11.9.2.5 for more information).</p> <p>Note: if “true” or “false” are omitted and just the keyword “GUESS” is specified, that is equivalent to “true”.</p>
HEADER	false	[] true false	<p>When “true” specifies that the data file has a header record which has the column names to use on an import, or has the column names from the result set for an export.</p> <p>When “false” specifies that there is no header record.</p> <p>Note: if “true” or “false” are omitted and just the keyword “HEADER” is specified, that is equivalent to “true”.</p>
TERMINATOR	'\n'	'\n'	<p>Specifies how records are terminated. For this release the TERMINATOR is fixed as '\n', with an optional '\r'</p>
TRAILING	false	[] true false	<p>When “true” IMPEX will ignore a trailing field separator (i.e. where the field separator is immediately followed by the record terminator character) on each line of a file being imported and will append a trailing separator to each line of a file being exported.</p> <p>When “false” a trailing field separator will indicate a null value for the last column of the record being imported. For export no trailing field separator will be written out.</p> <p>Note: if “true” or “false” are omitted and just the keyword “TRAILING” is specified, that is equivalent to “true”.</p>

Examples:

```
CREATE CONNECTOR CSV TYPE IMPEX WITH DELIMITER='|';
```

Would create an IMPEX Connector named “CSV” where the delimiter character is '|'. The “PATH” property was not set and so would default to “/var/tmp/rapids”.

```
CREATE CONNECTOR CSV TYPE IMPEX WITH DELIMITER='|', PATH='/';
```

Would create an IMPEX Connector named “CSV” where the delimiter character is '|' and the “PATH” property is set to the root directory (“/”).

11.5 CSV (Delimited) File Formatting

This section describes how IMPEX Connectors handle the different CSV file formatting properties described in the previous section when reading and writing delimited data.

11.5.1 Text Handling

11.5.1.1 ESCAPE SEQUENCES

There are a set of special characters that only come into effect when prefixed with the escape character (by default the escape character is set to the backslash character). In the following table, the ESCAPE_CHAR is set to the backslash character. When an escape sequence is detected in the input data it will get replaced with its associated ASCII character as shown in the table below:

Escape Sequence	ASCII Character
\b	A backspace character <x08>
\f	A form feed character <x0C>
\n	A newline (linefeed) character <x0A>
\r	A carriage return character <x0D>
\t	A tab character <x09>
\v	A vertical tab character <x0B>

On output, any ASCII escape characters will get replaced by their associated escape sequence.

Example 1:

This example shows the output for a file using the tab (\t) and newline (\n) escape characters.

Input file: /var/tmp/rapids/tab_and_newline.csv:

```
123456789012345678901234567890
\tTabbed field\nNewline
```

```

rapids > select * from ('node://db1/text/tab_and_newline.csv');
COL1
----
123456789012345678901234567890
      Tabbed field
Newline

2 row(s) returned (0.06 sec)

```

Example 2:

This example shows all of the possible escape sequences (using the default escape character) being read from the file “/var/tmp/rapids/text/escape_seq.csv” and then written out to the file “/var/tmp/rapids/text/escape_seq_out.csv”.

Input file /var/tmp/rapids/text/escape_seq.csv:

```

Tab: \t Form Feed: \f Backspace: \b Newline: \n Vertical: \v Return: \r

```

```

rapids > select * from ('node://db1/text/escape_seq.csv') to
('node://db1/text/escape_seq_out.csv');

0 row(s) returned (0.08 sec)

```

Output file: /var/tmp/rapids/text/escape_seq_out.csv:

```

"Tab: \t Form Feed: \f Backspace: \b Newline: \n Vertical: \v Return: \r"

```

11.5.1.2 Handling of Leading and Trailing Blanks

Leading and trailing space characters are considered part of a VARCHAR column.

When the ENCLOSED_BY (see 11.5.6) is used to enclose the string, the leading and trailing space characters are ONLY those characters contained within the enclosed string (see example below for more on this), any space characters outside of the enclosing characters are ignored. When the string is not enclosed by the ENCLOSED_BY character, then all characters in the field are included, including all leading and trailing blanks.

Example:

In this example the first record has two fields that are not enclosed by the ENCLOSED_BY character, and so all of the data between the field delimiters for those fields is included. In the third record, the second and third fields are enclosed, and so only the characters between the ENCLOSED_BY characters are included.

File: /var/tmp/rapids/text/lead_trail_blanks.csv:

```
1, 4 leading blanks,3 trailing blanks ,1
2,A2345678901234567890,A1234567890123456789,2
3," 4 leading blanks","3 trailing blanks ",3
```

```
rapids > select * from ('node://db1/text/lead_trail_blanks.csv');
  COL1 COL2                COL3                COL4
  ---- ----                ----                ----
    1    4 leading blanks  3 trailing blanks    1
    2 A2345678901234567890 A1234567890123456789  2
    3    4 leading blanks  3 trailing blanks    3

3 row(s) returned (0.07 sec)
rapids > select char_length(col3) from
('node://db1/text/lead_trail_blanks.csv');
 [1]
  ---
   20
   20
   20

3 row(s) returned (0.06 sec)
```

11.5.1.3 Empty Strings

An empty (zero-length) string is defined as a field with two adjacent ENCLOSED_BY characters (see 11.5.6) for more information on ENCLOSED_BY character). For example, the second field in the sample record below would be interpreted as an empty string assuming that the ENCLOSED_BY character is the double quote character:

Example:

File: /var/tmp/rapids/text/empty_string.csv:

```
1,"",1
```

```
rapids > select * from ('node://db1/text/empty_string.csv');
  COL1 COL2  COL3
  ---- ----  ----
    1           1

1 row(s) returned (0.05 sec)
rapids > select char_length(COL2) from ('node://db1/text/empty_string.csv');
 [1]
  ---
   0

1 row(s) returned (0.05 sec)
```

NOTE – this is different from an empty field, where there are two adjacent field delimiter characters, which is interpreted as a NULL value (see 11.5.4) for more information on nulls) as shown in the example below:

File: /var/tmp/rapids/text/null_string.csv:

```
1,,,
```

```
rapids > select * from ('node://db1/text/null_string.csv');
  COL1 COL2   COL3   COL4
  ---- ----   ----   ----
      1 NULL   NULL   NULL

1 row(s) returned (0.06 sec)
```

In this example, fields two through four are all interpreted as null values.

11.5.2 Dates and Timestamps

The format for dates is YYYY-MM-DD, and the format for the time portion of a timestamp is HH.MM.SS.nnnnnn

Example:

File: /var/tmp/rapids/text/date_and_timestamp.csv

```
1,2021-09-01,2021-09-01 11:17:23.123456
2,"2021-09-01","2021-09-01 11:17:23.123456"
```

```
rapids > select * from (FILE 'node://db1/text/date_and_timestamp.csv') AS
t(c1 integer, c2 date, c3 timestamp);
  C1 C2           C3
  -- --           --
   1 2021-09-01 2021-09-01 11:17:23.123456
   2 2021-09-01 2021-09-01 11:17:23.123456

2 row(s) returned (0.59 sec)
```

11.5.3 Booleans

The table below specifies the valid input values for booleans:

Column value	Possible Inputs
FALSE	0 any string start with one of the following characters: f, F, n, N
TRUE	>0 any string start with one of the following characters: t, T, y, Y

Example:

File: /var/tmp/rapids/text/booleans.csv:

```
0
false
FALSE
n
no
1
true
TRUE
y
YES
```

```
rapids > select * from ('node://db1/text/booleans.csv') AS t(c1 boolean);
  C1
  --
  false
  false
  false
  false
  false
  true
  true
  true
  true
  true
10 row(s) returned (0.06 sec)
```

11.5.4 NULL Values

A null value is represented by an empty field, where an empty field is defined as two adjacent delimiters with no intervening spaces, or a delimiter followed immediately by the record terminator.

NOTE, this is not the same as an empty string (see 11.5.1.3) which is defined as two adjacent “ENCLOSED_BY” characters.

Example:

File: /var/tmp/rapids/text/null_string.csv:

```
1,,,
```

```
rapids > select * from ('node://db1/text/null_string.csv') AS t(c1 integer,
c2 integer, c3 decimal, c4 varchar);
  C1      C2      C3 C4
  --      --      -- --
    1     NULL     NULL NULL

1 row(s) returned (0.07 sec)
```

In this example, fields two through four are all interpreted as null values.

11.5.5 DELIMITER='<char> | \t'

The field delimiter can be a single character or the tab character ('\t').

Example 1: using the pipe character as the delimiter

File: /var/tmp/rapids/text/delimiter.csv

```
1| 4 leading blanks|3 trailing blanks |1
2|12345678901234567890|12345678901234567890|2
3|" 4 leading blanks"|"3 trailing blanks "|3
```

```
rapids > select * from ('node://db1/text/delimiter.csv' WITH DELIMITER='|');
COL1  COL2                COL3                COL4
-----
1      4 leading blanks    3 trailing blanks    1
2      12345678901234567890 12345678901234567890 2
3      " 4 leading blanks"  "3 trailing blanks " 3

3 row(s) returned (0.07 sec)
```

Example 2: using the tab character as the delimiter:

File: /var/tmp/rapids/text/tab_delimiter.csv

```
1    4 leading blanks  3 trailing blanks  1
2    12345678901234567890 12345678901234567890 2
3    " 4 leading blanks" "3 trailing blanks " 3
```

```
rapids > select * from ('node://db1/text/tab_delimiter.csv' WITH
DELIMITER='\t');
COL1  COL2                COL3                COL4
```

```

-----
1          4 leading blanks    3 trailing blanks    1
2          12345678901234567890  12345678901234567890  2
3          4 leading blanks    3 trailing blanks    3

3 row(s) returned (0.06 sec)

```

11.5.6 ENCLOSED_BY='<char>' | ''''

Specifies whether an input field is optionally enclosed by the specified character. This is commonly used to specify that character fields are enclosed by either a single quote or double quote character and that character should not be included as part of the field data.

NOTES

1. To explicitly specify a single quote as the delimiter, you must enclose the single quote inside double quotes, all other characters are specified using single quotes.
2. Use of the ENCLOSED_BY for character fields is optional, and so an input record could include some fields using the enclosed_by character with other character fields not using the enclosed_by character as shown in the example below.
3. If the ENCLOSED_BY character is also included as part of the field data, then the character must be escaped (see ESCAPE_CHAR 11.5.7).
4. When exporting data, character fields will only be enclosed using the ENCLOSED_BY character when the data for that field includes one or more DELIMITER (see 11.5.5) characters.
5. Numerical and Boolean values cannot be enclosed

The default enclosed_by character is a double quote.

Examples:

ENCLOSED_BY	DATA TYPE	INPUT	DATA TO BE STORED
	VARCHAR	'DAVE's DATA'	'DAVE's DATA'
	VARCHAR	""DAVE's DATA""	'DAVE's DATA'
	VARCHAR	'DAVE\'s DATA'	'DAVE's DATA'
	VARCHAR	""DAVE"s DATA""	INVALID
	INTEGER	'9'	INVALID
	DECIMAL	'9.0'	INVALID
	FLOAT	'9.0'	INVALID
	TIMESTAMP	'2020-09-01 09:00:00'	2020-09-01 09:00:00
	BOOLEAN	'T'	FALSE
	BOOLEAN	"T"	TRUE
	INTEGER	"9"	9
	DECIMAL	"9.0"	9.0
	FLOAT	"9.0"	9.0

	TIMESTAMP	"2020-09-01 09:00:00"	2020-09-01 09:00:00
ENCLOSED_BY=""	VARCHAR	'DAVE's DATA'	INVALID
	VARCHAR	""DAVE's DATA""	INVALID
	VARCHAR	'DAVE\'s DATA'	DAVE's DATA
	VARCHAR	""DAVE"s DATA""	"DAVE"s DATA"
	INTEGER	'9'	9
	DECIMAL	'9.0'	9.0
	FLOAT	'9.0'	9.0
	TIMESTAMP	'2020-09-01 09:00:00'	2020-09-01 09:00:00
	BOOLEAN	'T'	TRUE
	BOOLEAN	"T"	FALSE
	INTEGER	"9"	INVALID
	DECIMAL	"9.0"	INVALID
	FLOAT	"9.0"	INVALID
	TIMESTAMP	"2020-09-01 09:00:00"	2020-09-01 09:00:00

Example 1: This example uses the default ENCLOSED_BY double quote character.

File: /var/tmp/rapids/text/default_enclosed_by.csv

```
1,'DAVE's DATA',"DAVE's DATA",T,9.0,9.0,"2020-09-01 09:00:00"
```

```
rapids > select * from ('node://db1/text/default_enclosed_by.csv') AS t(c1
integer, c2 varchar, c3 varchar, c4 boolean, c5 decimal, c6 float, c7
timestamp);
  C1 C2          C3          C4    C5    C6 C7
  -- --          --          --    --    -- --
  1 'DAVE's DATA'  DAVE's DATA  true  9.0  9.0 2020-09-01 09:00:00.0

1 row(s) returned (0.07 sec)
```

Example 2: This example sets the ENCLOSED_BY character to a single quote:

File: /var/tmp/rapids/text/single_quote_enclosed_by.csv

```
1,'DAVE\'s DATA',"DAVE"s DATA",T,9.0,9.0,'2020-09-01 09:00:00'
```

```
rapids > select * from ('node://db1/text/single_quote_enclosed_by.csv' WITH
ENCLOSED_BY='') AS t(c1 integer, c2 varchar, c3 varchar, c4 boolean, c5
decimal, c6 float, c7 timestamp);
  C1 C2          C3          C4    C5    C6 C7
  -- --          --          --    --    -- --
  1 DAVE's DATA  "DAVE"s DATA"  true  9.0  9.0 2020-09-01 09:00:00.0

1 row(s) returned (0.59 sec)
```

11.5.7 ESCAPE_CHAR='<char>'

Specifies the character to be used as the escape character. This allows the user to include embedded field delimiters and enclosed_by characters in the data .

Default: '\' (backslash)

Example 1:

Shows the escaping of the ENCLOSED_BY and DELIMITER characters when those characters are the defaults. The escaped characters are hilited:

File:/var/tmp/rapids/text/escape_char.csv

```
1,"Escaped ENCLOSED_BY\",Escaped DELIMITER ,,End of row
```

```
rapids > select * from ('node://db1/text/escape_char.csv');
  COL1 COL2                COL3                COL4
  ---- ----                ----                ----
    1 Escaped ENCLOSED_BY" Escaped DELIMITER , End of row

1 row(s) returned (0.56 sec)
```

Example 2:

Shows the same example as before except in this example the ESCAPE_CHAR is set to the dollar character. The escaped characters are hilited:

File: /var/tmp/rapids/text/escape_char_dollar.csv

```
1,"Escaped ENCLOSED_BY$",Escaped DELIMITER $,,End of row
```

```
rapids > select * from ('node://db1/text/escape_char_dollar.csv' WITH
ESCAPE_CHAR='$') ;
  COL1 COL2                COL3                COL4
  ---- ----                ----                ----
    1 Escaped ENCLOSED_BY" Escaped DELIMITER , End of row

1 row(s) returned (0.56 sec)
```

11.5.8 HEADER

Specifies whether the data file has a header record which has the column names to use.

File: /var/tmp/rapids/text/header.csv

```
id,name,dob
1,Jim Smith,2004-04-01
```

```

rapids > select * from ('node://db1/text/header.csv' WITH HEADER);
  id name      dob
  -- ----      ---
   1 Jim Smith  2004-04-01

1 row(s) returned (0.06 sec)

```

11.5.9 CHARSET

Specifies the character set to be used. Some examples:

- 'GBK'
- 'GB2312'
- 'GB18030'
- 'Big5'

Refer to <https://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html> for a list of the possible character set names.

Example:

This example shows the character set being set to the “GBK” character set:

File: /var/tmp/rapids/text/charset_gbk.csv

```
1, 今天天气很暖和上周六下午温度升至9度。太阳很明亮
```

```

rapids > select * from ('node://db1/text/charset_gbk.csv' WITH
CHARSET='GBK');

COL1    COL2
-----
1       湵婦お澶十敲寰塚燧鍛岍筑鍛∟段涓嬮峽媯|害銻囡嚙9辜┌◆倣お関冲緇鑄疼寒

1 row(s) returned (0.14 sec)

```

11.5.10 TRAILING

When “true” IMPEX will ignore a trailing field separator (i.e. where the field separator is immediately followed by the record terminator character) on each line of a file being imported and will append a trailing separator to each line of a file being exported.

When “false” a trailing field separator will indicate a null value for the last column of the record being imported. For export no trailing field separator will be written out.

Example 1:

This example shows how the trailing delimiter will be ignored:

File: /var/tmp/rapids/text/trailing.csv

```
1,field 2,field 3,
```

```
rapids > select * from ('node://db1/text/trailing.csv' WITH TRAILING);
COL1    COL2      COL3
----    ----      ----
1       field 2   field 3

1 row(s) returned (0.05 sec)
```

Example 2:

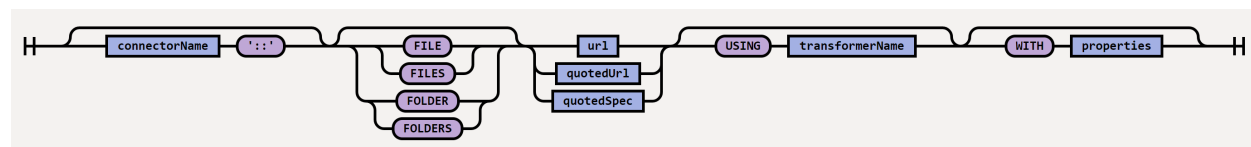
This example uses the same data file as example 1, but this time the “TRAILING” property is not set and so the trailing delimiter will be included in the import operation resulting in a null value for the final field:

```
rapids > select * from ('node://db1/text/trailing.csv');
COL1    COL2      COL3      COL4
----    ----      ----      ----
1       field 2   field 3   NULL

1 row(s) returned (0.05 sec)
```

11.6 IMPORT References

The user specifies the location and formatting information for the data to be imported using an Import Reference:



The table below describes each option:

Option	Required?	Default	Description
--------	-----------	---------	-------------

connectorName	No	IMPORT	The name of an existing IMPEX Connector.
FILE	No	FILE	Indicates that the name specified by the quotedUrl (see below) refers to a file to be imported. This is the default for non-bulk IMPORT/EXPORT operations
FILES	No	FILES	Used for bulk import operations (see 11.10). Indicates that name specified by the quotedUrl (see below) refers to a folder which contains a set of files to be imported. The name of each file (minus any dot suffixes) is the name of the table where the data from the file will be written. This is the default for bulk IMPORT/EXPORT operations
FOLDER	No		Indicates that the name specified by the quotedUrl (see below) refers to a folder which contains a set of files to be imported.
FOLDERS	No		Used for bulk import operations (see 11.10). Indicates that name specified by the quotedUrl (see below) refers to a folder, which contains a set of sub-folders to be imported. The name of each sub-folder is the name of the table where the imported data from the files in the sub-folder will be written.
url			see quotedUrl
quotedUrl	Yes		Specifies the location of the data to be imported using the following format: 'node://<RDP node>/< path name>' where <RDP node> is the RDP node name where the data to be imported is located <path name> is the relative path name to the location of the data (relative to the setting for the PATH property, see 11.4) Examples: With PATH set to the default "/var/tmp/rapids": 'node://db1/data' specifies that the data is to be imported from the directory "/var/tmp/rapids/data" on RapidsDB node "db1" For a custom IMPEX Connector with PATH set to "/": 'node://db1/data/log1.csv'

			specifies that the data is to be imported from the file “/data/log1.csv” on RapidsDB node “db1”
quotedSpec	No		See quotedUrl
transformerName	No		Not supported for this release
properties	No		Connector-defined properties (as key = value pairs) for the operation. By convention, explicitly specified properties override properties of the same name in the Connector (see 11.4 for the list of Properties).

Below are some examples of import references:

- `'node://db1/data/table1.csv'`

As no Connector name is specified, this import reference is for the default import Connector named “IMPORT” (see 11.9). The file name specified, “data/table1.csv” is relative to the PATH Property for the Connector, which for the “IMPORT” Connector is “/var/tmp/rapids/” (unless the PATH Property for the “IMPORT” Connector is changed – see 11.9.3), and so the fully qualified path name is “/var/tmp/rapids/data/table1.csv” on RapidsDB cluster node “db1”.

- `CSV:: 'node://db1/data/table1.csv' WITH DELIMITER='|'`

Specifies that the IMPEX Connector named “CSV” should be used and so the file name “data/table1.csv” will be relative to the PATH Property for the “CSV” Connector. For example if the “CSV” Connector has the PATH Property set to '/' (root directory), then the specified file name would get resolved as “/data/table1.csv”. The field delimiter is set to the pipe character '|’.

- `FOLDER 'node://db1/data/table1' WITH DELIMITER='|', HEADER`

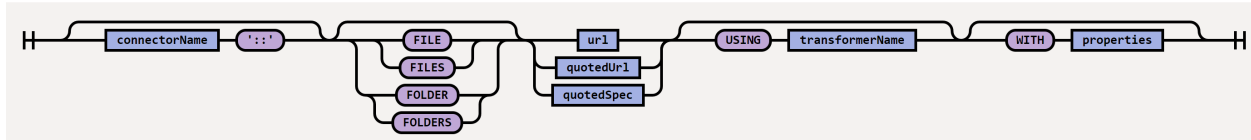
Using the default “IMPORT” Connector, the path for the folder “data/table1” would get resolved to “/var/tmp/rapids/data/table1” on RapidsDB cluster node “db1”. The field delimiter would be set to the pipe character and the HEADER option is set to indicate that the data file has a header record

- `FOLDERS CSV:: 'node://db1/data/tpch'`

For a bulk import, specifies that the IMPEX Connector named “CSV” should be used and so the folder name “data/tpch” will be relative to the PATH Property for the “CSV” Connector. For example if the “CSV” Connector has the PATH Property set to '/' (root directory), then the specified folder name would get resolved as “/data/tpch”.

11.7 EXPORT References

The user specifies the location and formatting information for the data to be exported using an Export Reference:



The table below describes each option:

Option	Required?	Default	Description
connectorName	No	EXPORT	The name of an existing IMPEX Connector
FILE	No	FILE	Indicates that the name specified by the quotedUrl (see below) refers to a file where the data for the table to be exported will be written.
FILES	No	FILES	Used for bulk export operations (see 11.12). Indicates that name specified by the quotedUrl (see below) refers to a folder which will contain the files for the set of tables being exported.
FOLDER	No		Indicates that the name specified by the quotedUrl (see below) refers to a folder where a file will be created that will store the data from the table being exported.
FOLDERS	No		Used for bulk export operations (see 11.12). Indicates that name specified by the quotedUrl (see below) refers to a folder where the sub-folders for the exported tables will be created (if needed) using the table name for the sub-folder name. Each sub-folder will then have a file created in it that will store the data from the table being exported
url	No		See quotedUrl below
quotedUrl	Yes		Specifies the location for the exported data using the following format: 'node://<RDP node>/< path name>' where <RDP node> is the RDP node name where the data to be imported is located <path name> is the relative path name to the location of the data (relative to the setting for the PATH property, see 11.4) With PATH set to the default "/var/tmp/rapids":

			<p>'node://db1/data' specifies that the data is to be exported to the directory “/var/tmp/rapids/data” on RapidsDB node “db1”</p> <p>For a custom IMPEX Connector with PATH set to “/”: 'node://db1/data/log1.csv' specifies that the data is to be exported to the file “/data/log1.csv” on RapidsDB node “db1”</p>
quotedSpec	No		See quotedUrl above
transformerName	No		Not supported for this release
properties	No		Connector-defined properties (as key = value pairs) for the operation. By convention, explicitly specified properties override properties of the same name in the Connector (see 11.4 for the list of Properties).

Below are some examples of export references:

- 'node://db1/data/table1.csv'

As no Connector name is specified, this import reference is for the default export Connector named “EXPORT” (see 11.8). The file name specified, “data/table1.csv” is relative to the PATH Property for the Connector, which for the “EXPORT” Connector is “/var/tmp/rapids/”(unless the PATH Property for the “IMPORT” Connector is changed – see 11.8.3), and so the fully qualified path name is “/var/tmp/rapids/data/table1.csv” on RapidsDB cluster node “db1”.

- CSV:: 'node://db1/data/table1.csv'

Specifies that the IMPEX Connector named “CSV” should be used and so the file name “data/table1.csv” will be relative to the PATH Property for the “CSV” Connector. For example if the “CSV” has Connector has the PATH Property set to '/' (root directory), then the specified file name would get resolved as “/data/table1.csv”.

- FOLDER 'node://db1/data/table1'

Using the default “EXPORT” Connector, the path for the folder “data/table1” would get resolved to “/var/tmp/rapids/data/table1” on RapidsDB cluster node “db1”.

- FOLDERS CSV:: 'node://db1/data/tpch_files'

For a bulk export, specifies that the IMPEX Connector named “CSV” should be used and so the folder name “data/tpch” will be relative to the PATH Property for the “CSV” Connector. For example, if the “CSV” has Connector has the PATH Property set to '/' (root directory), then the specified folder name would get resolved as “/data/tpch”.

11.8 Default IMPORT and EXPORT Connectors

11.8.1 Usage

RapidsDB comes with two built-in Connectors named “IMPORT” and “EXPORT”, that are used when an import reference (see 11.6) or an export reference (see 11.7) does not specify a Connector name. For example, 'node://db1/data/table1.csv'.

11.8.2 Default Properties

By default, the “IMPORT” and “EXPORT” Connectors have the following IMPEX Connector Properties (see 11.4):

Key:	Value
FORMAT	'CSV'
PATH	'/var/tmp/rapids'
ERROR_PATH	'/var/tmp/rapids_errors'
ERROR_LIMIT	10
BACKUP	false
CHARSET	'UTF-8'
DELIMITER	','
ENCLOSED_BY	''''
ESCAPE_CHAR	'\'
FILTER	'*.*'
GUESS	false
HEADER	false
TERMINATOR	'\n'
TRAILING	false

11.8.3 Changing the IMPEX Properties for the “IMPORT” and “EXPORT” Connectors

The user can change any of the properties for the “IMPORT” or “EXPORT” Connectors by dropping the Connector and then recreating the Connector with the same name.

NOTE: If the “IMPORT” or “EXPORT” Connector is dropped, it must be recreated with the same name because when doing an import or export operation, RapidsDB will attempt to use a Connector named “IMPORT” when an import reference (see 11.6) does not specify a Connector name, and similarly, the system will attempt to use a Connector named “EXPORT” when an export reference (see 11.7) does not specify a Connector name. If the system cannot find the relevant Connector then the import or export operation will fail.

One of the most common Properties to change would be the “PATH” property, to set an alternate default root path name. For example, if all data files will come from the folder “/data” then the default IMPORT Connector could be changed as shown below:

```
rapids > drop connector import;
0 row(s) returned (0.13 sec)
rapids > create connector import type impex with PATH='/data';
0 row(s) returned (2.37 sec)
```

The following import examples assume that the default “IMPORT” Connector was not reconfigured, and that an IMPEX Connector named “CSV” was created with the “PATH” Property set to the root directory (“/”):

```
rapids > create connector csv type impex with PATH='/data';
0 row(s) returned (2.09 sec)
SELECT * FROM ('node://db1/data/table1.csv');
```

This command would result in data being read from the file “table1.csv” in the directory “/var/tmp/rapids/data” on RapidsDB node “db1” using the default “IMPORT” Connector.

```
SELECT id, name FROM (CSV:: 'node://db1/data/table1.csv') AS t(id integer, name varchar) WHERE
t.name='BorayData';
```

This command would select the data for the first two fields from the file “table1.csv” using the Connector named “CSV” where the file “table1.csv” resides on the RapidsDB cluster node “db1” in the directory “/data” (because PATH='/'), where the “name” field is the string 'BorayData'.

The following export examples assume that the default “EXPORT” Connector was not reconfigured:

```
SELECT * FROM table1 TO 'node://db1/data/table1.csv';
```

This command would append the contents of the table named “table1” to the file “table1.csv” using the default EXPORT Connector where the file “table1.csv” resides in the directory “/var/tmp/rapids/data” on the RapidsDB cluster node “db1” and where the delimiter is ','.

```
SELECT * FROM table1 TO CSV:: 'node://db1/data/table1.csv';
```

In this example the EXPORT reference specifies the Connector name “CSV”, and so the default “EXPORT” Connector would not be used.

11.9 IMPORT using SELECT and INSERT

11.9.1 IMPORT Table Expressions

A table expression has now been extended to also include an import reference:



This means that an import reference (see 11.6) can now appear anywhere in a SELECT or INSERT statement where a regular table reference can appear, and it also means that the user can provide an alias name for the import reference, and can also specify a subset of the columns (fields) to be imported. For example:

```
SELECT * FROM ('node://db1/data/table1.csv');
```

The hilited text above is an import reference which replaces the usual table reference.

Section 11.9.2 provides examples of using an import reference with SELECT and INSERT statements.

11.9.2 IMPORT using a SELECT statement

11.9.2.1 Overview

The user can import the data as part of a regular SELECT statement, where the usual table reference is replaced with an import reference. In the following examples the hilited text is the import reference.

Example 1:

```
SELECT * FROM (FILE 'node://db1/data/table1.csv');
```

This command would result in data being read from the file “table1.csv” in directory “var/tmp/rapids/data” (the default PATH) on RapidsDB node “db1” using the default “IMPORT” IMPEX Connector (see 11.8).

Example 2:

```
SELECT id, name FROM (FILE 'node://db1/data/table1.csv' WITH PATH='/') AS t(id integer, name varchar) WHERE t.name='Jones';
```

This command would select the data for the first two fields from the file “table1.csv” using the default IMPEX Connector (see 11.8) where the file “table1.csv” resides on the RapidsDB cluster node “db1” in the directory “/data” (because PATH='/'), where the field delimiter is the character '|', with the column names “id” and “name” being used for the first two fields and where the “name” field is the string 'Jones'.

Example 3:

```
SELECT * FROM (CSV:: FOLDER 'node://db1/data/table1');
```

This command would select all of the data from the files in the folder “table1” using the IMPEX Connector named “CSV” where the folder “table1” resides on the RapidsDB cluster node “db1” in the directory “/data” (because PATH='/’ for the “CSV” IMPEX Connector), and where the field

delimiter is the character '|'. The column headings would be the usual default column headings for a result set, which are “COL1”, “COL2” ... etc.

See sections 11.9.2.10 and 11.9.2.11 for more detailed examples using SELECT statements.

11.9.2.2 Column Naming Using Default Column Names

If there are no column names associated with the input data (see the following sections for assigning column names), then the IMPEX Connector will use the default column names used by the RapidsDB Execution Engine which are “COL1”, “COL2”, etc.

Example:

```
rapids > SELECT * FROM ('node://db1/SFSMALL/region.csv') LIMIT 1;
COL1  COL2          COL3
----  ----          ----
1      UNITED STATES adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg

1 row(s) returned (0.06 sec)
```

11.9.2.3 Column Naming Using AS clause

The user can also specify the column names to be associated with each of the input fields using the “AS <tableAlias><(columnAliases)>” clause as shown in the example below:

```
rapids > SELECT * FROM ('node://db1/SFSMALL/region.csv') AS (r_regionkey ,r_name
,r_comment) LIMIT 1;

R_REGIONKEY  R_NAME          R_COMMENT
-----
1              UNITED STATES  adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg

1 row(s) returned (0.05 sec)
```

11.9.2.4 Column Naming Using HEADER option

It is common with csv files for the first record of the file to be a header record that contains a list of the column names. The IMPEX Connector allows this by setting the “HEADER” option to true. For example:

```
rapids > SELECT * FROM ('node://db1/SFSMALL/regionPipe.csv' WITH DELIMITER='|',
HEADER) LIMIT 1;

R_REGIONKEY  R_NAME          R_COMMENT
-----
1              UNITED STATES  adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg

1 row(s) returned (0.05 sec)
```

11.9.2.5 Column Data Typing Using GUESS Property

If no column data types are assigned to the columns (see sections 11.9.2.5, 11.9.2.6 and 11.9.2.8) then the IMPEX Connector will use the “GUESS” Property (see 11.4) to determine the data types as follows:

1. GUESS=FALSE (default) – the IMPEX Connector will treat all columns where the data type is not explicitly specified (see 11.9.2.6 and 11.9.2.8 for specifying data types) as polymorphic strings, which are strings that will be automatically cast into the appropriate data type based on the expression where the column is referenced in a query. For example,

```
SELECT * FROM ('node://db1/SFSMALL/region.csv') WHERE COL1>10;
```

In this example, the first column will be automatically cast to an integer

In the following example, the same field will be left as a string and not cast:

```
SELECT * FROM ('node://db1/SFSMALL/region.csv') WHERE COL1='10A';
```

If the column contains values that cannot be cast to the required data type, then an error will be returned.

2. GUESS=TRUE – for any columns where the data type is not explicitly specified (see 11.9.2.6 and 11.9.2.8 for specifying data types) the IMPEX Connector will examine a sample of the data from the file and use that sample to determine what is the best data type that fits each column in the data. If the data is uniform across the entire file, then the IMPEX Connector will generally assign the correct data type, but if the data is not uniform then the IMPEX Connector could assign the wrong data type which could result in a data type conversion error when querying the data. For example, if a field had mostly integer values, but there were a few values that were alphanumeric, then the IMPEX Connector could assign a data type of INTEGER to the column because in the data sample that was read to determine the data types, all of the values for that field were integers. This could result in an error when querying the column associated with that field. For example, in the query below, the IMPEX Connector assigned a data type of INTEGER to the column “r_regionkey”, but the query failed when looking for an alphanumeric value:

```
rapids > SELECT * FROM ('node://db1/SFSMALL/regionPipe.csv' WITH
HEADER,delimiter='|',GUESS) where r_regionkey='4A';
Unexpected Exception:
Line 1 position 90: Unresolved operator or function name: =(FastInteger, LiteralString)
```

The CAST function can be used to address this issue:

```
rapids > SELECT * FROM ('node://db1/SFSMALL/regionPipe.csv' WITH
HEADER,delimiter='|',GUESS) where cast(r_regionkey as varchar)='4A';
0 row(s) returned (0.04 sec)
```

Example 1:

File: /var/tmp/SFSMALL/partsupp.csv:

```
1,1,42,562.15,unsure
1,2,640,974.09,reliable
1,3,720,550.17,dishonest
1,4,644,461.39,weak
...
```

```
rapids > SELECT * FROM ('node://db1/SFSMALL/partsupp.csv' WITH GUESS)
LIMIT 1;
  COL1    COL2    COL3    COL4    COL5
  ----    ----    ----    ----    ----
      1      1      42     562.15 unsure
1 row(s) returned (0.07 sec)
```

The following data types will be assigned:

Column	Data type
COL1	integer
COL2	integer
COL3	integer
COL4	decimal
COL5	varchar

Example 2:

In this example the data type for the fourth column was specified using the “AS” clause (see 11.9.2.6), and all other data types will be imputed by the Connector:

```
rapids > SELECT * FROM ('node://db1/SFSMALL/partsupp.csv' WITH GUESS)
AS (COL1,COL2,COL3,COL4 float,COL5) LIMIT 1;
  COL1    COL2    COL3    COL4    COL5
  ----    ----    ----    ----    ----
      1      1      42     562.15 unsure
1 row(s) returned (0.05 sec)
```

The following data types will be assigned:

Column	Data type
COL1	integer
COL2	integer
COL3	integer
COL4	float
COL5	varchar

11.9.2.6 Column Data Typing Using AS clause

In addition to using the AS clause to name columns, the user can also use the AS clause to specify the data types to be used for the columns. For example:

```
rapids > SELECT * FROM ('node://db1/SFSMALL/region.csv') AS (r_regionkey integer,
r_name varchar, r_comment varchar) LIMIT 1;
  R_REGIONKEY R_NAME          R_COMMENT
  -----
          1 UNITED STATES  adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg
1 row(s) returned (0.05 sec)
```

The following data types will be assigned:

Column	Data type
r_regionkey	integer
r_name	varchar
r_comment	varchar

It is also possible to just specify the data types for some of the columns and let the IMPEX Connector assign the other data types by setting the “GUESS” Property to TRUE. In the example below, the data types for two of the columns were specified and the “GUESS” Property was set TRUE so that the Impex Connector would assign the data types for the other columns:

```
rapids > SELECT * FROM ('node://db1/SFSMALL/customer.csv' WITH GUESS) AS
(c_custkey integer, c_name, c_address, c_nationkey, c_phone, c_acctbal
decimal, c_mktsegment, c_comment) LIMIT 1;
  C_CUSTKEY C_NAME          C_ADDRESS    C_NATIONKEY  C_PHONE    C_ACCTBAL
C_MKTSEGMENT  C_COMMENT
  -----
          0 Richardson  Market      3            111        7994.73
FURNITURE      negative
1 row(s) returned (0.05 sec)
```

The following data types will be assigned:

Column	Data type
c_custkey	integer
c_name	varchar
c_address	varchar
c_nationkey	integer
c_phone	integer
c_acctbal	decimal
c_mktsegment	varchar
c_comment	varchar


```

R_REGIONKEY R_NAME          R_COMMENT
-----
          1 UNITED STATES  adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg

1 row(s) returned (0.05 sec)

```

11.9.2.9 RAW Data Format

In situations where a data file contains data that is completely unknown, such as where the field delimiter is not known, the user can use the “RAW” format option to import the data as a single VARCHAR column. The data can then be viewed and further processing can then be done based on the actual data. This format is also useful for fetching miscellaneous text files from locations within the RapidsDB cluster.

Example:

```

rapids > SELECT * FROM ('node://db1/SFSMALL/region.csv' WITH FORMAT='RAW');
RAW
---
1,UNITED STATES,adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg
2,NORTH AMERICA,aldkjlakngkjangkjnfkngakldflkadjlkajdlkajd
3,EUROPE,dxldgkzcsoisjoicnkjebfjhgwgrygwuihvokjdgojdvps
4,SOUTH AMERICA,csvbdcavbscdbvacdhvjhxdkgnlkgflfglksdnja shc asbvda
5,ASIA,i4y5qiuyrqghrushfxhghirohtiehtaytaiuerytaiurt

5 row(s) returned (0.04 sec)

```

```

rapids > SELECT * FROM ('node://db1/var/log/dmesg' WITH
PATH='/',FORMAT='RAW') LIMIT 2;
RAW
---
[ 0.000000] Initializing cgroup subsys cpuset
[ 0.000000] Initializing cgroup subsys cpu

2 row(s) returned (0.04 sec)

```

11.9.2.10 SELECT FROM FILE

The following examples illustrate the use of an IMPEX Connector for importing data directly from a file to be used in a SELECT statement. As “FILE” is the default option it is not necessary to specify that option when referencing a file.

Example 1:

Select from the file “text/lead_trail_blanks.csv” which is in the folder “/var/tmp/rapids”.

Below is the contents of the file “lead_trail_blanks.csv”:

```

[rapids@db1 text]$ cat /var/tmp/rapids/text/lead_trail_blanks.csv

```

```
1,    4 leading blanks,3 trailing blanks  ,1
2,A2345678901234567890,A1234567890123456789,2
3,"    4 leading blanks","3 trailing blanks  ",3
```

The SELECT command below selected all of the data from the file “lead_trail_blanks.csv” using the default “IMPORT” Connector where the file “lead_trail_blanks.csv” resides on the RapidsDB cluster node “db1” in the folder “/var/tmp/rapids/text”, and where the field delimiter is the character ',' (this is the default for “IMPORT” Connector). The column headings would be the usual default column headings for a result set, which are “COL1”, and “COL2”.

```
rapids > select * from ('node://db1/text/lead_trail_blanks.csv');
  COL1 COL2                COL3                COL4
----  ----                ----                ----
    1    4 leading blanks  3 trailing blanks    1
    2 A2345678901234567890 A1234567890123456789  2
    3    4 leading blanks  3 trailing blanks    3

3 row(s) returned (0.07 sec)
rapids > select char_length(col3) from
('node://db1/text/lead_trail_blanks.csv');
 [1]
---
   20
   20
   20

3 row(s) returned (0.06 sec)
```

Example 2:

Selecting from a file where the delimiter is the pipe character ('|') and the file includes a header record with the column names to use.

Below is the content of the file:

```
[rapids@db1 rapids]$ cat /var/tmp/rapids/SFSMALL/regionPipe.csv
R_REGIONKEY|R_NAME|R_COMMENT
1|UNITED STATES|adknladnganfbmanlgnalkfnglkaajglkafjglksjfglkajfgkjadg
2|NORTH AMERICA|aldkjlakngkjanganjfnfngakldflkadjlkajdlkajd
3|EUROPE|dxldgkzcsoisjoicnkjebfjhqwrygwuihvokjdgojdvps
4|SOUTH AMERICA|csvbdcavbscdvavcdhvjhxdkgnlkgflglksdnja shc asbvda
5|ASIA|i4y5qiuyrqghrushfxhghirohtiehtaytaiuerytaiurt
```

This command below selected all of the data from the file “regionPipe.csv” using the default “IMPORT” Connector, where the file “regionPipe.csv” resides on the RapidsDB cluster node “db1” in the folder

“/var/tmp/rapids/SFsmall”, and where the field delimiter is the character '|'. The file includes a header record which has the column names to use.

```
rapids > SELECT * FROM ('node://db1/SFsmall/regionPipe.csv' WITH
HEADER,delimiter='|');
```

R_REGIONKEY	R_NAME	R_COMMENT
1	UNITED STATES	adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg
2	NORTH AMERICA	aldkjlakngkjangkijnfkngakldflkadjlkajdkajd
3	EUROPE	dxldgkzcsoisjoicnkjebfjhqwrygwuihvokjdgojdvps
4	SOUTH AMERICA	csvbdcavbscdbvacdhvjhxdkgnlkgfglksdnja shc asbvda
5	ASIA	i4y5qiuyrqghrushfxhghirohtiehtaytaiurytaiurt

5 row(s) returned (0.05 sec)

Example 3:

This example shows the use of a custom IMPEX Connector named “CSV_HEADER” which has the default delimiter set to the pipe character, and with HEADER set true:

```
rapids > CREATE CONNECTOR CSV_HEADER TYPE IMPEX WITH DELIMITER='|', HEADER;
0 row(s) returned (2.21 sec)
```

```
rapids > SELECT * FROM (csv_header:: 'node://db1/SFsmall/regionPipe.csv');
```

R_REGIONKEY	R_NAME	R_COMMENT
1	UNITED STATES	adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg
2	NORTH AMERICA	aldkjlakngkjangkijnfkngakldflkadjlkajdkajd
3	EUROPE	dxldgkzcsoisjoicnkjebfjhqwrygwuihvokjdgojdvps
4	SOUTH AMERICA	csvbdcavbscdbvacdhvjhxdkgnlkgfglksdnja shc asbvda
5	ASIA	i4y5qiuyrqghrushfxhghirohtiehtaytaiurytaiurt

5 row(s) returned (0.05 sec)

Example 4:

This example shows data filtering by using a predicate (“r_regionkey=4”) on the input data, and illustrates the use of the “AS” clause for defining the column names.

```

rapids > SELECT * FROM ('node://db1/SFSMALL/region.csv') AS
region(r_regionkey integer, r_name varchar) WHERE r_regionkey=4;
  R_REGIONKEY R_NAME
  -----
          4 SOUTH AMERICA

1 row(s) returned (0.10 sec)

```

Example 5:

This example illustrates the use of a complex query to filter the data, where the query includes both predicates on the input data along with a join to another MOXE table:

```

rapids > SELECT * FROM (FILE 'node://db1/SFSMALL/customer.csv') AS IMPORTED
(c_custkey INTEGER, c_name VARCHAR, c_address VARCHAR, c_nationkey INTEGER,
c_phone VARCHAR, c_acctbal DECIMAL, c_mktsegment VARCHAR, c_comment
VARCHAR) WHERE c_acctbal > 3000 and c_nationkey=22 AND EXISTS (SELECT * FROM
vip_customer WHERE vip_customer.c_custkey = IMPORTED.c_custkey) ;
  C_CUSTKEY C_NAME      C_ADDRESS      C_NATIONKEY C_PHONE      C_ACCTBAL
C_MKTSEGMENT  C_COMMENT      [9]
  -----
-----
          20 Egerton    Main           22 111      3815.45
AUTOMOBILE    satisfied      1
          28 Stringer  Mission       22 111      8516.37
FURNITURE     dissatisfied   1
          61 Riley     Turk          22 111      6320.39
MACHINERY     angry         1

3 row(s) returned (0.18 sec)

```

Example 6:

This example illustrates the use of the LIKE clause to provide the column names and column data types for the input data. In this example, the table definition for the table “moxe.customer” is being used:

```

rapids > SELECT * FROM ('node://db1/SFSMALL/customer.csv') AS IMPORTED LIKE
moxe.customer WHERE IMPORTED.c_acctbal > 3000 and c_nationkey=22 AND EXISTS
(SELECT * FROM vip_customer WHERE vip_customer.c_custkey =
IMPORTED.c_custkey) ;
  C_CUSTKEY C_NAME      C_ADDRESS      C_NATIONKEY C_PHONE      C_ACCTBAL
C_MKTSEGMENT  C_COMMENT      [9]
  -----
-----
          20 Egerton    Main           22 111      3815.45
AUTOMOBILE    satisfied      1

```

	28	Stringer	Mission		22	111	8516.37
FURNITURE		dissatisfied		1			
	61	Riley	Turk		22	111	6320.39
MACHINERY		angry		1			

3 row(s) returned (0.18 sec)

NOTE: When filtering data using predicates, it is highly recommended that the AS clause includes the data types for all columns as shown in examples 3, and 4 above or the LIKE clause is used to provide the column definitions from an existing table as shown in example 5 above.

11.9.2.11 SELECT FROM FOLDER

The following examples illustrate the use of an IMPEX Connector for importing data directly from a folder to be used in a SELECT statement. The files to be accessed from the folder are controlled by the “FILTER” property, which by default is set to ‘*.*’ (to read all files).

Example 1:

This example shows reading all of the files from a folder:

Folder /var/tmp/rapids/tpch_small/region:

```
[rapids@db1 region]$ ls /var/tmp/rapids/tpch_small/region
region.csv
```

```
rapids > SELECT * FROM (FOLDER 'node://db1/tpch_small/region');

COL1      COL2          COL3
----      -
1         UNITED STATES adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg
2         NORTH AMERICA aldkjllakngkjjangkijnfkngakldflkadjlkajdlkajd
3         EUROPE         dxldgkzcsoisjoicnkjebfjhqwgrgygwuihvokjdgojdvps
4         SOUTH AMERICA csvbdcavbscdbvacdhvjhxdkgnlkgflglksdnja shc asbvda
5         ASIA          i4y5qiuyrqghrushfxhghirohtiuehtaytaiuerytaiurt

5 row(s) returned (0.49 sec)
```

Example 2:

In this example a “FILTER” option is used to only import files ending in “.csv”. The folder for the example below includes two “.csv” files which will get loaded and one file named “junk” that will be ignored:

Folder /var/tmp/rapids/SFSMALL/region:

```
[rapids@db1 region]$ ls /var/tmp/rapids/SFSMALL/region
junk region1.csv region2.csv
```

```

rapids > SELECT * FROM (FOLDER 'node://db1/SFSMALL/region' WITH
FILTER='*.csv');
COL1    COL2          COL3
----    ----          ----
1        UNITED STATES adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg
2        NORTH AMERICA aldkjllakngkjanganjnfkngakldflkadjlkajdlkajd
3        EUROPE         dxldgkzcsoisjoicnkjebfjhqwrygwuihvokjdgoidvpds
4        SOUTH AMERICA csvbdcavbscldbvacdhvjhxdkgnlkgflglksdnja shc asbvda
5        ASIA          i4y5qiuyrqghrushfxhghirohtiehtaytaiurytaiurt

5 row(s) returned (0.05 sec)

```

Example 3:

This example shows the use of a custom Connector named “CSV_HEADER” which has the “HEADER” Property set true and the delimiter character set to the pipe character:

```

rapids > SELECT * FROM (csv_header:: FOLDER 'node://db1/SFSMALL/regionPipe');
R_REGIONKEY  R_NAME          R_COMMENT
-----
1            UNITED STATES  adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg
2            NORTH AMERICA aldkjllakngkjanganjnfkngakldflkadjlkajdlkajd
3            EUROPE         dxldgkzcsoisjoicnkjebfjhqwrygwuihvokjdgoidvpds
4            SOUTH AMERICA csvbdcavbscldbvacdhvjhxdkgnlkgflglksdnja shc asbvda
5            ASIA          i4y5qiuyrqghrushfxhghirohtiehtaytaiurytaiurt

5 row(s) returned (0.08 sec)

```

Example 4:

This example illustrates the use of a predicate to filter the data, and also includes the “AS” clause to specify the column names and data types:

```

rapids > SELECT * FROM (FOLDER 'node://db1/SFSMALL/region' WITH
FILTER='*.csv') AS region(r_regionkey integer, r_name varchar) WHERE
r_regionkey=4;
R_REGIONKEY R_NAME
-----
4 SOUTH AMERICA

1 row(s) returned (0.07 sec)

```

Example 5:

This example is the same as the previous example except a “LIKE” clause is used in place of the “AS” clause to specify the column names and data types:


```

rapids > SELECT * FROM (FOLDER 'node://db1/SFSMALL/region' WITH
FILTER='*.csv') AS region LIKE moxe.region WHERE r_regionkey=4;
  R_REGIONKEY R_NAME          R_COMMENT
  -----
          4 SOUTH AMERICA  csvbdcavbscdbvacdhvjhxdkgnlkgflglsdnja shc
asbvda

1 row(s) returned (0.09 sec)

```

NOTE: When filtering data using predicates, it is highly recommended that the AS clause includes the data types for all columns in the AS clause as shown in example 2 above or includes the LIKE clause as shown in example 4 above.

11.9.2.12 INSERT ... SELECT

When doing an INSERT ... SELECT, the data types the IMPEX Connector will use when reading the data associated with any of the files specified in the SELECT statement will be controlled by the column data types assigned to that file using either the “AS” clause (see 11.10.2.6) or the “LIKE” clause (see 11.10.2.8). If the data types are not specified then, by default, the IMPEX Connector will

Example 1:

This example shows an insert into the MOXE table named “region” of the data from all of the files ending in “.csv” in the folder “region” using the default “IMPORT” Connector where the folder “region” resides on the RapidsDB cluster node “db1” in the folder “/var/tmp/rapids/SFSMALL”.

```

rapids > create table moxe.REGION (
  >   r_regionkey integer NOT NULL,
  >   r_name varchar(25),
  >   r_comment varchar(152)
  > );
0 row(s) returned (0.09 sec)
rapids > INSERT INTO moxe.region SELECT * FROM (FOLDER 'node://db1/SFSMALL/region'
WITH FILTER='*.csv');
0 row(s) returned (0.08 sec)
rapids > select * from moxe.region;
  R_REGIONKEY R_NAME          R_COMMENT
  -----
          1 UNITED STATES  adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg
          2 NORTH AMERICA  aldkjlakngkjankjnfkngakldflkadjlakjdlkajd
          3 EUROPE            dxldgkzcsoisjoienkjebfjhqwgrygwuihvokjdgoidvps
          4 SOUTH AMERICA  csvbdcavbscdbvacdhvjhxdkgnlkgflglsdnja shc asbvda
          5 ASIA            i4y5qiuyrqghrushfxhghirohtiehtaytaiuert

5 row(s) returned (0.05 sec)

```

Example 2:

This example demonstrates selecting a subset of the fields from the input file by using the “AS” clause to name the fields from the input file that are to be imported, and also the use of a predicate to filter the data being inserted:

```

rapids > create table moxe.REGION2 (
  >   r_regionkey integer NOT NULL,
  >   r_name varchar(25)
  > );
0 row(s) returned (0.10 sec)
rapids > INSERT into moxe.region2 SELECT r_regionkey, r_name FROM
('node://db1/SFSMALL/region.csv') AS r(r_regionkey integer, r_name varchar)
WHERE r.r_regionkey<3;
0 row(s) returned (0.08 sec)
rapids > select * from moxe.region2;
  R_REGIONKEY R_NAME
  -----
          1 UNITED STATES
          2 NORTH AMERICA

2 row(s) returned (0.05 sec)

```

Example 3:

This command also selects a subset of the fields to be inserted and uses a header record in the input file to name the fields in the input file which can then be used to specify which fields are to be imported:

```

rapids > truncate region2;
0 row(s) returned (0.05 sec)
rapids > INSERT into moxe.region2 SELECT r_regionkey, r_name FROM
('node://db1/SFSMALL/regionPipe.csv' WITH DELIMITER='|', HEADER);
0 row(s) returned (0.08 sec)
rapids > select * from region2;
  R_REGIONKEY R_NAME
  -----
          1 UNITED STATES
          2 NORTH AMERICA
          3 EUROPE
          4 SOUTH AMERICA
          5 ASIA

5 row(s) returned (0.05 sec)

```

Example 4:

This is the same as the previous example except this example uses a custom Connector named “CSV_HEADER” which has the “HEADER” Property set “true” and the “DELIMITER” set to the pipe character:

```

rapids > truncate region2;
0 row(s) returned (0.05 sec)
rapids > INSERT into moxe.region2 SELECT r_regionkey, r_name FROM
(csv_header::'node://db1/SFSMALL/regionPipe.csv' );
0 row(s) returned (0.08 sec)

```

```

rapids > select * from region2;
  R_REGIONKEY R_NAME
  -----
      1 UNITED STATES
      2 NORTH AMERICA
      3 EUROPE
      4 SOUTH AMERICA
      5 ASIA

5 row(s) returned (0.05 sec)

```

11.9.2.13 CREATE AS SELECT

The user can create a new table us the CREATE <table> as SELECT ... clause, where the SELECT can include an import reference. The column names and data types will follow the rules specified in sections 11.10.2.2 to 11.10.2.8.

NOTE:

1. When creating a MOXE table, if the “PARTITION BY” clause is not specified then the table will get created as a reference (replicated) table, where there will be a full copy of the data on every node in the RapidsDB cluster where the associated MOXE Connector is running. If the “PARTITION BY” clause is specified, then there will be one copy of the data distributed across all of the nodes in the RapidsDB cluster where the associated MOXE Connector is running, and the data will be partitioned using the column(s) specified in the “PARTITION BY” clause. Refer to sections 10.2.1 and 10.2.2 for more information on partitioned and reference tables. Examples 3 and 4 below shows the use of the “PARTITION BY” clause.
2. When creating a table managed by any of the Connectors except for MOXE, the “PARTITION BY” clause is not currently supported, and so the “CREATE TABLE” command will either fail with an error or the “PARTITION BY” clause will get ignored and the table will get created as a non-partitioned table.

Example 1:

This command creates a replicated MOXE table named “customer1” with the data from the file “customer.csv”. The column names are the default column names, “COL1”, “COL2” etc and the data types would all be set to VARCHAR because the “GUESS” property (see 11.4) is set false by default.

```

rapids > create table moxe.customer1 as select * from
('node://db1/SFSMALL/customer.csv');

0 row(s) returned (0.26 sec)

rapids > describe table customer1;

TABLE_NAME    COLUMN_NAME    DATA_TYPE    ORDINAL    IS_PARTITION_KEY
IS_NULLABLE   PRECISION     SCALE COMMENT    PROPERTIES

```

```

-----
-----
CUSTOMER1    COL1          VARCHAR      0          false
true        NULL         NULL NULL     NULL
CUSTOMER1    COL2          VARCHAR      1          false
true        NULL         NULL NULL     NULL
CUSTOMER1    COL3          VARCHAR      2          false
true        NULL         NULL NULL     NULL
CUSTOMER1    COL4          VARCHAR      3          false
true        NULL         NULL NULL     NULL
CUSTOMER1    COL5          VARCHAR      4          false
true        NULL         NULL NULL     NULL
CUSTOMER1    COL6          VARCHAR      5          false
true        NULL         NULL NULL     NULL
CUSTOMER1    COL7          VARCHAR      6          false
true        NULL         NULL NULL     NULL
CUSTOMER1    COL8          VARCHAR      7          false
true        NULL         NULL NULL     NULL

8 row(s) returned (0.50 sec)

```

Example 2:

This example creates a replicated MOXE table named “customer1” with the data from the file “customer.csv”. The column names are the default column names, “COL1”, “COL2” etc and the data types are derived by looking at a sample of the data because the “GUESS” property (see 11.4) is set to true.

```

rapids > create table moxe.customer1 as select * from
('node://db1/SFSMALL/customer.csv' WITH GUESS);
0 row(s) returned (0.27 sec)
rapids > describe table customer1;
TABLE_NAME    COLUMN_NAME    DATA_TYPE    ORDINAL    IS_PARTITION_KEY
IS_NULLABLE    PRECISION    SCALE COMMENT    PROPERTIES
-----
-----
CUSTOMER1    COL1          INTEGER      0          false
true        64          NULL NULL     NULL
CUSTOMER1    COL2          VARCHAR      1          false
true        NULL         NULL NULL     NULL

```

CUSTOMER1	COL3		VARCHAR	2	false
true	NULL	NULL	NULL NULL	NULL	
CUSTOMER1	COL4		INTEGER	3	false
true	64	NULL	NULL NULL	NULL	
CUSTOMER1	COL5		INTEGER	4	false
true	64	NULL	NULL NULL	NULL	
CUSTOMER1	COL6		DECIMAL	5	false
true	17	2	NULL NULL	NULL	
CUSTOMER1	COL7		VARCHAR	6	false
true	NULL	NULL	NULL NULL	NULL	
CUSTOMER1	COL8		VARCHAR	7	false
true	NULL	NULL	NULL NULL	NULL	

8 row(s) returned (0.31 sec)

Example 3:

This example creates a partitioned MOXE table using the "PARTITION BY" clause and also shows the use of the "AS" clause to specify the column names and data types to be used in the new table:

```

rapids > CREATE TABLE moxe.special_customer AS SELECT * FROM
('node://db1/SFSMALL/customer.csv') AS IMPORTED (c_custkey INTEGER, c_name
VARCHAR, c_address VARCHAR, c_nationkey INTEGER, c_phone VARCHAR, c_acctbal
DECIMAL, c_mktsegment VARCHAR, c_comment VARCHAR) PARTITION BY HASH ON
(c_custkey) WHERE c_acctbal > 0 AND EXISTS (SELECT * FROM vip_customer WHERE
vip_customer.c_custkey = IMPORTED.c_custkey) ;
0 row(s) returned (0.41 sec)
rapids > describe table special_customer;
TABLE_NAME          COLUMN_NAME        DATA_TYPE         ORDINAL    IS_PARTITION_KEY
IS_NULLABLE        PRECISION          SCALE COMMENT        PROPERTIES
-----
SPECIAL_CUSTOMER   C_CUSTKEY          INTEGER            0          false
true               64                NULL NULL        NULL
SPECIAL_CUSTOMER   C_NAME             VARCHAR            1          false
true               NULL              NULL NULL        NULL
SPECIAL_CUSTOMER   C_ADDRESS          VARCHAR            2          false
true               NULL              NULL NULL        NULL
SPECIAL_CUSTOMER   C_NATIONKEY        INTEGER            3          false
true               64                NULL NULL        NULL
SPECIAL_CUSTOMER   C_PHONE            VARCHAR            4          false
true               NULL              NULL NULL        NULL
SPECIAL_CUSTOMER   C_ACCTBAL          DECIMAL            5          false
true               17                2 NULL          NULL
SPECIAL_CUSTOMER   C_MKTSEGMENT       VARCHAR            6          false
true               NULL              NULL NULL        NULL
SPECIAL_CUSTOMER   C_COMMENT          VARCHAR            7          false
true               NULL              NULL NULL        NULL

```

```

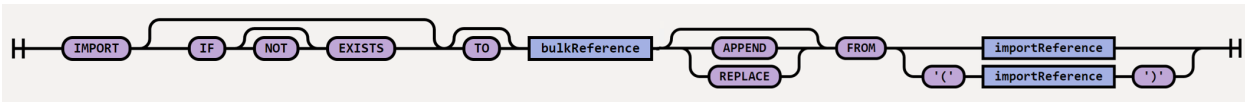
8 row(s) returned (0.31 sec)
rapids > select count(*) from special_customer;
[1]
---
75

1 row(s) returned (0.08 sec)

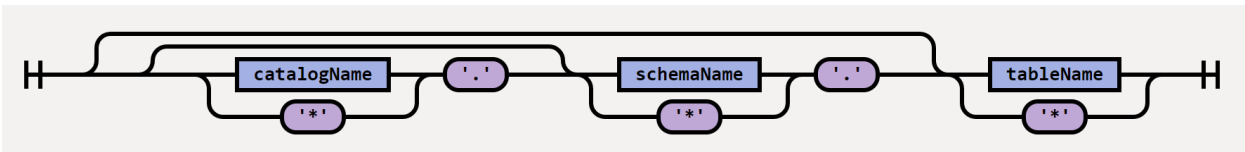
```

11.10 Bulk IMPORT

A new IMPORT statement is now supported for directly importing multiple tables with a single request:



bulkReference:



The table below describes each option:

Option	Required?	Default?	Description
IF EXISTS	No	No	Import only items whose name matches an existing table in the catalog and/or schema specified by the bulkReference (see below)
IF NOT EXISTS	No	No	Import only items whose name does <i>not</i> match an existing table in the catalog and/or schema specified by the bulkReference (see below)
APPEND	No	Yes	Append the imported data to an existing table, if any.
REPLACE	No	No	Truncate an existing table before new data is imported
bulkReference	Yes	No	Specifies the three-level (catalog, schema, table) naming for the target table(s) into which data will be imported. Wildcards may be specified (using an asterisk '*') for any of the name components. If the catalog name and/or schema name are omitted, the <i>CURRENT_CATALOG</i> and <i>CURRENT_SCHEMA</i> session settings are used, if set. The Connector may support properties (see below) which impact how the bulk reference is interpreted. NOTE: All of the tables to be imported into must be managed by the same Connector, if that is not the case then an error will be returned (see Example 4 in section 11.10.2 below for more details). To ensure

			that this does not happen it is recommended that the schema name is always specified and if this is not unique, then the catalog name should also be included.
importReference	Yes	No	An Import Reference (see 11.7) identifying the data to be imported

NOTE:

1. When doing a bulk import, if the target table does not exist then the table will get created by the Connector associated with the bulkReference (see above) using the following rules:
 - a. The column names will be the default column names described in section 11.10.2.2
 - b. The column data types will depend on the setting of the “GUESS” property as described in section 11.10.2.5. If the “GUESS” Property is set to FALSE, then all of the columns will be created as VARCHAR columns.
 - c. The table will get created with defaults for any primary keys or partitioning keys, which could result in unexpected performance or memory limits issues. For example, a MOXE Connector will create the table as a Reference table (see 10.2.2) because there is no partitioning information, and this means that each node in the RapidsDB cluster where the associated MOXE Connector is running will have a full copy of the imported data, and this could result in MOXE running out of memory. To avoid any such issues, it is highly recommended that all target tables in a bulk import are first created with the appropriate primary and partitioning keys, and then the bulk import executed against the existing tables. Alternatively, the tables can be imported individually using a CREATE ... AS SELECT statement where the primary and partitioning keys can be specified (see 11.10.2.12).

2. When doing a bulk import, if the target tables already exist, then it is recommended that the “GUESS” property is set to FALSE so that all data fields are read in as polymorphic strings (see 11.10.2.5) and then automatically cast to the data type for the column in the target table.

11.10.1 Bulk IMPORT Using FILES Option

When doing a bulk import operation the “FILES” option indicates that the path name specified in the import reference (see 11.6) refers to a folder which contains a set of files to be imported. The name of each file (minus any dot suffixes) is the name of the table where the data from the file will be written. The “FILES” option is the default and so it does not need to be specified.

Example 1:

Bulk import a set of files from the folder “/var/tmp/rapids/tpch_small_files” from RapidsDB Cluster node “db1” using the default “IMPORT” Connector (see 11.8).

Folder /var/tmp/rapids/tpch_small_files:

```
[rapids@db1 rapids]$ ls tpch_small_files
customer.csv  lineitem.csv  nation.csv  orders.csv  part.csv  partsupp.csv
region.csv  supplier.csv
```

```
rapids > IMPORT MOXE.* FROM 'node://db1/tpch_small_files' WITH GUESS;
0 row(s) returned (12.17 sec)
```

This command imported all the files from folder “/var/tmp/rapids/tpch_small_files” (“customer.csv”, “lineitem.csv”, “nation.csv”, etc) creating new tables (see “show table;” output below) of the same names (minus the “.csv” suffix) in the schema MOXE if they did not already exist and where the column names for any newly created table were set to “COL1”, “COL2”, etc, and where the data types were imputed from the data in the files (based on the “GUESS” property being set TRUE (see 11.9.2.5) (see “describe table” output below). If the tables already existed, the new data would be inserted alongside the existing data.

```
rapids > show tables;
```

CATALOG_NAME	SCHEMA_NAME	TABLE_NAME
MOXE	MOXE	CUSTOMER
MOXE	MOXE	LINEITEM
MOXE	MOXE	NATION
MOXE	MOXE	ORDERS
MOXE	MOXE	PART
MOXE	MOXE	PARTSUPP
MOXE	MOXE	REGION
MOXE	MOXE	SUPPLIER

...

```
rapids > describe table region;
```

TABLE_NAME	COLUMN_NAME	DATA_TYPE	ORDINAL	IS_PARTITION_KEY
IS_NULLABLE	PRECISION	SCALE		
REGION	COL1	INTEGER	0	false
true	64	NULL		
REGION	COL2	VARCHAR	1	false
true	NULL	NULL		
REGION	COL3	VARCHAR	2	false
true	NULL	NULL		

```
3 row(s) returned (0.28 sec)
```

Finally, the query below shows that the correct number of records were loaded into the table:

```
[rapids@db1 rapids]$ cat tpch_small_files/lineitem.csv | wc -l
3000
```



```

rapids > select count(*) from lineitem;
      [1]
      ---
      3000

1 row(s) returned (0.22 sec)

```

Example 2:

This example shows the use of the “IF NOT EXISTS” clause to restrict a bulk import to only those tables that do not exist.

Bulk import a set of files from the folder “/var/tmp/rapids/tpch_small_files” on RapidsDB Cluster node “db1”.

Folder /var/tmp/rapids/tpch_small_files:

```

[rapids@db1 rapids]$ ls tpch_small_files
customer.csv  lineitem.csv  nation.csv  orders.csv  part.csv  partsupp.csv
region.csv   supplier.csv

```

Drop the existing tables “customer” and “region”, show the existing row count for the “nation” table to show that the row count does not change after the import, and that only the “customer” and “region” tables are imported:

```

rapids > drop table customer;
0 row(s) returned (0.10 sec)
rapids > drop table region;
0 row(s) returned (0.10 sec)
rapids > show tables;
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME
-----
MOXE          MOXE         LINEITEM
MOXE          MOXE         NATION
MOXE          MOXE         ORDERS
MOXE          MOXE         PART
MOXE          MOXE         PARTSUPP
MOXE          MOXE         SUPPLIER
...
24 row(s) returned (0.28 sec)
rapids > select count(*) from nation;
      [1]
      ---
      25

```

IMPORT command with the “IF NOT EXISTS” clause

```
1 row(s) returned (0.09 sec)
rapids > IMPORT IF NOT EXISTS MOXE.* FROM 'node://db1/tpch_small_files' WITH
GUESS;
0 row(s) returned (0.70 sec)
```

The “SHOW TABLES” command shows that the “customer” and “region” tables were imported, and the row count for the “nation” table has not changed, showing that no data was imported into that table:

```
rapids > show tables;
CATALOG_NAME    SCHEMA_NAME    TABLE_NAME
-----
MOXE            MOXE           CUSTOMER
MOXE            MOXE           LINEITEM
MOXE            MOXE           NATION
MOXE            MOXE           ORDERS
MOXE            MOXE           PART
MOXE            MOXE           PARTSUPP
MOXE            MOXE           REGION
MOXE            MOXE           SUPPLIER
...
26 row(s) returned (0.27 sec)
rapids > select count(*) from nation;
 [1]
  ---
  25
1 row(s) returned (0.09 sec)
```

Example 3:

This example shows importing a set of files which were created using the EXPORT command (see 11.12.2):

```
rapids > EXPORT MOXE.* TO 'node://db1/tpch_small_file_with_headers' WITH
HEADER;
0 row(s) returned (8.33 sec)
```

Now drop the current tables and then import the files specifying the HEADER and GUESS options:

```
rapids > drop table moxe.lineitem;
0 row(s) returned (0.19 sec)
rapids > drop table moxe.orders;
0 row(s) returned (0.10 sec)
rapids > drop table moxe.customer;
```

```

0 row(s) returned (0.09 sec)
rapids > drop table moxe.supplier;
0 row(s) returned (0.09 sec)
rapids > drop table moxe.part;
0 row(s) returned (0.10 sec)
rapids > drop table moxe.partsupp;
0 row(s) returned (0.11 sec)
rapids > drop table moxe.nation;
0 row(s) returned (0.09 sec)
rapids > drop table moxe.region;
0 row(s) returned (0.09 sec)
rapids > IMPORT MOXE.* FROM 'node://db1/tpch_small_file_with_headers' WITH
HEADER, GUESS;
0 row(s) returned (16.26 sec)
rapids > show tables;
CATALOG_NAME      SCHEMA_NAME      TABLE_NAME
-----
MOXE              MOXE             CUSTOMER
MOXE              MOXE             LINEITEM
MOXE              MOXE             NATION
MOXE              MOXE             ORDERS
MOXE              MOXE             PART
MOXE              MOXE             PARTSUPP
MOXE              MOXE             REGION
MOXE              MOXE             SUPPLIER
...
31 row(s) returned (0.27 sec)
rapids > describe table region;
TABLE_NAME      COLUMN_NAME      DATA_TYPE      ORDINAL      IS_PARTITION_KEY
IS_NULLABLE      PRECISION      SCALE
-----
REGION          'R_REGIONKEY'   INTEGER         0            false
true           64            NULL
REGION          'R_NAME'        VARCHAR         1            false
true           NULL          NULL
REGION          'R_COMMENT'     VARCHAR         2            false
true           NULL          NULL
3 row(s) returned (0.27 sec)

```

Example 4:

This example shows that when the tables already exist a bulk import operation will append to the existing tables.

Current row counts:

```
rapids > select count(*) from customer;
```

```

[1]
---
75

1 row(s) returned (0.06 sec)
rapids > select count(*) from lineitem;
[1]
---
3000

1 row(s) returned (0.06 sec)
rapids > select count(*) from nation;
[1]
---
25

1 row(s) returned (0.06 sec)
rapids > select count(*) from orders;
[1]
---
750

1 row(s) returned (0.06 sec)
rapids > select count(*) from part;
[1]
---
100

1 row(s) returned (0.07 sec)
rapids > select count(*) from partsupp;
[1]
---
400

1 row(s) returned (0.06 sec)
rapids > select count(*) from region;
[1]
---
5

1 row(s) returned (0.06 sec)
rapids > select count(*) from supplier;
[1]
---
5

1 row(s) returned (0.05 sec)

```

In this example the “IF EXISTS” clause is used to only import into those tables that already exist. The “GUESS” Property is set to FALSE (by default) so that all data will be read as polymorphic strings and then cast to the data type for each column. The “FILES” option is explicitly stated (although as mentioned earlier it is not needed because it is the default), and the FILTER option is specified to only include those files with the suffix “.csv”. The new row counts show that the number of rows in each table doubled as a result of the bulk import inserting the new data:

```
rapids > IMPORT IF EXISTS MOXE.* FROM FILES 'node://db1/tpch_small_files'
WITH FILTER='*.csv';
0 row(s) returned (9.94 sec)
rapids > select count(*) from customer;
[1]
---
150

1 row(s) returned (0.06 sec)
rapids > select count(*) from lineitem;
[1]
---
6000

1 row(s) returned (0.06 sec)
rapids > select count(*) from nation;
[1]
---
50

1 row(s) returned (0.06 sec)
rapids > select count(*) from orders;
[1]
---
1500

1 row(s) returned (0.06 sec)
rapids > select count(*) from part;
[1]
---
200

1 row(s) returned (0.05 sec)
rapids > select count(*) from partsupp;
[1]
---
800

1 row(s) returned (0.06 sec)
rapids > select count(*) from region;
[1]
```

```
---
 10
1 row(s) returned (0.06 sec)
rapids > select count(*) from supplier;
 [1]
---
 10
1 row(s) returned (0.06 sec)
```

Example 5:

This command demonstrates the use of the “REPLACE” option to replace the existing data in the tables (by doing a “TRUNCATE” operation before importing the data) with new data.

Below are the current row counts for the tables:

```
rapids > select count(*) from customer;
 [1]
---
 150
1 row(s) returned (0.06 sec)
rapids > select count(*) from lineitem;
 [1]
---
 6000
1 row(s) returned (0.06 sec)
rapids > select count(*) from nation;
 [1]
---
  50
1 row(s) returned (0.06 sec)
rapids > select count(*) from orders;
 [1]
---
 1500
1 row(s) returned (0.06 sec)
rapids > select count(*) from part;
 [1]
---
  200
1 row(s) returned (0.05 sec)
rapids > select count(*) from partsupp;
 [1]
```

```

---
800

1 row(s) returned (0.06 sec)
rapids > select count(*) from region;
[1]
---
10

1 row(s) returned (0.06 sec)
rapids > select count(*) from supplier;
[1]
---
10

1 row(s) returned (0.06 sec)
rapids > IMPORT MOXE.* REPLACE FROM 'node://db1/tpch_small_files';
0 row(s) returned (10.13 sec)

```

Note that the table counts below now reflect a single copy of the data:

```

rapids > select count(*) from customer;
[1]
---
75

1 row(s) returned (0.06 sec)
rapids > select count(*) from lineitem;
[1]
---
3000

1 row(s) returned (0.06 sec)
rapids > select count(*) from nation;
[1]
---
25

1 row(s) returned (0.06 sec)
rapids > select count(*) from orders;
[1]
---
750

1 row(s) returned (0.06 sec)
rapids > select count(*) from part;
[1]
---

```

```

100

1 row(s) returned (0.06 sec)
rapids > select count(*) from partsupp;
 [1]
 ---
 400

1 row(s) returned (0.06 sec)
rapids > select count(*) from region;
 [1]
 ---
   5

1 row(s) returned (0.06 sec)
rapids > select count(*) from supplier;
 [1]
 ---
   5

1 row(s) returned (0.06 sec)

```

11.10.2 Bulk IMPORT Using FILES Option With FILTER

The FILTER property allows the user to control which files are imported in a wildcard import operation and, optionally, how table names are created from the names of imported files. The FILTER value is a character string containing a Java regular expression (a “regex”).

When performing a wildcard import, an IMPEX Connector examines each filename available from the import source. Only files whose names satisfy the FILTER regex are imported. (For a tutorial on Java regular expressions, see <https://www.oracle.com/technical-resources/articles/java/regex.html>)

Example 1:

Bulk import a set of files from the folder “/var/tmp/rapids/tpch_small_files” on RapidsDB Cluster node “db1”.

Folder /var/tmp/rapids/tpch_small_files:

```

[rapids@db1 rapids]$ ls tpch_small_files
customer.csv  lineitem.csv  nation.csv  orders.csv  part.csv  partsupp.csv
region.csv  supplier.csv

```

```

rapids > show tables;
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME
-----
RAPIDS        SYSTEM       AUTHENTICATORS
RAPIDS        SYSTEM       AUTHENTICATOR_CONFIG

```



```

RAPIDS      SYSTEM      CATALOGS
RAPIDS      SYSTEM      COLUMNS
RAPIDS      SYSTEM      CONNECTORS
RAPIDS      SYSTEM      FEDERATIONS
RAPIDS      SYSTEM      INDEXES
RAPIDS      SYSTEM      NODES
RAPIDS      SYSTEM      PATTERN_MAPS
RAPIDS      SYSTEM      QUERIES
RAPIDS      SYSTEM      QUERY_STATS
RAPIDS      SYSTEM      SCHEMAS
RAPIDS      SYSTEM      SESSIONS
RAPIDS      SYSTEM      TABLES
RAPIDS      SYSTEM      TABLE_PROVIDERS
RAPIDS      SYSTEM      USERNAME_MAPS
RAPIDS      SYSTEM      USERS
RAPIDS      SYSTEM      USER_CONFIG

```

```
18 row(s) returned (0.22 sec)
```

```
rapids > IMPORT MOXE.* FROM 'node://db1/tpch_small_files' WITH
FILTER='*.csv',GUESS;
```

```
0 row(s) returned (12.17 sec)
```

This command imported all the files from folder “/var/tmp/rapids/tpch_small_files” with a file suffix of “.csv” creating new tables (see “show tables;” output below) of the same names (minus the “.csv” suffix) in the schema MOXE if they did not already exist and where the column names for any newly created table were set to “COL1”, “COL2”, etc, and where the data types were imputed from the data in the files (based on the “GUESS” property being set TRUE (see 11.9.2.5) (see “describe table” output below). If the tables already existed, the new data would be inserted alongside the existing data.

```
rapids > show tables;
```

CATALOG_NAME	SCHEMA_NAME	TABLE_NAME
MOXE	MOXE	CUSTOMER
MOXE	MOXE	LINEITEM
MOXE	MOXE	NATION
MOXE	MOXE	ORDERS
MOXE	MOXE	PART
MOXE	MOXE	PARTSUPP
MOXE	MOXE	REGION
MOXE	MOXE	SUPPLIER
RAPIDS	SYSTEM	AUTHENTICATORS
RAPIDS	SYSTEM	AUTHENTICATOR_CONFIG
RAPIDS	SYSTEM	CATALOGS
RAPIDS	SYSTEM	COLUMNS
RAPIDS	SYSTEM	CONNECTORS
RAPIDS	SYSTEM	FEDERATIONS
RAPIDS	SYSTEM	INDEXES
RAPIDS	SYSTEM	NODES

```

RAPIDS      SYSTEM      PATTERN_MAPS
RAPIDS      SYSTEM      QUERIES
RAPIDS      SYSTEM      QUERY_STATS
RAPIDS      SYSTEM      SCHEMAS
RAPIDS      SYSTEM      SESSIONS
RAPIDS      SYSTEM      TABLES
RAPIDS      SYSTEM      TABLE_PROVIDERS
RAPIDS      SYSTEM      USERNAME_MAPS
RAPIDS      SYSTEM      USERS

CATALOG_NAME  SCHEMA_NAME  TABLE_NAME
-----
RAPIDS        SYSTEM      USER_CONFIG

26 row(s) returned (0.22 sec)
rapids > describe table region;
TABLE_NAME    COLUMN_NAME    DATA_TYPE    ORDINAL    IS_PARTITION_KEY
IS_NULLABLE   PRECISION     SCALE COMMENT  PROPERTIES
-----
REGION        COL1           INTEGER       0           false
true          64            NULL NULL     NULL
REGION        COL2           VARCHAR       1           false
true          NULL          NULL NULL     NULL
REGION        COL3           VARCHAR       2           false
true          NULL          NULL NULL     NULL

3 row(s) returned (0.20 sec)

```

Example 2:

Import the file whose name starts with the string “region” (this is the capturing group, see hilited text below), and create a table of that name. This example uses the same folder as the previous example, but this time only the “region” table will get imported:

```

rapids > drop table region;
0 row(s) returned (0.09 sec)
rapids > IMPORT MOXE.* FROM 'node://db1/tpch_small_files' WITH
FILTER='(region).*',GUESS;
0 row(s) returned (0.12 sec)
rapids > select count(*) from region;
[1]
---
5

1 row(s) returned (0.06 sec)

```

11.10.3 Bulk IMPORT Using FOLDERS Option

When doing a bulk import operation the “FOLDERS” option indicates that the path name specified in the import reference (see 11.6) refers to a folder, which contains a set of sub-folders to be imported. The name of each sub-folder is the name of the table where the imported data from the files in the sub-folder will be written.

Example 1:

This example shows the use of the FOLDERS option to import the data from a set of sub-folders where each sub-folder corresponds to a table of the same name.

Folder structure under parent folder /var/tmp/rapids/small_tpch:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/tpch_small/*
/var/tmp/rapids/tpch_small/customer:
customer.csv

/var/tmp/rapids/tpch_small/lineitem:
lineitem.csv

/var/tmp/rapids/tpch_small/nation:
nation.csv

/var/tmp/rapids/tpch_small/orders:
orders.csv

/var/tmp/rapids/tpch_small/part:
part.csv

/var/tmp/rapids/tpch_small/partsupp:
partsupp.csv

/var/tmp/rapids/tpch_small/region:
region.csv

/var/tmp/rapids/tpch_small/supplier:
supplier.csv:
```

The bulk import command below is using the “IF NOT EXISTS” clause to only import those tables that do not currently exist along with the “FOLDERS” option. As the tables are being created, the “GUESS” Property is set TRUE so that the IMPEX Connector will assign the data types. In this example only the “lineitem” table will be imported because all of the other tables already exist:

```
rapids > show tables;
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME
-----
MOXE          MOXE         CUSTOMER
```

```

MOXE          MOXE          NATION
MOXE          MOXE          ORDERS
MOXE          MOXE          PART
MOXE          MOXE          PARTSUPP
MOXE          MOXE          REGION
MOXE          MOXE          SUPPLIER
...

25 row(s) returned (0.22 sec)
rapids > select count(*) from customer;
  [1]
  ---
   75

1 row(s) returned (0.06 sec)

rapids > IMPORT IF NOT EXISTS MOXE.* FROM FOLDERS 'node://db1/tpch_small'
WITH GUESS;
0 row(s) returned (10.63 sec)
rapids > show tables;
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME
-----
MOXE          MOXE          CUSTOMER
MOXE          MOXE          LINEITEM
MOXE          MOXE          NATION
MOXE          MOXE          ORDERS
MOXE          MOXE          PART
MOXE          MOXE          PARTSUPP
MOXE          MOXE          REGION
MOXE          MOXE          SUPPLIER
...

25 row(s) returned (0.23 sec)

rapids > select count(*) from lineitem;
  [1]
  ---
 3000

1 row(s) returned (0.06 sec)
rapids > select count(*) from customer;
  [1]
  ---
   75

1 row(s) returned (0.06 sec)

```

Example 2:

This example shows the use of the “REPLACE” option, where the tables being imported will first be truncated and then the data inserted.

Current row count for “lineitem” table:

```
rapids > select count(*) from lineitem;
  [1]
  ---
  3000

1 row(s) returned (0.06 sec)
```

Import command with “REPLACE” option specified. The “GUESS” Property is set to FALSE (by default) so that all data will be read as polymorphic strings and then cast to the data type for each column:

```
rapids > IMPORT MOXE.* REPLACE FROM FOLDERS 'node://db1/tpch_small' WITH
GUESS=FALSE;
0 row(s) returned (10.03 sec)
```

Current row count showing that only one copy of the data is in the table:

```
rapids > select count(*) from lineitem;
  [1]
  ---
  3000

1 row(s) returned (0.06 sec)
```

Example 3:

This example shows a bulk import where the input files include header records with the column names.

```
rapids@db1:/var/tmp/rapids$ ls tpch_small_folders_with_headers/*/*
tpch_small_folders_with_headers/CUSTOMER/customer.csv
tpch_small_folders_with_headers/PART/part.csv
tpch_small_folders_with_headers/LINEITEM/lineitem.csv
tpch_small_folders_with_headers/PARTSUPP/partsupp.csv
tpch_small_folders_with_headers/NATION/nation.csv
tpch_small_folders_with_headers/REGION/region.csv
tpch_small_folders_with_headers/ORDERS/orders.csv
tpch_small_folders_with_headers/SUPPLIER/supplier.csv
```

Below is the file “region.csv” showing the header row:

```
[rapids@db1 tpch_small_folders_with_headers]$ cat region/REGION.csv
R_REGIONKEY,R_NAME,R_COMMENT
1,UNITED STATES,adknladnganfbmanlgnalkfnlglkajglkafjglksjfglkajfgkjadg
```

```

2,NORTH AMERICA,aldkjlakngkjankgkfnkngakldflkadjlkajdlkajd
3,EUROPE,dxldgkzcssoisjoicnkjebfjhwgrygwuihvokjdgojdvpsd
4,SOUTH AMERICA,csvbdacavbscdbvacdhvjhxdkgnlkgflglksdnja shc asbvda
5,ASIA,i4y5qiuyrqghrushfxhghirohtiuehtaytaiuerytaiurt

```

Below is the bulk import command with the “FOLDERS” option set, and the “HEADER” option set to indicate that each file in a sub-folder includes a header record with the column names:

```

rapids > drop table customer;
0 row(s) returned (0.10 sec)
rapids > drop table lineitem;
0 row(s) returned (0.10 sec)
rapids > drop table nation;
0 row(s) returned (0.10 sec)
rapids > drop table orders;
0 row(s) returned (0.10 sec)
rapids > drop table part;
0 row(s) returned (0.09 sec)
rapids > drop table partsupp;
0 row(s) returned (0.09 sec)
rapids > drop table region;
0 row(s) returned (0.10 sec)
rapids > drop table supplier;
0 row(s) returned (0.09 sec)
rapids > IMPORT MOXE.* FROM FOLDERS 'node://db1/
tpch_small_folders_with_headers WITH HEADER;
0 row(s) returned (0.32 sec)

```

The output below shows the table definitions for two of the tables imported, “nation” and “region”, where you can see that the column names in the header record were used:

```

rapids > describe table nation;
TABLE_NAME      COLUMN_NAME      DATA_TYPE      ORDINAL      IS_PARTITION_KEY
IS_NULLABLE     PRECISION        SCALE COMMENT     PROPERTIES
-----
NATION          N_NATIONKEY     VARCHAR         0            false
true           NULL            NULL NULL         NULL
NATION          N_NAME          VARCHAR         1            false
true           NULL            NULL NULL         NULL
NATION          N_REGIONKEY     VARCHAR         2            false
true           NULL            NULL NULL         NULL
NATION          N_COMMENT       VARCHAR         3            false
true           NULL            NULL NULL         NULL

4 row(s) returned (0.17 sec)
rapids > describe table region;

```

TABLE_NAME	COLUMN_NAME	DATA_TYPE	ORDINAL	IS_PARTITION_KEY
IS_NULLABLE	PRECISION	SCALE	COMMENT	PROPERTIES
REGION	R_REGIONKEY	VARCHAR	0	false
true	NULL	NULL	NULL	NULL
REGION	R_NAME	VARCHAR	1	false
true	NULL	NULL	NULL	NULL
REGION	R_COMMENT	VARCHAR	2	false
true	NULL	NULL	NULL	NULL

3 row(s) returned (0.19 sec)

Example 4:

This example shows an example where an attempt is made to do a bulk import operation where the target tables are managed by different Connectors, which is not allowed:

Below are the current tables that are managed by the “MOXE” and “MOXE2” Connectors:

```
rapids > show tables;
```

CATALOG_NAME	SCHEMA_NAME	TABLE_NAME
MOXE	MOXE	CUSTOMER
MOXE	MOXE	LINEITEM
MOXE	MOXE	NATION
MOXE	MOXE	ORDERS
MOXE	MOXE	REGION
MOXE	MOXE	SUPPLIER
MOXE2	MOXE2	PART
MOXE2	MOXE2	PARTSUPP

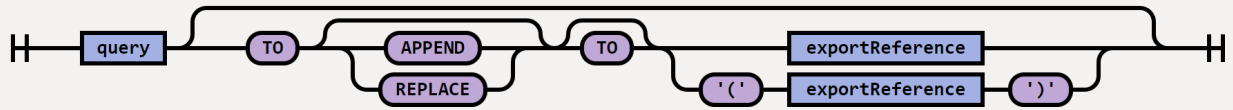
...
29 row(s) returned (0.23 sec)

Below is the bulk import command that will attempt to do imports against all of the tables shown above, which is not allowed because the tables are managed by two different Connectors. The error message shows one table from each Connector which would have been imported into, which in this example are the “PART” table managed by the “MOXE2” Connector and “NATION” table managed by the “MOXE” Connector.

```
rapids > IMPORT * FROM FOLDERS 'node://db1/tpch_small';
Unexpected Exception:
Wildcard import to multiple Connectors: PART, NATION
```

11.11 EXPORT Using SELECT

This section covers the exporting of the results of a query to a file or folder.



11.11.1 EXPORT Using SELECT TO a File

This section provides examples for writing the results of a query to a specified file using the “TO” clause as shown below. In the examples below the hilited text is the export reference (see 11.7):

Example 1:

This example shows exporting the contents of a table, “moxe.region”, to a file “region.csv”, where the file is located in the folder “/var/tmp/rapids/tpch_small_file_backups” on RapidsDB Cluster node “db1”.

Folder: /var/tmp/rapids/tpch_small_file_backups:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/tpch_small_file_backups
[rapids@db1 rapids]$
```

Query to export the data using the default “EXPORT” IMPEX Connector (see 11.8):

```
rapids > SELECT * FROM moxe.region TO
'node://db1/tpch_small_file_backups/region.csv';
0 row(s) returned (0.10 sec)
```

Folder “/var/tmp/rapids/tpch_small_file_backups” after the export, showing that the file has 5 records:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/tpch_small_file_backups
region.csv
[rapids@db1 rapids]$ cat /var/tmp/rapids/tpch_small_file_backups/region.csv |
wc -l
5
```

Example 2:

This example shows that the data being exported will by default be appended to the target.

Current contents of file

```
[rapids@db1 rapids]$ cat /var/tmp/rapids/tpch_small_file_backups/region.csv |
wc -l
5
```

```
rapids > SELECT * FROM moxe.region TO
'node://db1/tpch_small_file_backups/region.csv';
0 row(s) returned (0.10 sec)
```


New count:

```
[rapids@db1 rapids]$ cat /var/tmp/rapids/tpch_small_file_backups/region.csv | wc -l  
10
```

Example 3:

In this example, the “REPLACE” option is used, which will result in the target file getting deleted prior to writing the results of the query, and then the results of the query will get written to the file. In addition, the “HEADER” option is used which will result in the first record of the file being a header record which contains the names of the columns from the result set.

Current contents of file “region.csv”:

```
[rapids@db1 rapids]$ cat /var/tmp/rapids/tpch_small_file_backups/region.csv  
1,UNITED STATES,adknladnganfbmanlgnalkfnlglkajglkafjglksjfglkajfgkjadg  
2,NORTH AMERICA,aldkjlakngkjangkijnfkngakldflkadjlkajdlkajd  
3,EUROPE,dxldgkzcsoisjoicnkjebfjhqwrygwuihvokjdgojdvps  
4,SOUTH AMERICA,csvbdcavbscdbvacdhvjhxdkgnlkgflglksdnja shc asbvda  
5,ASIA,i4y5qiuyrqhrushfxhghirohtiehtaytaiuerytaiurt  
1,UNITED STATES,adknladnganfbmanlgnalkfnlglkajglkafjglksjfglkajfgkjadg  
2,NORTH AMERICA,aldkjlakngkjangkijnfkngakldflkadjlkajdlkajd  
3,EUROPE,dxldgkzcsoisjoicnkjebfjhqwrygwuihvokjdgojdvps  
4,SOUTH AMERICA,csvbdcavbscdbvacdhvjhxdkgnlkgflglksdnja shc asbvda  
5,ASIA,i4y5qiuyrqhrushfxhghirohtiehtaytaiuerytaiurt
```

Export command specifying the “REPLACE” and “HEADER” options:

```
rapids > SELECT * FROM moxe.region TO REPLACE  
'node://db1/tpch_small_file_backups/region.csv' WITH HEADER;  
0 row(s) returned (0.08 sec)
```

Contents of exported file, with a header record with the column names from the result set:

```
[rapids@db1 rapids]$ cat /var/tmp/rapids/tpch_small_file_backups/region.csv  
R_REGIONKEY,R_NAME,R_COMMENT  
1,UNITED STATES,adknladnganfbmanlgnalkfnlglkajglkafjglksjfglkajfgkjadg  
2,NORTH AMERICA,aldkjlakngkjangkijnfkngakldflkadjlkajdlkajd  
3,EUROPE,dxldgkzcsoisjoicnkjebfjhqwrygwuihvokjdgojdvps  
4,SOUTH AMERICA,csvbdcavbscdbvacdhvjhxdkgnlkgflglksdnja shc asbvda  
5,ASIA,i4y5qiuyrqhrushfxhghirohtiehtaytaiuerytaiurt
```

Example 4:

This is the same as the previous example, except this time the BACKUP Property is set “true” which results in the “region.csv” file getting moved to a backup folder named “_backup.<internal_timestamp>” and then the export of the query results is done.

where <internal_timestamp> is a numerical value for the timestamp when the query results were generated.

```
rapids > SELECT * FROM moxe.region TO REPLACE
'node://db1/tpch_small_file_backups/region.csv' WITH HEADER, BACKUP;

0 row(s) returned (0.08 sec)
```

Target folder with exported file along with backup folder “/_backup.777214467718270673”, has the original “region.csv” file before the export was done.

```
[rapids@db1 tpch_small_file_backups]$ ls
/var/tmp/rapids/tpch_small_file_backups/*
/var/tmp/rapids/tpch_small_file_backups/region.csv

/var/tmp/rapids/tpch_small_file_backups/_backup.777214467718270673:
region.csv
```

11.11.2 EXPORT Using SELECT TO a Folder

This section provides examples for writing the results of a query to a specified folder using the “TO FOLDER” clause as shown below. The query results are written to a file named:

- query_results_<internal_timestamp>.csv

where,

<internal_timestamp> is the timestamp when the query results were generated.

Example 1:

This example uses the “FOLDER” option to write the results of the specified query to the specified folder, “/var/tmp/rapids/query_results”. The file with the query results will be named “query.<timestamp>”, where <timestamp> is the timestamp when the query was executed.

Folder: /var/tmp/rapids/query_results

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/query_results
[rapids@db1 rapids]$
```

```
rapids > SELECT * FROM moxe.region TO FOLDER 'node://db1/query_results';
0 row(s) returned (0.08 sec)
```

Target folder, “/var/tmp/rapids/query_results”, after export:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/query_results
query_results.3959867675829689450.csv

[rapids@db1 rapids]$ cat
/var/tmp/rapids/query_results/query_results.3959867675829689450.csv
1,UNITED STATES,adknladnganfbmanlgnalkfnglkajglkafjglksjfglkajfgkjadg
2,NORTH AMERICA,aldkjlakngkjanganjfnkngakldflkadjlkajdlkajd
3,EUROPE,dxldgkzcsoisjoicnkjebfjhqwgrgywuihvokjdgojdvps
4,SOUTH AMERICA,csvbdcavbscdvacdhvjhxdkgnlkgflglksdnja shc asbvda
5,ASIA,i4y5qiuyrqhrushfxhghirohtiehtaytaiuerytaiurt
```

Example 2:

This example shows a second query results file being written to the same target folder:

Current contents of target folder:

```
[rapids@db1 query_results]$ ls /var/tmp/rapids/query_results
query_results.2320079842552510970.csv
```

Export command:

```
rapids > SELECT * FROM moxe.customer WHERE customer.c_acctbal >0 AND EXISTS
(SELECT * FROM vip_customer WHERE vip_customer.c_custkey =
customer.c_custkey) TO FOLDER 'node://db1/query_results';

0 row(s) returned (0.41 sec)
```

Contents of target folder with new query results file:

```
[rapids@db1 query_results]$ ls /var/tmp/rapids/query_results
query_results.1910530767001465758.csv query_results.2320079842552510970.csv
```

Example 3:

This example shows the use of the “REPLACE” and “BACKUP” options where the existing files (with a “.csv” suffix) from the specified folder are first moved to a backup folder and then the new query results files are written:

Current target folder:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/query_results
query_results.1500591431418200400.csv query_results.3449950617860697261.csv
```

Export command with “REPLACE” option set:

```
rapids > SELECT * FROM moxe.customer WHERE customer.c_acctbal > 0 AND EXISTS
(SELECT * FROM vip_customer WHERE vip_customer.c_custkey =
customer.c_custkey) TO REPLACE FOLDER 'node://db1/query_results' WITH BACKUP;

0 row(s) returned (0.76 sec)
```

The backup folder will get created in the parent directory of the folder specified in the export reference, which in this case would be “/var/tmp/rapids”:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids
.backup.4636680165345356631  query_results  SFSMALL  text
tpch_small_backup            tpch_small_files
formats                       SF100         single    tpch_small
tpch_small_file_backups      tpch_small_with_headers
```

The contents of the backup folder are the original files prior to the export:

```
[rapids@db1 rapids]$ ls
/var/tmp/rapids/.backup.4636680165345356631/query_results
query_results.1500591431418200400.csv  query_results.3449950617860697261.csv
```

Example 4:

This example shows the use of the “REPLACE” option with “BACKUP=false” to first delete all existing files from the specified folder before writing the new query results file.

Contents of target folder with new query results file:

```
[rapids@db1 query_results]$ ls /var/tmp/rapids/query_results
query_results.1910530767001465758.csv  query_results.2320079842552510970.csv
```

Contents of the parent directory:

```
[rapids@db1 /var/tmp/rapids]$ ls /var/tmp/rapids
.backup.4107715935291356825  SFSMALL  tpch_small
tpch_small_files              tpch_small_folders_with_headers
query_results                 text      tpch_small_file_backups
tpch_small_file_with_headers
```

Export command using “REPLACE” option:

```
rapids > SELECT * FROM moxe.customer WHERE customer.c_acctbal > 0 AND EXISTS
(SELECT * FROM vip_customer WHERE vip_customer.c_custkey =
customer.c_custkey) TO REPLACE FOLDER 'node://db1/query_results';
```

```
0 row(s) returned (0.37 sec)
```

Contents of target folder after export showing that two previous files were deleted:

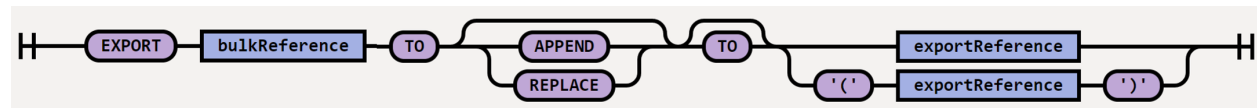
```
[rapids@db1 query_results]$ ls /var/tmp/rapids/query_results
query_results.9088214351952178143.csv
```

No new backup folder was created:

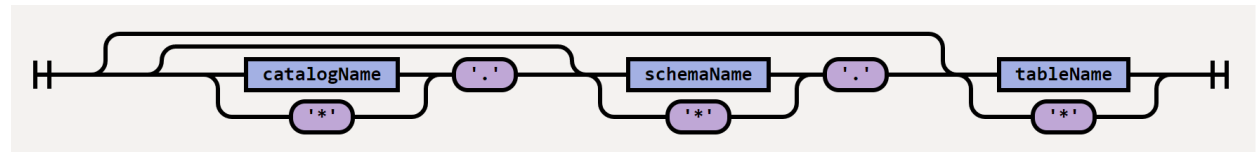
```
[rapids@db1 /var/tmp/rapids]$ ls /var/tmp/rapids
.backup.4107715935291356825  SFSMALL  tpch_small
tpch_small_files            tpch_small_folders_with_headers
query_results                text      tpch_small_file_backups
tpch_small_file_with_headers
```

11.12 Bulk EXPORT

The *EXPORT* statement is used for directly exporting multiple tables in a single request:



bulkReference:



Option	Required?	Default?	Description
bulkReference	Yes	N/A	Specifies the three-level (catalog/schema/table) names for the table(s) to be exported. Wildcards may be specified (using asterisk '*') for any of the name components. If catalog name and/or schema name are omitted, CURRENT_CATALOG and CURRENT_SCHEMA are used (if set).
APPEND	No	Yes	Append the exported data to any existing file or folder at the destination (as specified by the exportReference – see below). (Note: may not be supported for some Connectors and/or destinations.) This is the default behavior
REPLACE	No	No	Delete any existing files with a suffix of “.csv” (when the “FILES” option is specified in the export reference (see 11.8)) or all the files with a suffix of “.csv” in a sub-folder

			(when the “FOLDERS” option is specified in the export reference (see folder 11.7)) . If the “BACKUP” Property for the Connector (see 11.4) is “true” then instead of deleting the files in the folder (or sub-folder), the files will be moved to a backup folder.
exportReference	Yes	N/A	An Export Reference (see 11.7) identifying the destination for the exported data.

11.12.1 Backing Up Files/Sub-Folders When Doing a REPLACE

When doing a bulk export operation with the “REPLACE” option (see above), the user is requesting that the target files for the export (when the export reference is using the “FILES” option, which is the default - see 11.12.2 for examples), or the files in the target sub-folders when the export reference is using the “FOLDERS” option (see 11.12.3 for examples), are to be replaced with new copies. In order to allow the user to recover any replaced files, the IMPEX Connector supports the “BACKUP” Property (see 11.12.2 for examples), which when set to “true” results in the system moving the files to be replaced to a backup folder so that they can be recovered after the export operation if needed. See 11.12.1.1 for more information on backup with the “FILES” option, and 11.12.1.2 for more information on backup with the “FOLDERS” option. By default, all IMPEX Connectors have the “BACKUP” Property set “false” which means that all bulk export operations where the “REPLACE” option is specified no backup of the existing files will be done.

11.12.1.1 Backup for FILES option

When doing a backup for the FILES option (BACKUP=true), a backup folder will get created in the parent directory of the folder specified in the export reference, with the name of the backup folder being:

`_.backup.<epoch timestamp>/<export folder>`

where,

<epoch timestamp> is the Unix Epoch timestamp when the export command was executed

<export folder> is the name of the folder specified in the export reference in the bulk export command

All files with a suffix of “.csv” from the folder specified in the export reference in the bulk export command will be moved to the backup folder.

NOTE: Since the “BACKUP” Property is set “false” by default, no backup of existing files will be performed when the “REPLACE” option is specified. If needed, the user can change the default setting for the “BACKUP” Property to “true” to ensure that a backup copy is made:

CREATE CONNECTOR EXPORT_NOBACKUP TYPE IMPEX WITH **BACKUP,...;**

or, by setting the “BACKUP” Property as part of the bulk export command:

```
EXPORT MOXE.* TO REPLACE 'node://db1/tpch_small_file_backups' WITH BACKUP, HEADER, DELIMITER='|', ENCLOSED_BY='\"';
```

See Examples 2 and 3 in section 11.13.2 below for more information.

Example:

This example is using the default “EXPORT” Connector, which for this example is using the default setting for the “PATH” Property which is “/var/tmp/rapids”.

```
EXPORT MOXE.* TO REPLACE 'node://db1/tpch_small_file_backups' WITH HEADER, DELIMITER='|', ENCLOSED_BY='\"';
```

In this example any files with a suffix of “.csv” that are present in the folder specified in the export reference, which in this example would be “/var/tmp/rapids/tpch_small_backups”, would be moved to a folder in the parent directory, which in this example would be “/var/tmp/rapids”, where the backup folder would be named similarly to the following:

```
“/var/tmp/rapids/tpch_small_file_backups/_backup.777214467718270673/ tpch_small_file_backups”
```

If needed, the user can then recover any needed files from the backup folder.

11.12.1.2 Backup for FOLDERS option

When doing a backup for the FOLDERS option (in the export reference), a backup folder will get created in each subfolder which holds the export files for each table being exported. The name of the backup folder will be:

```
_backup.<epoch timestamp>
```

where,

<epoch timestamp> is the Unix Epoch timestamp when the export command was executed

All files with a suffix of “.csv” from the sub-folder moved to the backup folder. See example below for more details.

NOTE: Since the “BACKUP” Property is set “false” by default, no backup of existing files will be performed when the “REPLACE” option is specified. If needed, the user can change the default setting for the “BACKUP” Property to “true” to ensure that a backup copy is made:

```
CREATE CONNECTOR EXPORT_NOBACKUP TYPE IMPEX WITH BACKUP,...;
```

or, by setting the “BACKUP” Property as part of the bulk export command:

```
rapids > EXPORT MOXE.* TO FOLDERS 'node://db1/tpch_small_backup' WITH HEADER, BACKUP;
```

See Examples 2 and 3 in section 11.12.3 below for more information.

Example:

This example is using the default “EXPORT” Connector, which for this example is using the default setting for the “PATH” Property which is “/var/tmp/rapids”. In this example, assume that one of the tables being exported is named “NATION”.

```
rapids > EXPORT MOXE.* TO FOLDERS 'node://db1/tpch_small_backup' WITH BACKUP, HEADER;
```

In this example any files with a suffix of “.csv” that are present in the sub-folder associated with each table being exported, such as “/var/tmp/rapids/tpch_small_backup/NATION” would be moved to a backup folder in the sub-folder directory for that table, which in this example would be named similarly to the following:

```
“/var/tmp/rapids/tpch_small_backup/NATION/_.backup.777214467718270673/”
```

If needed, the user can then recover any needed files from any of the backup folders.

11.12.2 Bulk EXPORT Using FILES Option

The “FILES” option for a bulk export indicates that each table should be written out to the specified folder with name <table name>.csv.

Example 1:

This is an example of a simple bulk export where all of the tables from the schema “MOXE” are to be exported to the folder /var/tmp/rapids/tpch_small_file_backups, with each file having a header record with the column names.

Current contents of folder /var/tmp/rapids/tpch_small_file_backups:

```
[rapids@db1 tpch_small_file_backups]$ ls
/var/tmp/rapids/tpch_small_file_backups
AAREADme.txt
```

Tables to be exported:

```
rapids > show tables;
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME
-----
MOXE          MOXE         CUSTOMER
MOXE          MOXE         LINEITEM
MOXE          MOXE         NATION
MOXE          MOXE         ORDERS
MOXE          MOXE         PART
MOXE          MOXE         PARTSUPP
MOXE          MOXE         REGION
MOXE          MOXE         SUPPLIER
MOXE          MOXE         VIP_CUSTOMER
```


Export command. The “FILES” option is not required because it is the default. The “HEADER” property is set to indicate that a header record is to be written for each exported table, and the “DELIMITER” property is set to the semicolon character and the “ENCLOSED_BY” property is set to the single quote character:

```
rapids > EXPORT MOXE.* TO 'node://db1/tpch_small_file_backups' WITH HEADER,  
DELIMITER=';', ENCLOSED_BY="'";  
0 row(s) returned (9.43 sec)
```

Folder after the export showing the exported tables:

```
[rapids@db1 tpch_small_file_backups]$ ls  
AAREADME.txt CUSTOMER.csv LINEITEM.csv NATION.csv ORDERS.csv PART.csv  
PARTSUPP.csv REGION.csv SUPPLIER.csv VIP_CUSTOMER.csv
```

Example of header record in one of the files, along with the field delimiter being set to the semicolon character and the enclosed_by character being set to single quote. Note that some of the varchar fields are enclosed in single quotes, and this is because those fields include the delimiter character which is a semicolon. The only time that character fields are enclosed is when they contain the delimiter character.:

```
[rapids@db1 tpch_small_file_backups]$ cat NATION.csv  
N_NATIONKEY;N_NAME;N_REGIONKEY;N_COMMENT  
1;UNITED STATES;1;adknladnganfbmanlgnalkfnglkaajglkafjglksjfglkaajfgkjadg  
2;CANADA;2;aldkjlakngkjanganjnfkngakldflkadjlkajdlkajd  
3;MEXICO;2;94iuakdjvoakdjvoadjogadhgkjagkjazdgkjzgd  
4;GERMANY;3;zoidhgdjhtehgkhgnkjsdzdkjg  
5;FRANCE;3;odijzoietjoizejtoizejgioazgkjan  
6;ENGLAND;3;oaidjfozdjfoizjeofjelkjtkljlkjg  
7;NORWAY;3;oaieurlajelgtkjalkgjlkajsgl  
8;DENMARK;3;zokjvzlzkjdlgfjzdlkgjlkadjglkdjglkaajglkjglkdjglkjfdlhk  
9;SWEDEN;3;zkjdghfkjdvnm mv bmz lknvzkzdfng  
10;ITALY;3;'dkjzlz;f;lsfkh;lakhr;lajlfgkagjdba'  
11;SWITZERLAND;3;'ozkdjvdjtkjaldg;lg;lkfh;lks;flhadfbjfhd'  
12;POLAND;3;kdjlkzdtljelktjlzjlkrlkdhkbafjebe  
13;RUSSIA;3;'zflkzdgjlzjglzdkg;lrdkh;ldfkh;dfllk'  
14;CHILE;4;'akdjflkj;d;ljfal;djgsdjglkdsfjgkjasjfbjha'  
15;PERU;4;'ldkjflaje;ajlgdjzlkdgkzjdhfzbsjdhfb'  
16;ARGENTINA;4;dkjzkljlkgjzldgjkhghfjgfgsg  
17;BOLIVIA;4;alkjdfllkajdljgaljglkaajgljadfg  
18;BRAZIL;4;jadalkjakdjlkjakgjalkfjglkaajfgjalfg  
19;GUYANA;4;zlkjdzkjdldkgjaldkgjlkadjglkdjglkjdkgjlsdfkjhllksjfh  
20;CHINA;5;lkdjaldlkajdgkjafghskfdjghksjffhkjslkh  
21;JAPAN;5;'skfjg;lksjfhksjkhjsjhakjbgbgbs'  
22;INDIA;5;'kadjflkajdlkjgs;lksjglksfjglksfjhks'
```

```
23;PHILLIPINES;5;'aldkjflkadjflkjadvjz;lvlkdlgjajkdg'  
24;THAILAND;5;aldkjflakdjglkajgflkfjgkjsflkjgalkjfg  
25;SINGAPORE;5;dlkjalkdjflkajdlkzvnknzkdfglkzjdfhglkjflkh
```

Example 2:

This example shows the use of the “REPLACE” option to replace the existing files in the specified folder, (with the default “BACKUP” option set “false”), which will result in the existing files in the target folder getting deleted prior to the export of the tables.

Current contents of target folder:

```
[rapids@db1 tpch_small_file_backups]$ ls -l  
/var/tmp/rapids/tpch_small_file_backups  
total 145032  
-rw-rw-r--. 1 rapids rapids      20 Sep 20 20:21 AAREADME.txt  
-rw-rw-r--. 1 rapids rapids  4516901 Sep 20 20:52 CUSTOMER.csv  
-rw-rw-r--. 1 rapids rapids 125657939 Sep 20 20:52 LINEITEM.csv  
-rw-rw-r--. 1 rapids rapids      85 Sep 20 20:52 MYTABLE.csv  
-rw-rw-r--. 1 rapids rapids   2348 Sep 20 20:52 NATION.csv  
-rw-rw-r--. 1 rapids rapids  7550516 Sep 20 20:52 ORDERS.csv  
-rw-rw-r--. 1 rapids rapids  4526668 Sep 20 20:52 PART.csv  
-rw-rw-r--. 1 rapids rapids  4739265 Sep 20 20:52 PARTSUPP.csv  
-rw-rw-r--. 1 rapids rapids     82 Sep 20 20:52 REGION2.csv  
-rw-rw-r--. 1 rapids rapids    439 Sep 20 20:52 REGION.csv  
-rw-rw-r--. 1 rapids rapids   3344 Sep 20 20:52 SPECIAL_CUSTOMER.csv  
-rw-rw-r--. 1 rapids rapids  1478162 Sep 20 20:52 SUPPLIER.csv  
-rw-rw-r--. 1 rapids rapids   4601 Sep 20 20:52 VIP_CUSTOMER.csv
```

Export command with “REPLACE” option using the default “BACKUP” Property (“false”):

```
rapids > EXPORT MOXE.* TO REPLACE 'node://db1/tpch_small_file_backups' WITH  
HEADER, DELIMITER='|', ENCLOSED_BY="";  
  
0 row(s) returned (9.06 sec)
```

Contents of target folder after the export showing new copies of the exported files and no backups:

```
[rapids@db1 tpch_small_file_backups]$ ls -l  
/var/tmp/rapids/tpch_small_file_backups  
total 145032  
-rw-rw-r--. 1 rapids rapids      20 Sep 20 20:21 AAREADME.txt  
-rw-rw-r--. 1 rapids rapids  4516901 Sep 20 21:12 CUSTOMER.csv  
-rw-rw-r--. 1 rapids rapids 125657939 Sep 20 21:12 LINEITEM.csv  
-rw-rw-r--. 1 rapids rapids      85 Sep 20 21:12 MYTABLE.csv  
-rw-rw-r--. 1 rapids rapids   2348 Sep 20 21:12 NATION.csv  
-rw-rw-r--. 1 rapids rapids  7550516 Sep 20 21:12 ORDERS.csv  
-rw-rw-r--. 1 rapids rapids  4526668 Sep 20 21:12 PART.csv
```

```

-rw-rw-r--. 1 rapids rapids 4739265 Sep 20 21:12 PARTSUPP.csv
-rw-rw-r--. 1 rapids rapids 82 Sep 20 21:12 REGION2.csv
-rw-rw-r--. 1 rapids rapids 439 Sep 20 21:12 REGION.csv
-rw-rw-r--. 1 rapids rapids 3344 Sep 20 21:12 SPECIAL_CUSTOMER.csv
-rw-rw-r--. 1 rapids rapids 1478162 Sep 20 21:12 SUPPLIER.csv
-rw-rw-r--. 1 rapids rapids 4601 Sep 20 21:12 VIP_CUSTOMER.csv

```

Example 3:

This example shows the use of the “REPLACE” option to replace the existing files (ending in “.csv”) in the specified folder, with the “BACKUP” option set to “true”, which will result in the existing files being moved to a backup folder so that they can be recovered in the future if needed (see 11.12.1) for more information).

Current backup folder:

```

[rapids@db1 tpch_small_file_backups]$ ls
/var/tmp/rapids/tpch_small_file_backups
total 145032
-rw-rw-r--. 1 rapids rapids 20 Sep 20 20:21 AAREADME.txt
-rw-rw-r--. 1 rapids rapids 4516901 Sep 20 20:46 CUSTOMER.csv
-rw-rw-r--. 1 rapids rapids 125657939 Sep 20 20:46 LINEITEM.csv
-rw-rw-r--. 1 rapids rapids 85 Sep 20 20:46 MYTABLE.csv
-rw-rw-r--. 1 rapids rapids 2348 Sep 20 20:46 NATION.csv
-rw-rw-r--. 1 rapids rapids 7550516 Sep 20 20:46 ORDERS.csv
-rw-rw-r--. 1 rapids rapids 4526668 Sep 20 20:46 PART.csv
-rw-rw-r--. 1 rapids rapids 4739265 Sep 20 20:46 PARTSUPP.csv
-rw-rw-r--. 1 rapids rapids 82 Sep 20 20:46 REGION2.csv
-rw-rw-r--. 1 rapids rapids 439 Sep 20 20:46 REGION.csv
-rw-rw-r--. 1 rapids rapids 3344 Sep 20 20:46 SPECIAL_CUSTOMER.csv
-rw-rw-r--. 1 rapids rapids 1478162 Sep 20 20:46 SUPPLIER.csv
-rw-rw-r--. 1 rapids rapids 4601 Sep 20 20:46 VIP_CUSTOMER.csv

```

Bulk export command with the “REPLACE” option specified, and the “BACKUP” Property set:

```

rapids > EXPORT MOXE.* TO REPLACE 'node://db1/tpch_small_file_backups' WITH
BACKUP, HEADER, DELIMITER='|', ENCLOSED_BY="";
0 row(s) returned (9.29 sec)

```

Contents of target folder after the export. Note that there are new copies of the files for the exported tables, and that there is a backup folder, “_backup.8186816724347336840” that contains the original files:

```

[rapids@db1 tpch_small_file_backups]$ ls -l
/var/tmp/rapids/tpch_small_file_backups
total 145032

```

```

drwx----- . 2 rapids rapids      246 Sep 20 20:52
_.backup.8186816724347336840
-rw-rw-r-- . 1 rapids rapids        20 Sep 20 20:21 AAREADME.txt
-rw-rw-r-- . 1 rapids rapids    4516901 Sep 20 20:52 CUSTOMER.csv
-rw-rw-r-- . 1 rapids rapids 125657939 Sep 20 20:52 LINEITEM.csv
-rw-rw-r-- . 1 rapids rapids      85 Sep 20 20:52 MYTABLE.csv
-rw-rw-r-- . 1 rapids rapids    2348 Sep 20 20:52 NATION.csv
-rw-rw-r-- . 1 rapids rapids  7550516 Sep 20 20:52 ORDERS.csv
-rw-rw-r-- . 1 rapids rapids  4526668 Sep 20 20:52 PART.csv
-rw-rw-r-- . 1 rapids rapids  4739265 Sep 20 20:52 PARTSUPP.csv
-rw-rw-r-- . 1 rapids rapids     439 Sep 20 20:52 REGION.csv
-rw-rw-r-- . 1 rapids rapids  1478162 Sep 20 20:52 SUPPLIER.csv
-rw-rw-r-- . 1 rapids rapids    4601 Sep 20 20:52 VIP_CUSTOMER.csv

```

Here is the backup folder with the original files:

```

[rapids@db1 tpch_small_file_backups]$ ls -l
/var/tmp/rapids/tpch_small_file_backups/_.backup.8186816724347336840
total 145032
-rw-rw-r-- . 1 rapids rapids    4516901 Sep 20 20:46 CUSTOMER.csv
-rw-rw-r-- . 1 rapids rapids 125657939 Sep 20 20:46 LINEITEM.csv
-rw-rw-r-- . 1 rapids rapids    2348 Sep 20 20:46 NATION.csv
-rw-rw-r-- . 1 rapids rapids  7550516 Sep 20 20:46 ORDERS.csv
-rw-rw-r-- . 1 rapids rapids  4526668 Sep 20 20:46 PART.csv
-rw-rw-r-- . 1 rapids rapids  4739265 Sep 20 20:46 PARTSUPP.csv
-rw-rw-r-- . 1 rapids rapids     439 Sep 20 20:46 REGION.csv
-rw-rw-r-- . 1 rapids rapids  1478162 Sep 20 20:46 SUPPLIER.csv
-rw-rw-r-- . 1 rapids rapids    4601 Sep 20 20:46 VIP_CUSTOMER.csv

```

11.12.3 Bulk EXPORT Using FOLDERS Option

The “FOLDERS” option for a bulk export indicates that each table should be written out in a file in a separate sub-folder (under the folder name specified in the export reference) of the same name. The name of the file for the exported table will be: <table name>>internal timestamp>.csv, for example “SUPPLIER.5674361502309579557.csv.”

The following examples all assume that the schema “MOXE” has the following tables: CUSTOMER, LINEITEM, NATION, ORDERS, PART, PARTSUPP, REGION, SUPPLIER, and VIP_CUSTOMER

Example 1:

This is an example of a bulk export where all of the tables from the schema “MOXE” are to be exported to sub-folders under the folder “/var/tmp/rapids/tpch_small_backup”, with each export file having a header record with the column names.

Current contents of target folder:

```

[rapids@db1 rapids]$ ls /var/tmp/rapids/tpch_small_folder_backup

```

```
[rapids@db1 rapids]$
```

Export command with the “FOLDERS” option

```
rapids > EXPORT MOXE.* TO FOLDERS 'node://db1/tpch_small_folder_backup' WITH  
HEADER;  
0 row(s) returned (9.43 sec)
```

Contents of target folder after export:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/tpch_small_folder_backup/*  
/var/tmp/rapids/tpch_small_folder_backup/CUSTOMER:  
customer.7842865417157613111.csv  
  
/var/tmp/rapids/tpch_small_folder_backup/LINEITEM:  
lineitem.5733617788919186767.csv  
  
/var/tmp/rapids/tpch_small_folder_backup/NATION:  
nation.1554588814455789912.csv  
  
/var/tmp/rapids/tpch_small_folder_backup/ORDERS:  
orders.6236921272679811628.csv  
  
/var/tmp/rapids/tpch_small_folder_backup/PART:  
part.5359586359844433055.csv  
  
/var/tmp/rapids/tpch_small_folder_backup/PARTSUPP:  
partsupp.9185786343941211758.csv  
  
/var/tmp/rapids/tpch_small_folder_backup/REGION:  
region.7738184263372301381.csv  
  
/var/tmp/rapids/tpch_small_folder_backup/SUPPLIER:  
supplier.4865294074583220638.csv  
  
/var/tmp/rapids/tpch_small_folder_backup/VIP_CUSTOMER:  
vip_customer.6098121580374720698.csv
```

Example 2:

This example shows the use of the “REPLACE” option which will result in the existing files (ending in “.csv”) in the sub-folder for each table being deleted prior to the export being executed.

Below is the current contents of one of the sub-folders for one of the tables (MOXE.LINEITEM)being exported:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/tpch_small_folder_backup/LINEITEM  
lineitem.6482498489887946522.csv
```

Export command with the “REPLACE” option :

```
rapids > EXPORT MOXE.* TO REPLACE FOLDERS
'node://db1/tpch_small_folder_backup' WITH HEADER;

0 row(s) returned (9.06 sec)
```

Contents of the target folder showing the sub-folders for each table, and then the contents of the sub-folder for the “LINEITEM” table showing that a new copy of the exported file has been created, with no backup folder getting created:

```
[rapids@db1 tpch_small_folder_backup]$ ls
CUSTOMER LINEITEM NATION ORDERS PART PARTSUPP REGION SUPPLIER
VIP_CUSTOMER
[rapids@db1 tpch_small_backup]$ ls LINEITEM
lineitem.7547050142483739703.csv
```

Example 3:

This example again shows the use of the “REPLACE” option to replace the existing files (ending in “.csv”) in the sub-folder for each table, but in this example the “BACKUP” option is set which results in the existing files being moved to a backup folder so that they can be recovered in the future if needed (see 11.13.1.2 for more information). The backup folder will be created in the folder specified in the export reference.

Below is the current contents of one of the sub-folders for one of the tables (MOXE.LINEITEM)being exported:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/tpch_small_folder_backup/LINEITEM
lineitem.6260590681045515755.csv
```

Export command specifying the “REPLACE” option:

```
rapids > EXPORT MOXE.* TO REPLACE FOLDERS
'node://db1/tpch_small_folder_backup' WITH HEADER, BACKUP;

0 row(s) returned (9.06 sec)
```

Contents of sub-folder after the export showing the new export file:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/tpch_small_backup/LINEITEM
lineitem.6482498489887946522.csv
```

Contents of folder, “/var/tmp/rapids/tpch_small_backup”, specified in the export reference, showing the backup folder that was created, “_backup.6582410041113988538”:

```
[rapids@db1 rapids]$ ls /var/tmp/rapids/tpch_small_folder_backup
_.backup.6582410041113988538 CUSTOMER LINEITEM NATION ORDERS PART
PARTSUPP REGION SUPPLIER VIP_CUSTOMER
```

Contents of backup folder showing all of the original sub-folders:

```
[rapids@db1 rapids]$ ls
/var/tmp/rapids/tpch_small_folder_backup/_.backup.6582410041113988538
CUSTOMER LINEITEM NATION ORDERS PART PARTSUPP REGION SUPPLIER
VIP_CUSTOMER
```

Sample contents of the backup LINEITEM sub-folder from the backup showing that it contains the original export file for the “LINEITEM” table:

```
[rapids@db1 rapids]$ ls
/var/tmp/rapids/tpch_small_folder_backup/_.backup.6582410041113988538/LINEITE
M
lineitem.6260590681045515755.csv
```

11.13 Error Handling

11.13.1 ERROR_PATH

The “ERROR_PATH” Property (see 11.4) specifies the fully qualified path name to use as the base path for the error files generated if an import operation fails. By default, the ERROR_PATH is set to “/var/tmp/rapids_errors”. For each failed import, a sub-folder will be created in the folder specified by the “ERROR_PATH”:

The sub-folder name is of the form:

SSN_<session number>_<node name>_<query number>

where,

<session number> is the session number for the query that failed

<node name> is the name of the RapidsDB cluster node where the query was submitted

<query number> is the query number for that session

For example, “SSN_2_DB1_67” indicates that this error occurred on session #2 on the RapidsDB Cluster node “DB1” and it was query #67.

The sub-folder will include two files:

- 1 A log file containing details on the conversion errors, one line per error. The format for the log file name is:

`<source>-messages.log`

where,

`<source>` is used to identify the source file or folder with the errors, and has the format:

`node__<node name>_<path name>-messages.log`

where,

`<node name>` is the RapidsDB Cluster node name where the input file or folder resides

`<path name>` is the path name to the source file or folder

Example:

```
rapids > insert into region_b SELECT * FROM (csv_header::  
'node://db1/SFSMALL/regionPipe.csv' );  
  
Error: import errors (5) in /var/tmp/rapids_errors/SSN_1_DB1_234
```

In this example, the errors occurred in the input file

“/var/tmp/rapids/SFSMALL/regionPipe.csv” on RapidsDB cluster node “db1”, and the details for the errors will be in the folder “/var/tmp/rapids_errors/SSN_1_DB1_234” on the same node as the input file, which is RapidsDB cluster node “db1”. The log file will be the file

/var/tmp/rapids_errors/SSN_1_DB1_234/node__db1_var_tmp_rapids_SFSMALL_regionPipe_csv-messages.log:

```
[rapids@db1 tpch_small_backup]$ cat  
/var/tmp/rapids_errors/SSN_1_DB1_234/node__db1_var_tmp_rapids_SFSM  
ALL_regionPipe_csv-messages.log  
segment 0 line 1: java.lang.NumberFormatException: For input  
string: "UNITED STATES"  
segment 0 line 2: java.lang.NumberFormatException: For input  
string: "NORTH AMERICA,aldkjlakngkjjangkjnfkngakldflkadjlkajdlkajd"  
segment 0 line 3: java.lang.NumberFormatException: For input  
string: "EUROPE"  
segment 0 line 4: java.lang.NumberFormatException: For input  
string: "SOUTH AMERICA"  
segment 0 line 5: java.lang.NumberFormatException: For input  
string: "ASIA"
```


- 2 A data file containing the records with the conversion errors, where the records in the data file match up with the error line in the log file described in the previous section. The format for the data file name follows the same format as for the log file:

<source>-records.csv

where,

<source> is used to identify the source file or folder with the errors, and has the format:

node__<node name>_<path name>-records.csv

where,

<node name> is the RapidsDB Cluster node name where the file or folder resides

<path name> is the path name to the source file or folder

Example:

From the previous example, the data file with the error records will be the file:

“/var/tmp/rapids_errors/SSN_1_DB1_234/node__db1_var_tmp_rapids_SFSSMALL_regionPipe_csv-records.csv”:

```
[rapids@db1 tpch_small_backup]$ cat
/var/tmp/rapids_errors/SSN_1_DB1_234/node__db1_var_tmp_rapids_SFSSM
ALL_regionPipe_csv-records.csv
1|UNITED
STATES|adknladnganfbmanlgnalkfnglka jglkafjglksjfglka jfgkjadg
2|NORTH AMERICA,aldkjlakngkj angkjnfkngakldflkadjlkajdlkajd
3|EUROPE|dxldgkzcsoisjoicnkjebfjhqwgrygwuihvokjdgodjvdpds
4|SOUTH AMERICA|csvbdcavbscdbvacdhvjhxdkgnlkgflglsdnja shc asbvda
5|ASIA|i4y5qiuyrqghrushfxhghirohtiehtaytaiuerytaiurt
```

See 11.13.3 for more examples

The default for the “ERROR_PATH” is “/var/tmp/rapids_errors”, but it can be changed either at the Connector level or as part of the import command:

```
rapids > create connector csv_error type impex with
error_path='/data/errors';
0 row(s) returned (2.31 sec)
```

```
rapids > insert into bad2 select col1, col2 from
('node://db1/text/lead_trail_blanks.csv' WITH
ERROR_PATH='/var/tmp/dcerrors');
Unexpected Exception:
import errors (6) in /var/tmp/dcerrors/SSN_2_DB1_76
```

11.13.2 ERROR_LIMIT

The “ERROR_LIMIT” Property specifies the maximum number of allowable errors for an import. Once the limit is reached the import operation will terminate. By default, the limit is set to ten. The possible values for ERROR_LIMIT are:

- -1 No limit, the import will continue regardless of the number of errors
- 0 The import will terminate on the first error
- >0 The import will terminate after the specified number

Example 1:

This example shows an import hitting the default limit:

```
rapids > create table moxe.nation_a(c1 integer, c2 integer, c3 integer, c4
integer);
0 row(s) returned (0.12 sec)
rapids > insert into moxe.nation_a SELECT * FROM
('node://db1/SFSMALL/nation.csv' );
Unexpected Exception:
Error limit (10) reached:
/var/tmp/rapids_errors/SSN_1_DB1_241/node___db1_var_tmp_rapids_SFSMALL_nation
_csv-messages.log
```

Example 2:

This example shows the effect of increasing the limit:

```
rapids > insert into moxe.nation_a SELECT * FROM
('node://db1/SFSMALL/nation.csv' WITH ERROR_LIMIT=100);
Error: import errors (25) in /var/tmp/rapids_errors/SSN_1_DB1_242
```

11.13.3 Data Conversion Errors

As described in the section 11.13.1, when data conversion errors happen, a sub-folder will be created in the folder referenced by the “ERROR_PATH” property, with two files created in that sub-folder, a “*.log” log file with details of the errors and a “*.csv” file with the errant data records.

The following examples show how the errors are reported for an INSERT ... SELECT, bulk import using the “FILES” option and a bulk insert using the “FOLDERS” option.

Example 1 INSERT ... SELECT:

In this example, an attempt was made to insert a text field into an integer column:

```
rapids > create table moxe.REGION_B (
> r_regionkey integer NOT NULL,
> r_id integer,
> r_comment varchar(152)
> ) PARTITION(r_regionkey);
```

```
0 row(s) returned (0.10 sec)
```

```
rapids > insert into region_b SELECT * FROM (csv_header::  
'node://db1/SFSMALL/regionPipe.csv' );  
Error: import errors (5) in /var/tmp/rapids_errors/SSN_1_DB1_243
```

The error message above indicates that the log file containing a description of the errors will be in the log file:

```
"/var/tmp/rapids_errors/SSN_1_DB1_243/node__db1_var_tmp_rapids_SFSMALL_regionPipe_csv-  
messages.log"
```

```
[rapids@db1 SSN_1_DB1_243]$ cat  
/var/tmp/rapids_errors/SSN_1_DB1_243/node__db1_var_tmp_rapids_SFSMALL_region  
Pipe_csv-messages.log  
segment 0 line 1: java.lang.NumberFormatException: For input string: "UNITED  
STATES"  
segment 0 line 2: java.lang.NumberFormatException: For input string: "NORTH  
AMERICA,aldkjlakngkjangkijnfkngakldflkadjlkajdlkajd"  
segment 0 line 3: java.lang.NumberFormatException: For input string: "EUROPE"  
segment 0 line 4: java.lang.NumberFormatException: For input string: "SOUTH  
AMERICA"  
segment 0 line 5: java.lang.NumberFormatException: For input string: "ASIA"
```

The data associated with the error messages will be in the file

```
"/var/tmp/rapids_errors/SSN_1_DB1_243/node__db1_var_tmp_rapids_SFSMALL_regionPipe_csv-  
records.csv"
```

```
[rapids@db1 SSN_1_DB1_243]$ cat  
/var/tmp/rapids_errors/SSN_1_DB1_243/node__db1_var_tmp_rapids_SFSMALL_region  
Pipe_csv-records.csv  
1|UNITED STATES|adknladnganfbmanlgnalkfnlglkajglkafjglksjfglkajfgkjadg  
2|NORTH AMERICA|aldkjlakngkjangkijnfkngakldflkadjlkajdlkajd  
3|EUROPE|dxldgkzcsoisjoicnkjebfjhqwgrygwuihvokjdgojdvps  
4|SOUTH AMERICA|csvbdcavbscdbvacdhvjhxdkgnlkgflglksdnja shc asbvda  
5|ASIA|i4y5qiuyrqghrushfxhghirohtiehtaytaiuerytaiurt
```

The log file is indicating that there were conversion errors in the second data fields, where the data was a text string whereas the table was expecting an integer.

Example 2 Bulk IMPORT with "FILES" option:

This example shows how errors are reported when doing a bulk import using the "FILES" option:

```
rapids > IMPORT MOXE.* REPLACE FROM 'node://db1/tpch_small_files';  
Unexpected Exception:  
Import partially succeeded: succeeded: 7, failed: 1
```

```
Error limit (10) reached:
/var/tmp/rapids_errors/SSN_3_DB1_68/node___db1_var_tmp_rapids_tpch_small_files_nation_csv-messages.log
```

The error message above indicates that the log file containing a description of the errors will be in the file:

```
"/var/tmp/rapids_errors/SSN_3_DB1_68/node___db1_var_tmp_rapids_tpch_small_files_nation_csv-messages.log"
```

Below is the content of the log file:

```
[rapids@db1 rapids]$ cat
/var/tmp/rapids_errors/SSN_3_DB1_68/node___db1_var_tmp_rapids_tpch_small_files_nation_csv-messages.log
segment 0 line 1: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
segment 0 line 2: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
segment 0 line 3: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
segment 0 line 4: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
segment 0 line 5: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
segment 0 line 6: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
segment 0 line 7: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
segment 0 line 8: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
segment 0 line 9: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
segment 0 line 10: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
```

NOTE: The column number shown is zero-based, and so the error refers to the second field in the input data. In the above example, the input data with the reported error will be the second data field of the input record (column 1).

The data associated with the error messages will be in the file:

```
/var/tmp/rapids_errors/SSN_3_DB1_68/node___db1_var_tmp_rapids_tpch_small_files_nation_csv-records.csv
```

Contents of the above file, with the data fields in error hilited:

```
[rapids@db1 rapids]$ cat
/var/tmp/rapids_errors/SSN_3_DB1_68/node___db1_var_tmp_rapids_tpch_small_file
s_nation_csv-records.csv
0,ALGERIA,0, haggle. carefully final deposits detect slyly agai
7,GERMANY,3,1 platelets. regular accounts x-ray: unusual\, regular acco
24,UNITED STATES,1,y final packages. slow foxes cajole quickly. quickly
silent platelets breach ironic accounts. unusual pinto be
3,CANADA,1,eas hang ironic\, silent packages. slyly regular packages are
furiously over the tithes. fluffily bold
14,KENYA,0, pending excuses haggle furiously deposits. pending\, express
pinto beans wake fluffily past t
20,SAUDI ARABIA,4,ts. silent requests haggle. closely express packages sleep
across the blithely
2,BRAZIL,1,y alongside of the pending deposits. carefully special packages
are about the ironic forges. slyly special
22,RUSSIA,3, requests against the platelets use never according to the
quickly regular pint
5,ETHIOPIA,0,ven packages wake quickly. regu
12,JAPAN,2,ously. final\, express gifts cajole a
```

Example 3 Bulk IMPORT with “FOLDERS” option:

This example shows how errors are reported when doing a bulk import using the “FILES” option:

```
rapids > import MOXE.* FROM FOLDERS 'node://db1/tpch_small';
Unexpected Exception:
Import partially succeeded: succeeded: 7, failed: 1
  Error limit (10) reached:
/var/tmp/rapids_errors/SSN_3_DB1_69/node___db1_var_tmp_rapids_tpch_small_nati
on-messages.log
```

The error message above indicates that the log file containing a description of the errors will be in the file:

“/var/tmp/rapids_errors/SSN_3_DB1_69/node___db1_var_tmp_rapids_tpch_small_nation-messages.log”

and the associated data file will be in the file:

“/var/tmp/rapids_errors/SSN_3_DB1_69/node___db1_var_tmp_rapids_tpch_small_nation-records.csv”

As for the “FILES” option, the column number reported in the log file will be zero-based, for example:

```
segment 0 line 1: com.rapidsdata.impex.ImpexParseException: Invalid
characters between delimiter ( , ) and enclosed_by ( " ) column: 1, integer
```

Would refer to the second data field in the input data.

11.13.4 Mismatched Number of Fields and Columns on INSERT

The following error will be returned when the number of fields in the data file being imported does not match the number of columns in the target table:

Unexpected Exception:

Line 1 position 1: Column lists differ, x column(s) vs y column(s).

where, x is the number of columns in the import file and y is the number of columns in the target table

Example:

Target table with 3 columns:

```
rapids > create table moxe.bad1(c1 integer, c2 integer, c3 integer);
0 row(s) returned (0.09 sec)
rapids > insert into bad1 select * from
('node://db1/text/lead_trail_blanks.csv');
Unexpected Exception:
Line 1 position 1: Column lists differ, 4 column(s) vs 3 column(s).
```

Import file, with 4 fields:

```
[rapids@db1 text]$ cat lead_trail_blanks.csv
1, 4 leading blanks,3 trailing blanks ,1
2,A2345678901234567890,A1234567890123456789,2
3," 4 leading blanks","3 trailing blanks ",3
```

11.13.5 Wildcard import to multiple connectors

This error occurs when an attempt is made to do a bulk import operation where the target tables are managed by different Connectors, which is not allowed:

Example:

Below are the current tables that are managed by the “MOXE” and “MOXE2” Connectors:

```
rapids > show tables;
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME
-----
MOXE          MOXE         CUSTOMER
MOXE          MOXE         LINEITEM
MOXE          MOXE         NATION
MOXE          MOXE         ORDERS
MOXE          MOXE         REGION
MOXE          MOXE         SUPPLIER
MOXE2         MOXE2        PART
MOXE2         MOXE2        PARTSUPP
...
29 row(s) returned (0.23 sec)
```

Below is the bulk import command that will attempt to do imports against all of the tables shown above, which is not allowed because the tables are managed by two different Connectors. The error message

shows one table from each Connector which would have been imported into, which in this example are the “PART” table managed by the “MOXE2” Connector and “NATION” table managed by the “MOXE” Connector.

```
rapids > IMPORT * FROM FOLDERS 'node://db1/tpch_small';  
Unexpected Exception:  
Wildcard import to multiple Connectors: PART, NATION
```

12 REFRESH Command

The REFRESH command must be used when the underlying table metadata in a Data Store has been changed, where the change was not the result of a CREATE or DROP table request that was sent from RapidsDB (see section 10) and the user wishes to access the metadata information from RapidsDB. An example would be when a new table is created in Postgres using the native Postgres psql command interface.

The syntax for the refresh command is:

```
refresh [<Connector>];
```

If the Connector name is specified then the refresh command will only be applied to that Connector, if the Connector name is omitted then the refresh command will be applied to all enabled Connectors.

13 SYSTEM METADATA TABLES

13.1 OVERVIEW

RapidsDB provides a set of system metadata tables which provide metadata information about the RapidsDB system which is similar to the information provided by the ANSI Information Schema. The system metadata tables reside in the RAPIDS.SYSTEM catalog and schema. Each Federation has a METADATA Connector that maintains the system metadata tables for that Federation. The table below lists the system metadata tables:

Table Name	Description
NODES	A list of all of the nodes in the RapidsDB Cluster
FEDERATIONS	A list of all the Federations
CONNECTORS	A list of all of the Connectors in the current Federation
CATALOGS	A list of the catalogs that can be accessed from the current Federation
SCHEMAS	A list of the schemas that can be accessed from the current Federation

TABLES	Metadata for the tables and views that can be accessed from the current Federation.
INDEXES	Metadata for any indexes defined on tables that can be accessed from the current Federation.
COLUMNS	A list of all the columns that can be accessed from the current Federation
TABLE_PROVIDERS	A list of all of the tables from each Connector, including any duplicates.
AUTHENTICATORS	A list of all authenticator instances that have been created in the system.
AUTHENTICATOR_CONFIG	Lists any additional custom properties about the authenticator instances that have been created in the system.
USERS	A list of all users that exist in the system.
USER_CONFIG	Any additional custom properties about users that exist in the system.
SESSIONS	A list of all active sessions across the cluster.
USERNAME_MAPS	A list of defined mappings from an external identifier to RapidsDB usernames.
PATTERN_MAPS	A list of defined patterns for transforming an external identifier to a RapidsDB username.
QUERIES	A list of all the active queries
QUERY_STATS	Query statistics for all the active queries. This table is not fully operational as of this release and should be ignored

The system metadata tables are treated the same as any other tables by RapidsDB, and as for any user tables, it is only necessary to include the catalog and/or schema name when there are multiple tables in the current Federation that use the same name. Assuming that system metadata table names are all unique within the current Federation, then the following queries will all be successful:

- i) `select * from rapids.system.tables;`
- ii) `select * from system.tables;`
- iii) `select * from tables;`

13.2 NODES Table

The NODES table contains a list of the nodes in the RapidsDB Cluster. The table below shows the columns in the NODES table:

Column Name	Description
NODE_NAME	The name assigned by the user to this node
IS_DQC	Set to 'true' if this node is the DQC node, otherwise set to false
HOSTNAME	The host name or ip address for this node
CLIENT_PORT	The port number that the wireline protocol is listening on, which will be used by the RapidsDB Unified JDBC Driver for connecting to the RapidsDB cluster
CLUSTER_PORT	The port number that this node will be listening on
INSTALLATION_DIR	The installation directory for the RapidsDB Cluster software
WORKING_DIR	The working directory used for the RapidsDB Cluster software

Example:

```

rapids > select * from nodes;
NODE_NAME  IS_DQC  HOSTNAME      CLIENT_PORT  CLUSTER_PORT  INSTALLATION_DIR  WORKING_DIR
-----
DB1         true    192.168.1.98  4333         4334  /opt/rdp        /opt/rdp/current
DB2         false   192.168.1.76  4333         4334  /opt/rdp        /opt/rdp/current
DB3         false   192.168.1.56  4333         4334  /opt/rdp        /opt/rdp/current
DB4         false   192.168.1.126 4333         4334  /opt/rdp        /opt/rdp/current

4 row(s) returned (0.10 sec)

```

This example shows a 4 node RapidsDB Cluster with the node having ip address 192.168.1.98 being assigned the node name "DB1" and acting as the DQC node. The other node is a DQE node.

13.3 FEDERATIONS Table

The FEDERATIONS table contains a list of the Federations in the RapidsDB Cluster. The table below shows the columns in the FEDERATIONS table:

Column Name	Description
FEDERATION_NAME	The name of the Federation. By default there will always be one Federation named DEFAULTFED.
IS_DEFAULT	Set to 'true' if this is the default Federation, otherwise set to false

Example:

```

rapids > select * from federations;
FEDERATION_NAME    IS_DEFAULT
-----
DEFAULTFED         true

1 row(s) returned

```

13.4 CONNECTORS Table

The CONNECTORS table contains a list of the Connectors in the RapidsDB Cluster. The table below shows the columns in the CONNECTORS table:

Column Name	Description
FEDERATION_NAME	The name that of the Federation that this Connector belongs to
CONNECTOR_NAME	The name of the Connector
CONNECTOR_TYPE	The type of Connector, such as IMPEX, METADATA, MOXE or MYSQL
CONNECTOR_DDL	The CREATE CONNECTOR command that was used to create this Connector
IS_ENABLED	Set to 'true' if the Connector is enabled, otherwise it is set to false

The query below shows the sample output for querying this table:

```

rapids > select * from connectors;
FEDERATION_NAME    CONNECTOR_NAME    CONNECTOR_TYPE    IS_ENABLED    CONNECTOR_DDL
-----
DEFAULTFED         METADATA          METADATA          true          CREATE CONNECTOR METADATA TYPE METADATA NODE *
CATALOG * SCHEMA * TABLE *
DEFAULTFED         PARQSF10          HADOOP            true          CREATE CONNECTOR PARQSF10 TYPE HADOOP WITH
HDFS='hdfs://192.168.10.15:8020', FORMAT='parquet' NODE * CATALOG * SCHEMA * TABLE ORDERS (O_ORDERKEY INTEGER,
O_CUSTKEY INTEGER, O_ORDERSTATUS VARCHAR(1 CHARS), O_TOTALPRICE DECIMAL(15, 2), O_ORDERDATE DATE,
O_ORDERPRIORITY VARCHAR(15 CHARS), O_CLERK VARCHAR(15 CHARS), O_SHIPPRIORITY INTEGER, O_COMMENT VARCHAR(79
CHARS)) WITH USER='rapids', PATH='/user/rapids/warehouse/tpch/sf100/parquet/orders/' TABLE LINEITEM (L_ORDERKEY
INTEGER, L_PARTKEY INTEGER, L_SUPPKEY INTEGER, L_LINENUMBER INTEGER, L_QUANTITY DECIMAL(15, 2), L_EXTENDEDPRICE
DECIMAL(15, 2), L_DISCOUNT DECIMAL(15, 2), L_TAX DECIMAL(15, 2), L_RETURNFLAG VARCHAR(1 CHARS), L_LINESTATUS
VARCHAR(1 CHARS), L_SHIPDATE DATE, L_COMMITDATE DATE, L_RECEIPTDATE DATE, L_SHIPINSTRUCT VARCHAR(25 CHARS),
L_SHIPMODE VARCHAR(10 CHARS), L_COMMENT VARCHAR(44 CHARS)) WITH USER='rapids',
PATH='/user/rapids/warehouse/tpch/sf100/parquet/lineitem/'
DEFAULTFED         MOXE1             MOXE              true          CREATE CONNECTOR MOXE1 TYPE MOXE NODE * CATALOG
* SCHEMA * TABLE *
3 row(s) returned (0.06 sec)

```

In this example there are 3 Connectors in the DEFAULTFED Federation:

1. METADATA – manages the metadata for the DEFAULTFED Federation.
2. PARQSF10 – a Hadoop Connector
3. MOXE1 – a MOXE Connector

13.5 CATALOGS Table

The CATALOGS table contains a list of the catalogs in the current Federation. The table below shows the columns in the CATALOGS table:

Column Name	Description
CATALOG_NAME	The name of the catalog

The query below shows the sample output for querying this table:

```
rapids > select * from catalogs;
CATALOG_NAME
-----
MOXE_1
MYSQL_A
RAPIDS

3 row(s) returned
rapids >
```

In this example there are 3 catalogs:

1. MOXE_1 – this is the catalog for MOXE Connector named “MOXE_1”
2. MYSQL_A – this is the catalog for the MemSQL Connector named “MEM1”
3. RAPIDS – this is the catalog for the METADATA Connector

13.6 SCHEMAS Table

The SCHEMAS table contains a list of the schemas in the current Federation. The table below shows the columns in the SCHEMAS table:

Column Name	Description
CATALOG_NAME	The name of the catalog
SCHEMA_NAME	The name of the schema

The query below shows the sample output for querying this table:

```
rapids > select * from schemas;
CATALOG_NAME  SCHEMA_NAME
-----
MOXE_1        MOXE_1
MYSQL_A       CUSTOMER
RAPIDS        SYSTEM

3 row(s) returned
rapids >
```

In this example there are 3 schemas:

1. MOXE_1 – this is the schema for the MOXE Connector, “MOXE_1”
2. CUSTOMER – this is the schema for the MySQL Connector, “MYSQL_A”
3. SYSTEM – this is the schema for the Metadata Connector

13.7 TABLES Table

The TABLES table contains a list of the tables that can be accessed from the current Federation. The table below shows the columns in the TABLES table:

Column Name	Description
CATALOG_NAME	The name of the catalog for this table
SCHEMA_NAME	The name of the schema for this table
TABLE_NAME	The name of the table
IS_PARTITIONED	Set to 'true' if this table is partitioned, otherwise it is set to false
COMMENT	Comment for the table, if any
PROPERTIES	Indicates any properties associated with the table, such as the HDFS path name for tables managed by a Hadoop Connector

Example:

The following example shows tables from a MySQL database where there are comments on the columns and tables:

```
rapids > select * from tables where schema_name='test';
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME  IS_PARTITIONED  COMMENT
PROPERTIES
-----
test          test          COMMENTS     false           Test table for
comments      NULL
test          test          TESTING01    false           NULL
NULL
test          test          dctest       false           This is a test
table for comments  NULL

3 row(s) returned (0.05 sec)
```

13.8 INDEXES Table

The INDEXES table contains a list of the tables that can be accessed from the current Federation. The table below shows the columns in the INDEXES table:

Column Name	Description
CATALOG_NAME	The name of the catalog for this table

SCHEMA_NAME	The name of the schema for this table
TABLE_NAME	The name of the table
INDEX_NAME	The name of the index
IS_UNIQUE	true if the index is unique
IS_PRIMARY	true if the index is the primary key
INDEX_TYPE	The type of index
ORDINAL	The position of the column in the index
COLUMN_NAME	The column name

The example output below shows the index metadata for indexes defined on tables in MemSQL:

```
rapids > select * from indexes;
```

CATALOG_NAME	SCHEMA_NAME	TABLE_NAME	INDEX_NAME	IS_UNIQUE	IS_PRIMARY	INDEX_TYPE	ORDINAL	COLUMN_NAME
MEMSQL	TPCH	LINEITEM	PRIMARY	true	true	TREE	1	L_ORDERKEY
MEMSQL	TPCH	LINEITEM	PRIMARY	true	true	TREE	2	L_LINENUMBER
MEMSQL	TPCH	LINEITEM	li_com_dt_idx	false	false	TREE	1	L_COMMITDATE
MEMSQL	TPCH	LINEITEM	li_rcpt_dt_idx	false	false	TREE	1	L_RECEIPTDATE
MEMSQL	TPCH	LINEITEM	li_shp_dt_idx	false	false	TREE	1	L_SHIPDATE
MEMSQL	TPCH	LINEITEM	lineitem_fk1	false	false	TREE	1	L_ORDERKEY
MEMSQL	TPCH	LINEITEM	lineitem_fk2	false	false	TREE	1	L_SUPPKEY
MEMSQL	TPCH	LINEITEM	lineitem_fk3	false	false	TREE	1	L_PARTKEY
MEMSQL	TPCH	LINEITEM	lineitem_fk3	false	false	TREE	2	L_SUPPKEY
MEMSQL	TPCH	LINEITEM	lineitem_fk4	false	false	TREE	1	L_PARTKEY

13.9 COLUMNS Table

The COLUMNS table contains a list of the columns for all of the tables that can be accessed in the current Federation. The table below shows the columns in the COLUMNS table:

Column Name	Description
CATALOG_NAME	The name of the catalog for this table
SCHEMA_NAME	The name of the schema for this table
TABLE_NAME	The name of the table
COLUMN_NAME	The name of the column
DATA_TYPE	The data type for the column
ORDINAL	The column number (one-based)

IS_PARTITION_KEY	Set to 'true' if this column part of the partition (shard) key
IS_NULLABLE	True if the column is nullable
PRECISION	Precision for numerical columns
PRECISION_RADIX	If data_type identifies a numeric type, this column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10 as follows: INTEGER, FLOAT: 2 DECIMAL: 10
PRECISION_SCALE	Scale for decimal and float columns
CHARACTER_SET	Character set for column
COLLATION	Not used
COMMENT	Column comment
PROPERTIES	Properties associated with column

Example 1:

This example shows the column definition for the TPC-H "ORDERS" table:

```
rapids > select * from columns where table_name='ORDERS';
```

CATALOG_NAME	SCHEMA_NAME	TABLE_NAME	COLUMN_NAME	DATA_TYPE	PRECISION_RADIX	PRECISION_SCALE	CHARACTER_SET	COLLATION	COMMENT	PROPERTIES
MOXE	MOXE	ORDERS	O_ORDERKEY	INTEGER	2	64	NULL	NULL	serial:Kind=integer64	NULL NULL
MOXE	MOXE	ORDERS	O_CUSTKEY	INTEGER	2	64	NULL	NULL	serial:Kind=integer64	NULL NULL
MOXE	MOXE	ORDERS	O_ORDERSTATUS	VARCHAR	NULL	NULL	UTF16	BINARY	serial:Kind=stringRA2	NULL NULL
MOXE	MOXE	ORDERS	O_TOTALPRICE	DECIMAL	10	17	NULL	NULL	serial:Kind=decimal64	2 NULL

```

MOXE          MOXE          ORDERS          O_ORDERDATE          DATE
4            false         true            NULL                  NULL  NULL NULL
NULL          NULL          serial:Kind=timestamp
MOXE          MOXE          ORDERS          O_ORDERPRIORITY      VARCHAR
5            false         true            NULL                  NULL  NULL
UTF16         BINARY          NULL            serial:Kind=stringRA2
MOXE          MOXE          ORDERS          O_CLERK              VARCHAR
6            false         true            NULL                  NULL  NULL
UTF16         BINARY          NULL            serial:Kind=stringRA2
MOXE          MOXE          ORDERS          O_SHIPPRIORITY        INTEGER
7            false         true            64                    2    NULL NULL
NULL          NULL          serial:Kind=integer64
MOXE          MOXE          ORDERS          O_COMMENT            VARCHAR
8            false         true            NULL                  NULL  NULL
UTF16         BINARY          NULL            serial:Kind=stringRA2

9 row(s) returned (0.05 sec)

```

Example 2

This example shows a table with column comments:

```

rapids > select * from columns where table_name='COMMENTS';
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME  COLUMN_NAME  DATA_TYPE
ORDINAL      IS_PARTITION_KEY  IS_NULLABLE  PRECISION  PRECISION_RADIX
SCALE CHARACTER_SET  COLLATION  COMMENT  PROPERTIES
-----
test         test         COMMENTS    C1         INTEGER
0           false         true        64         2
NULL NULL          NULL        Integer column  NULL
test         test         COMMENTS    C2         DECIMAL
1           false         true        15         10
2 NULL          NULL        decimal column  NULL
test         test         COMMENTS    C3         FLOAT
2           false         true        53         2
NULL NULL          NULL        float column  NULL

3 row(s) returned (0.05 sec)

```

13.10 TABLE_PROVIDERS Table

The table below shows the columns in the TABLE_PROVIDERS table:

Column Name	Description
-------------	-------------

CATALOG_NAME	The name of the catalog for this table
SCHEMA_NAME	The name of the schema for this table
TABLE_NAME	The name of the table
CONNECTOR_NAME	The name of the Connector that is managing access to this table

```

rapids > select * from table_providers;
CATALOG_NAME  SCHEMA_NAME  TABLE_NAME  CONNECTOR_NAME
-----
RAPIDS        SYSTEM       FEDERATIONS  METADATA
RAPIDS        SYSTEM       CONNECTORS   METADATA
RAPIDS        SYSTEM       NODES        METADATA
RAPIDS        SYSTEM       CATALOGS     METADATA
RAPIDS        SYSTEM       SCHEMAS      METADATA
RAPIDS        SYSTEM       TABLES      METADATA
RAPIDS        SYSTEM       COLUMNS     METADATA
RAPIDS        SYSTEM       TABLE_PROVIDERS METADATA
RAPIDS        SYSTEM       INDEXES      METADATA
RAPIDS        SYSTEM       QUERY_STATS  METADATA
RAPIDS        SYSTEM       QUERIES      METADATA
RAPIDS        SYSTEM       SESSIONS     METADATA
RAPIDS        SYSTEM       USERS        METADATA
RAPIDS        SYSTEM       USER_CONFIG  METADATA
RAPIDS        SYSTEM       AUTHENTICATORS METADATA
RAPIDS        SYSTEM       AUTHENTICATOR_CONFIG METADATA
RAPIDS        SYSTEM       USERNAME_MAPS METADATA
RAPIDS        SYSTEM       PATTERN_MAPS METADATA
PARQSF10     PUBLIC       ORDERS       PARQSF10
PARQSF10     PUBLIC       LINEITEM     PARQSF10
MOXE1        MOXE1       T1           MOXE1
MOXE1        MOXE1       T2           MOXE1

22 row(s) returned (0.11 sec)

```

13.11 AUTHENTICATORS Table

The table below shows the columns in the AUTHENTICATORS table:

Column Name	Description
AUTHNAME	The name of the authenticator instance.
TYPE	The name of the type of authenticator.
ENABLED	Whether the authenticator is enabled or disabled.
DDL	The DDL string to recreate this authenticator.

```

rapids > select * from authenticators;
AUTHNAME  TYPE  ENABLED DDL
-----
RDPAUTH   RDP   true    CREATE AUTHENTICATOR RDPAUTH TYPE RDP ;

```



```
KRB      KERBEROS      true CREATE AUTHENTICATOR KRB TYPE KERBEROS WITH REALM = 'HOME';
```

13.12 AUTHENTICATOR_CONFIG Table

The table below shows the columns in the AUTHENTICATOR_CONFIG table:

Column Name	Description
AUTHNAME	The name of the authenticator instance.
KEY	The name of the custom property for this authenticator instance.
VALUE	The value of the custom property for this authenticator instance.

```
rapids > select * from authenticator_config;
AUTHNAME  KEY                VALUE
-----  ---                -
RDPAUTH   rdp.authenticator.name  RDPAUTH
RDPAUTH   rdp.authenticator.type   RDP
KRB       REALM              HOME
KRB       rdp.authenticator.name   KRB
KRB       rdp.authenticator.type   KERBEROS
```

13.13 USERS Table

The table below shows the columns in the USERS table:

Column Name	Description
USERNAME	The unique name of the user.
ENABLED	Whether the user is enabled or disabled.
AUTHNAME	The name of the authenticator instance this user is associated with.

```
rapids > select * from users;
USERNAME  ENABLED AUTHNAME
-----  -
CRAIG     true    KRB
RAPIDS    true    RDPAUTH
john      true    RDPAUTH
```

13.14 USER_CONFIG Table

The table below shows the columns in the USER_CONFIG table:

Column Name	Description
-------------	-------------

USERNAME	The username.
KEY	The name of the custom property for this authenticator instance.
VALUE	The value of the custom property for this authenticator instance.

```

rapids > select * from user_config;
USERNAME  KEY          VALUE
-----  ---          -
CRAIG     PRINCIPAL   craig@HOME

```

13.15 SESSIONS Table

The table below shows the columns in the SESSIONS table:

Column Name	Description
SESSION_ID	The unique name of the session across the cluster.
USERNAME	The username that the client has authenticated as, or null.
NODE	The node that the client connected to.
CLIENT_IP	The IP address of the client.
CLIENT_PORT	The port address that the client is connecting from.
SERVER_PORT	The port address that the client is connected to.
ESTABLISHED	The timestamp when the client first connected.

```

rapids > select * from sessions;
SESSION_ID  USERNAME  NODE  CLIENT_IP  CLIENT_PORT  SERVER_PORT  ESTABLISHED
-----  -
S1@NODE1    RAPIDS    NODE1  127.0.0.1  50547        4333 2019-04-18 07:00:22.376

```

13.16 USERNAME_MAPS Table

The table below shows the columns in the USERNAME_MAPS table:

Column Name	Description
ID	The external identifier to map from.
USERNAME	The RapidsDB username to map this external identifier to.

```

rapids > select * from username_maps;

```

ID	USERNAME
--	-----
craig@HOME	CRAIG

13.17 PATTERN_MAPS Table

The table below shows the columns in the PATTERN_MAPS table:

Column Name	Description
PRIORITY	The order in which the pattern mapping is tried (highest first).
SEARCH	The pattern to test against the external user ID.
REPLACE	The replacement pattern to be applied against the external user ID in conjunction with the search pattern.

```

rapids > select * from pattern_maps;
  PRIORITY SEARCH                                REPLACE
  -----
    100 ^(.+)/admin@COMPANY.COM$                ADMIN
    90 ^(.+?)(/[^@]*)?@COMPANY.COM$            $1
    80 ^(.+?)(/[^@]*)?@EXAMPLE.COM$            $1_EXAMPLE

```

13.18 QUERIES Table

The table below shows the columns in the QUERIES table:

Column Name	Description
QUERY_ID	The query id
SESSION_ID	The session id where this query is running
NODE	The RapidsDB node where the query was started
USERNAME	The name of the user running this query
START_TIME	The timestamp when the query was started
QUERY_TEXT	The SQL query

When querying the QUERIES table, only the queries submitted by the current user will be displayed unless the userid is "RAPIDS" in which case the queries for all users will be displayed.

Example:

Below is a query submitted from the rapids-shell:

```
rapids > select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price,
>          sum(l_extendedprice*(1-l_discount)) as sum_disc_price, sum(l_extendedprice*(1-
l_discount)*(1+l_tax)) as sum_charge,
>          avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc,
count(*) as count_order
>   from LINEITEM
>  where l_shipdate <= timestamp '1998-12-01 00:00:00' - interval '90' day
>  group by l_returnflag, l_linestatus
>  order by l_returnflag, l_linestatus;
```

Below is the content of the QUERIES table while this query is running:

```
rapids > select * from queries;
QUERY_ID      SESSION_ID      NODE      USERNAME      START_TIME      QUERY_TEXT
-----
SSN_3@BORAY01#7  SSN_3@BORAY01  BORAY01  RAPIDS        2020-07-30 21:17:11.608  select * from queries;
SSN_1@BORAY01#32  SSN_1@BORAY01  BORAY01  RAPIDS        2020-07-30 21:17:09.838  select l_returnflag,
l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice*(1-
l_discount)) as sum_disc_price, sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge, avg(l_quantity) as
avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count(*) as count_order from LINEITEM
where l_shipdate <= timestamp '1998-12-01 00:00:00' - interval '90' day group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;

2 row(s) returned (0.21 sec)
```

14 Cancelling a Query

There are three ways that an active query can be cancelled as explained in the following sections.

14.1 rapids-shell

If the query was started from the rapids-shell then the user can enter Ctrl-k from the rapids-shell window where the query is running. This will result in a message being sent to RapidsDB to cancel the query currently running on that connection. This requires rapids-shell version 4 and the RapidsDB JDBC Driver version 4, both of which are included with this release.

Example:

```
rapids > select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price,
>          sum(l_extendedprice*(1-l_discount)) as sum_disc_price, sum(l_extendedprice*(1-
l_discount)*(1+l_tax)) as sum_charge,
>          avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc,
count(*) as count_order
>   from LINEITEM
>  where l_shipdate <= timestamp '1998-12-01 00:00:00' - interval '90' day
>  group by l_returnflag, l_linestatus
>  order by l_returnflag, l_linestatus;
```

Checking the QUERIES table from another window shows this query running:

```

rapids > select * from queries;
QUERY_ID      SESSION_ID      NODE      USERNAME      START_TIME      QUERY_TEXT
-----
SSN_3@BORAY01#7  SSN_3@BORAY01  BORAY01  RAPIDS        2020-07-30 21:17:11.608  select * from queries;
SSN_1@BORAY01#32  SSN_1@BORAY01  BORAY01  RAPIDS        2020-07-30 21:17:09.838  select l_returnflag,
l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice*(1-
l_discount)) as sum_disc_price, sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge, avg(l_quantity) as
avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count(*) as count_order from LINEITEM
where l_shipdate <= timestamp '1998-12-01 00:00:00' - interval '90' day group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;

2 row(s) returned (0.21 sec)

```

Press Ctrl-K:

```

[CANCELLING QUERY]
Cancellation of query SSN_1@BORAY01#33 requested.
[CANCELED]
rapids>

```

Checking the QUERIES table shows this query no longer running:

```

rapids > select * from queries;
QUERY_ID      SESSION_ID      NODE      USERNAME      START_TIME      QUERY_TEXT
-----
SSN_2@BORAY01#1  SSN_2@BORAY01  BORAY01  RAPIDS        2020-07-30 21:18:11.365  select * from queries;

1 row(s) returned (0.14 sec)

```

14.2 JDBC

Programmatically via the RapidsDB JDBC Driver using the Statement.cancel() interface:

- When a Statement instance is being executed, a second thread can call the cancel() method on it. This will cause the JDBC driver to create a temporary connection to the server, authenticate and issue a query cancellation. When the query is cancelled, the execution of the Statement object will return with a JDBC SQLException indicating that the query was cancelled.
- The call to Statement.cancel() will return once the cancellation request has been submitted to RapidsDB. This is not necessarily the same time that the cancelled query actually exits early.
- If Statement.cancel() is called on a statement that has already completed or hasn't been executed in RapidsDB yet then an error is returned to the caller.
- Requires RapidsDB JDBC driver version 4.

14.3 CANCEL QUERY command

Using the SQL command CANCEL QUERY.

The syntax for the CANCEL QUERY command is:

```
CANCEL QUERY [ IF EXISTS ] <queryId>;
```

- The <queryId> can be found from the QUERIES metadata table (see 13.18).

- Queries can be cancelled on remote nodes as well as the local node.
- Only the user that initiated the query can cancel the query, unless the user is the RAPIDS user in which case that user can cancel any query

Example:

Example: In the example below the query is initiated on Session 1, and then cancelled from a different session, Session 2.

Session 1:

```
rapids > select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price,
>          sum(l_extendedprice*(1-l_discount)) as sum_disc_price, sum(l_extendedprice*(1-
l_discount)*(1+l_tax)) as sum_charge,
>          avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc,
count(*) as count_order
>   from LINEITEM
>  where l_shipdate <= timestamp '1998-12-01 00:00:00' - interval '90' day
>  group by l_returnflag, l_linestatus
>  order by l_returnflag, l_linestatus;
```

Session 2 – from another session, list the currently active queries and then cancel the query started from session 1:

```
rapids > select * from queries;
QUERY_ID          SESSION_ID      NODE      USERNAME      START_TIME      QUERY_TEXT
-----
SSN_3@BORAY01#7   SSN_3@BORAY01  BORAY01   RAPIDS        2020-07-30 21:17:11.608  select * from queries;
SSN_1@BORAY01#32  SSN_1@BORAY01  BORAY01   RAPIDS        2020-07-30 21:17:09.838  select l_returnflag,
l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice*(1-
l_discount)) as sum_disc_price, sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge, avg(l_quantity) as
avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count(*) as count_order from LINEITEM
where l_shipdate <= timestamp '1998-12-01 00:00:00' - interval '90' day group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;

2 row(s) returned (0.21 sec)
rapids > cancel query SSN_1@BORAY01#32 ;
0 row(s) returned (0.70 sec)
```

Session 1 – this is the exception returned on session 1 after the query is cancelled:

```
rapids > select l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price,
>          sum(l_extendedprice*(1-l_discount)) as sum_disc_price, sum(l_extendedprice*(1-
l_discount)*(1+l_tax)) as sum_charge,
>          avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc,
count(*) as count_order
>   from LINEITEM
>  where l_shipdate <= timestamp '1998-12-01 00:00:00' - interval '90' day
>  group by l_returnflag, l_linestatus
>  order by l_returnflag, l_linestatus;
[CANCELLED]
rapids>
```

15 Performance Tuning

15.1 EXPLAIN

EXPLAIN instructs the rapids-shell to output a schematic representation of the query plan for the associated statement, including the SQL queries that will be sent to the underlying Data Store and the internal operators and routing for operations that will be performed by the RapidsDB Execution Engine.

Syntax:

```
EXPLAIN <SQL statement>
```

Example 1 – simple select

```
rapids > explain select sum(o_totalprice) from orders group by o_orderkey;
QUERY_PLAN
-----
Execute "select sum(o_totalprice) from orders group by o_orderkey;"
P2:Project 01
A1:Aggregate {O_ORDERKEY} SUM(O_TOTALPRICE) `01`
P1:Project ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE
T1:Stream ORDERS

1 row(s) returned (0.83 sec)
```

Example 2 – 1-way join

```
rapids > explain select o_orderkey from orders, lineitem where o_orderdate=l_commitdate and
o_orderdate < '2019-10-20 09:00:00';
QUERY_PLAN
-----
Execute "select o_orderkey from orders, lineitem where o_orderdate=l_commitdate and
o_orderdate < '2019-10-20 09:00:00';"
P4:Project O_ORDERKEY
P3:Project O_ORDERKEY

J1:LpJoin O_ORDERDATE = L_COMMITDATE
F1:Filter 012
P1:Project ORDERS.O_ORDERDATE, ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE < 2019-10-20 `012`
T1:Stream ORDERS

F2:Filter L18
P2:Project LINEITEM.L_COMMITDATE, LINEITEM.L_COMMITDATE < 2019-10-20 `L18`
T2:Stream LINEITEM

1 row(s) returned (1.94 sec)
```

15.2 JOIN Order

For SELECT statements with JOINS the user should order the tables in the query from left to right so that the table with the smallest number of rows, given the supplied WHERE clause predicates, is to the left and the table with the largest number of rows is the last table. For example, with the following query:

```

SELECT l_orderkey,
       SUM(l_extendedprice) AS revenue,
       o_orderdate,
       o_shippriority
FROM   lineitem
       join orders
       ON l_orderkey = o_orderkey
       join customer
       ON c_custkey = o_custkey
WHERE  c_mktsegment = 'FURNITURE'
       AND o_orderdate < '2014-05-01 12:00:00'
       AND l_shipdate > '2014-04-01 12:00:00'
GROUP BY l_orderkey,
         o_orderdate,
         o_shippriority;

```

If the rows returned from the customer table are the smallest and the rows returned from the lineitem table are the largest, then the query should be changed to the following in order to achieve the best query performance:

```

SELECT l_orderkey,
       SUM(l_extendedprice) AS revenue,
       o_orderdate,
       o_shippriority
FROM   customer
       join orders
       ON c_custkey = o_custkey
       join lineitem
       ON l_orderkey = o_orderkey
WHERE  c_mktsegment = 'FURNITURE'
       AND o_orderdate < '2014-05-01 12:00:00'
       AND l_shipdate > '2014-04-01 12:00:00'
GROUP BY l_orderkey,
         o_orderdate,
         o_shippriority;

```

15.3 Restrict Amount of Data

Care should be taken to restrict the amount of data in JOINS by using the appropriate predicates in the WHERE clause. Failure to do this could result in too much data being requested from the Data Store

which could cause one or more of the DQE nodes to fail due to exhausting their heaps. In the previous example, the timestamps in the WHERE clause should be further restricted, for example:

```
WHERE c_mktsegment = 'FURNITURE'  
AND o_orderdate < '2014-05-01 12:00:00'  
AND o_orderdate > '2014-04-01 12:00:00'  
AND l_shipdate < '2014-05-01 12:00:00'  
AND l_shipdate > '2014-04-01 12:00:00'
```

16 Error Messages

16.1 RapidsDB shell Messages

- **Failed to submit statement**

The RapidsDB shell program was unable to send the statement to the DQC. The likely cause is a failed node or a network problem. The bootstrapper (refer to the RapidsDB Installation Manual) HEALTHCHECK option can be used to verify that the cluster is operating normally.

- **System exception on node <hostname:port>: <exception>: <message>**

A system exception occurred. This most likely indicates a software issue but may occur because of a network problem. Before resubmitting the query, the bootstrapper HEALTHCHECK option should be used to verify that the cluster is operating normally. (Note: system exceptions are normally followed by "traceback" information. If possible, this information should be preserved and included with any bug report.)

- **DQS exception on node <hostname:port>: <exception>: <message>**

An exception occurred while processing the query. The query can be retried.

- **Subsystem exception on node <hostname:port>: <exception>: <message>**

An exception occurred while processing the query. The query can be retried. The most common source of subsystem errors is exceptions from the underlying Data Store(see Data Store messages below).

16.2 Query Rejection Messages

In addition to the above, statements submitted to the RapidsDB shell may be rejected with any of the messages below.

- **Unrecognized token**

A token (i.e. keyword or term) in the query was misplaced, misspelled or contained illegal characters.

- **Syntax error near <token>**

The parser could not recognize the syntax of the statement. The last token (i.e. keyword or term) recognized is shown.

Check that the structure of the statement is correct, keywords and table names are correctly spelled and no table or columns have the same name as a SQL keyword.

- **Syntax error at <token> expected <tokens>**

The parser could not recognize the syntax of the statement. A recognizable token (i.e. keyword or term) was encountered, but it was not one of the tokens expected. Check that the structure of the statement is

correct, keywords and table names are correctly spelled and no table or columns have the same name as a SQL keyword.

- **Reference to undefined table or stream: <name>**

The statement refers to a table name that is neither a table in the database nor a properly defined table or subquery alias within the query. Check that all table names or aliases are correctly spelled.

- **Reference to undefined column: <name>**

The statement refers to a variable or column name that is not defined in any of the tables or subqueries in the statement.

Check that column names are correctly spelled.

- **Ambiguous reference to table or stream: <name>**

The statement refers to table name that is defined in more than catalog and schema. The table name must be qualified with either the schema name or the catalog and schema name to disambiguate the table name.

- **Ambiguous reference to column: <name>**

The statement refers to a variable or column name that is defined in more than one table or subquery. Qualify the column name with a table name or a table or subquery alias.

- **Type mismatch in <name>**

The arguments to the named operator or function were of the wrong type. For details on the number and type of function arguments (see 3.2 for more information on operators and functions).

- **Wrong number of arguments for function: <name>**

The wrong number of arguments were supplied for the named function. For details on the number and type of function arguments, (see 3.2 for more information on operators and functions).

- **Unsupported operator or function: <name>**

The named operator or function is valid in SQL but is not supported in RapidsDB.

- **JOIN predicate must be boolean expression**

The expression in the ON clause of a join must be of boolean type (i.e. must produce a true or false result).

- **WHERE predicate must be boolean expression**

The expression in the WHERE clause must be of boolean type (i.e. must produce a true or false result).

- **FULL OUTER join not supported**

RapidsDB does not support full outer joins. Only inner joins and left or right outer joins are supported.

- **Invalid GROUP BY expression**

The expression specified in the GROUP BY clause contains an aggregate function (COUNT, MIN, MAX, SUM or AVG). Only scalar operators and functions can be used in GROUP BY.

- **Expression not in aggregate of GROUP BY column.**

In a query or subquery that specifies aggregation (COUNT, MIN, MAX, SUM or AVG functions) and/or grouping (GROUP BY clause), each expression in the SELECT list must be either an aggregate or a duplicate of one of the GROUP BY expressions. In other words, each SELECT expression must produce only a single value per group (or, if there is no grouping, a single value for the entire query).

- **Invalid column index in ORDER BY**

The ORDER BY <column number> clause was used but the specified column number is greater than the number of columns produced by the query or subquery.

- **LIMIT value must not be negative**

The LIMIT clause specified a negative limit. The value in the LIMIT clause must be zero or positive.

- **OFFSET value must not be negative**

The OFFSET clause specified a negative offset. The value in the OFFSET clause must be zero or positive.

- **Select list count mismatch**

The statement contains a UNION clause in which the query on the left side produces a different number of columns than the query on the right side. The queries on either side of a UNION must produce a matching number of columns.

- **Select list type mismatch**

The statement contains a UNION clause in which the query on the left side produces columns whose types are incompatible with the columns produced by the query on the right side. The queries on either side of a UNION must produce columns with matching types. Any numeric type is considered a match for any other numeric type. All other types must match exactly.

16.3 Data Store-Related Messages

Note: messages related to the underlying Data Store may occur as either RapidsDB Exceptions or Subsystem Exceptions (see above).

- **Timed out waiting for queries sent to VoltDB to complete**

A query that was sent to the Data Store took an excessive amount of time to produce a result. The query can be retried. This may indicate that Data Store is too heavily loaded on one or more hosts. If the error recurs, consider reducing the number of concurrent queries (to reduce the number of queries that are concurrently executed by RapidsDB, change -q option passed to the bootstrapper – see RapidsDB Installation and Management Manual).

- **No connections.**

The Data Store has disconnected from one or more RapidsDB nodes due to inactivity. The query can be retried.

- **SQL ERROR More than nnn MB of temp table memory used while executing SQL.**

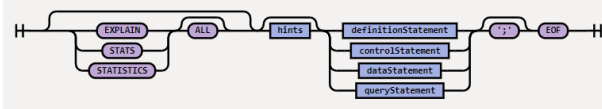
The intermediate results of the query are too large for the internal tables in the Data Store. You can adjust the maximum size of the internal tables in the Data Store to accommodate the query in the Data Store deployment file. (For more information, see RapidsDB Installation and Management Manual)

- **SQL ERROR Output from SQL stmt overflowed output/network buffer of 50mb**

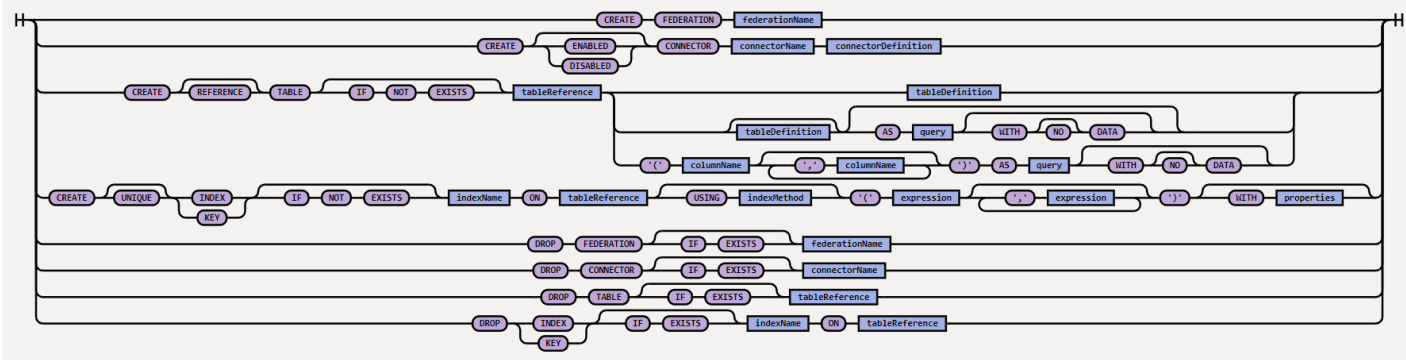
The final result of the query is too large for the Data Store output buffers. You may be able to limit the size of the result using the LIMIT clause or break the query into sections using the LIMIT, OFFSET and UNION clauses.

Appendix A SQL Grammar

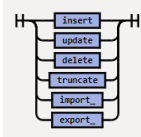
statement



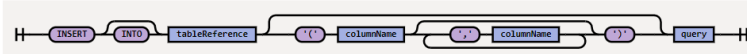
definitionStatement



dataStatement



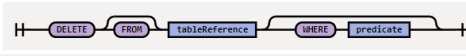
insert



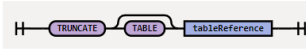
update



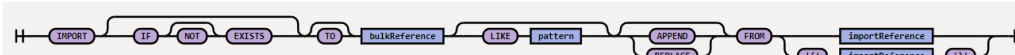
delete



truncate



import_



export_

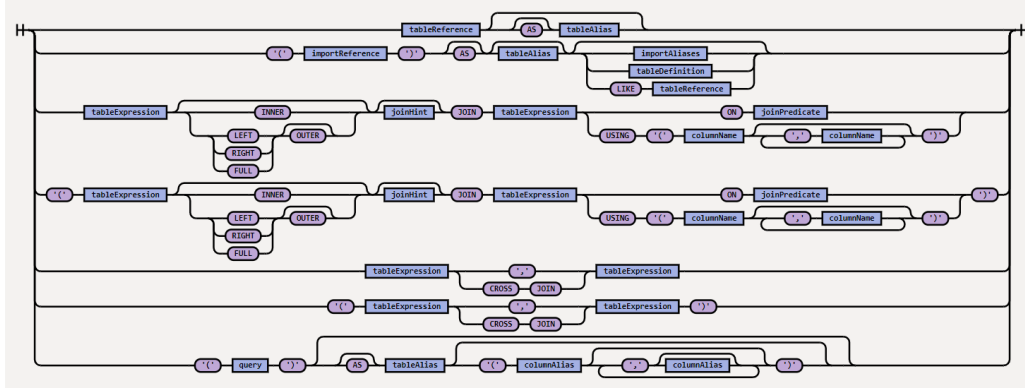


queryStatement





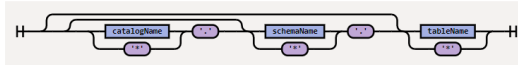
tableExpression



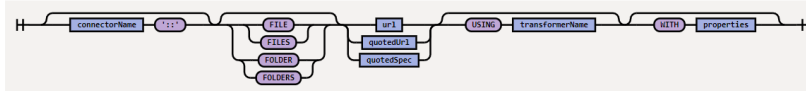
tableReference



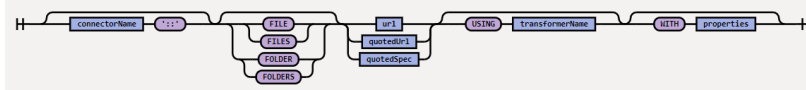
bulkReference



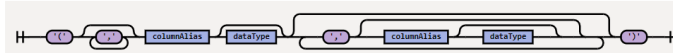
importReference



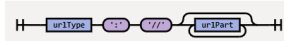
exportReference



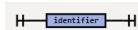
importAliases



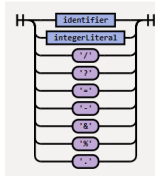
uri



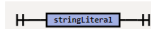
uriType



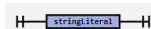
uriPart



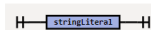
quotedUri



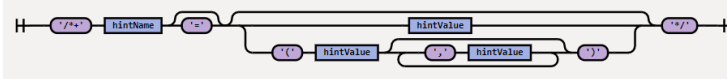
quotedSpec



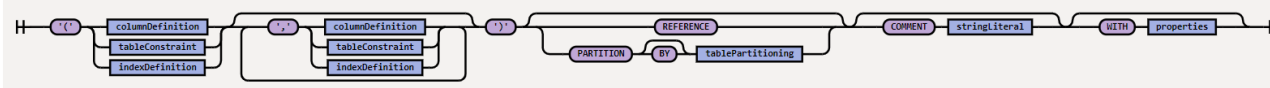
pattern



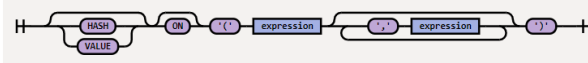
joinHint



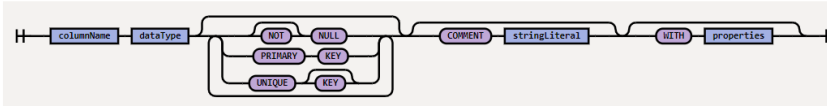
tableDefinition



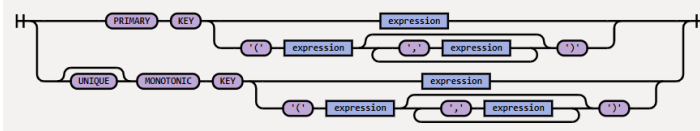
tablePartitioning



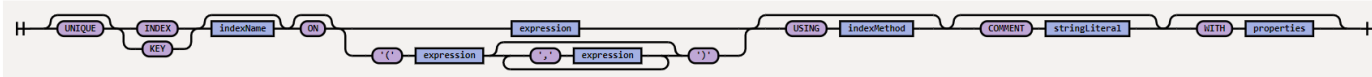
columnDefinition



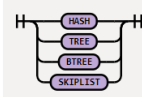
tableConstraint



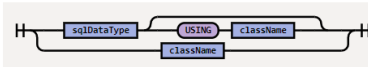
indexDefinition



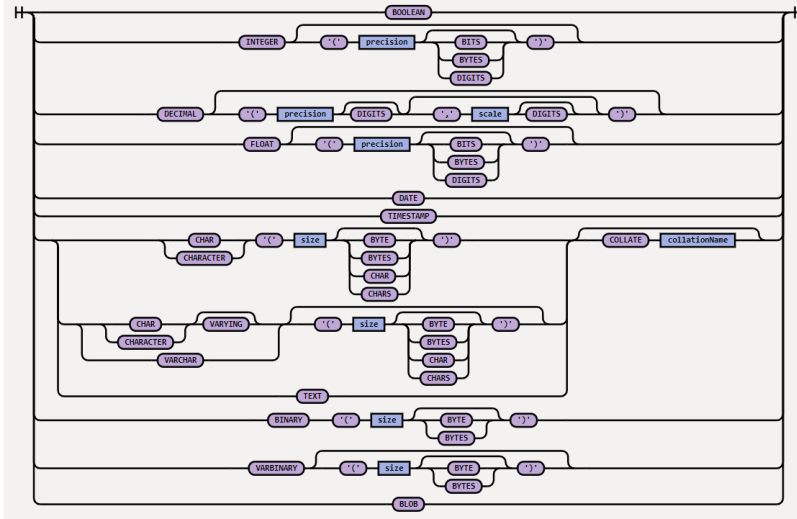
indexMethod



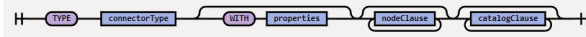
dataType



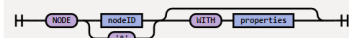
sqlDataType



connectorDefinition



nodeClause



catalogClause



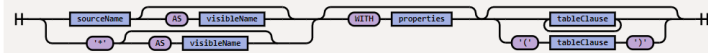
catalogDescriptor



schemaClause



schemaDescriptor



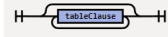
tableClause



tableDescriptor



freeSchema



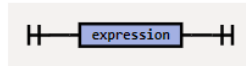
properties



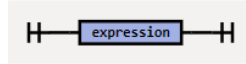
property



predicate



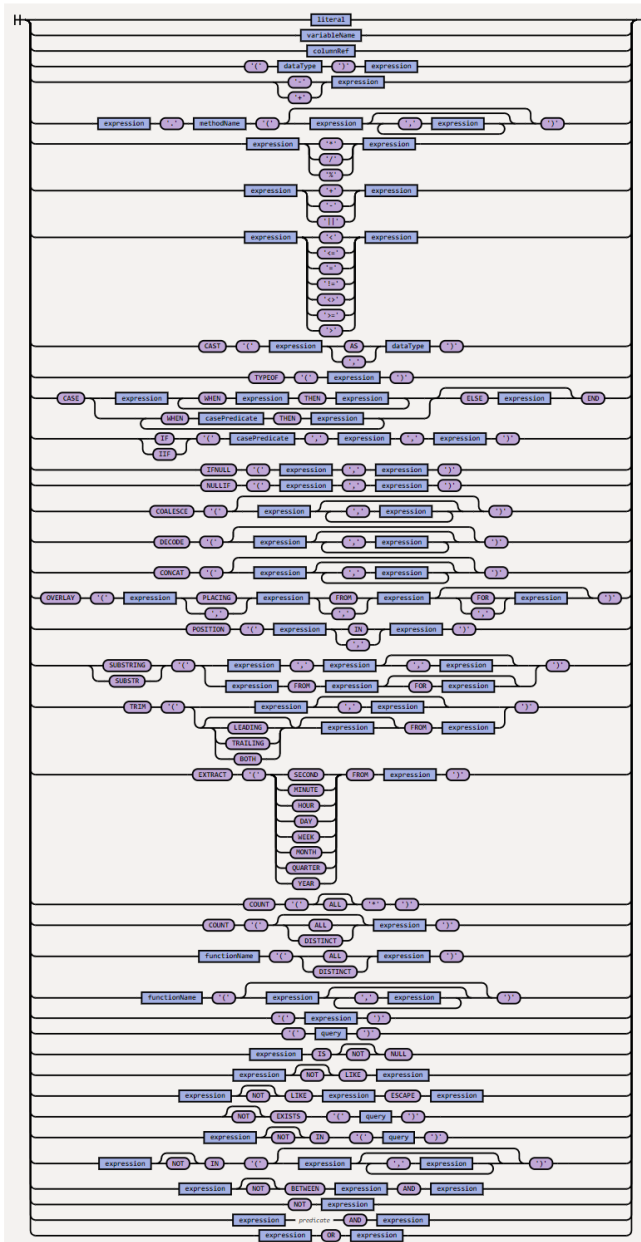
joinPredicate




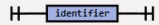
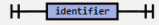
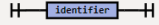
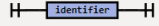
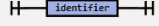
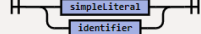
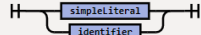
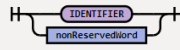
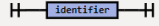
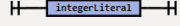
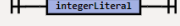
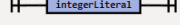
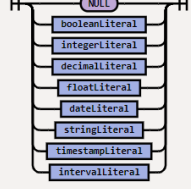
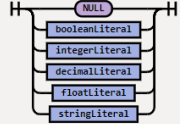
casePredicate



expression

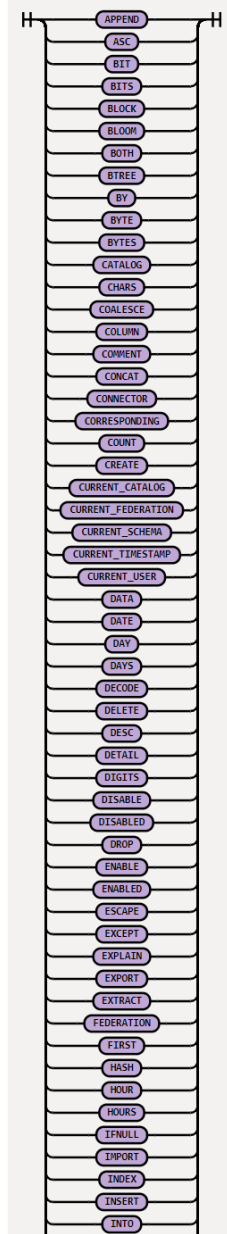


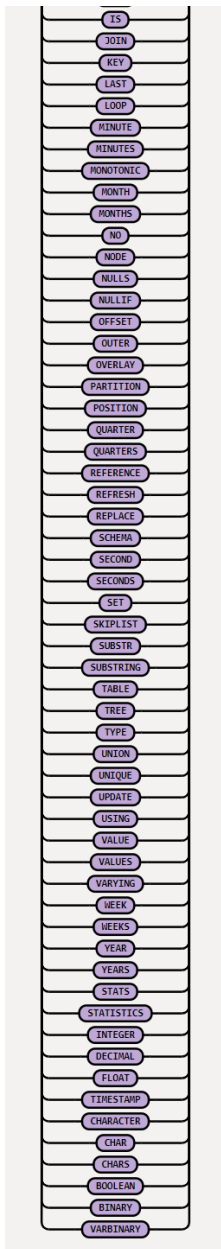
variableName	
columnRef	
columnWildcard	
subscript	
subscriptRange	
fieldName	
subfieldName	
className	
functionName	
methodName	
connectorType	
nodeID	
federationName	
connectorName	
transformerName	
catalogName	
schemaName	
sourceName	
visibleName	
withQueryName	
tableName	

tableAlias	
columnName	
columnAlias	
collationName	
indexName	
propertyName	
hintName	
propertyValue	
hintValue	
identifier	
qualifier	
precision	
scale	
size	
literal	
simpleLiteral	



nonReservedWord





AND	H—A—N—D—H
ANY	H—A—N—Y—H
APPEND	H—A—P—P—E—N—D—H
AS	H—A—S—H
ASC	H—A—S—C—H
BETWEEN	H—B—E—T—W—E—E—N—H
BOTH	H—B—O—T—H—H
BY	H—B—Y—H
CATALOG	H—C—A—T—A—L—O—G—H
COLLATE	H—C—O—L—L—A—T—E—H
COLUMN	H—C—O—L—U—M—N—H
COMMENT	H—C—O—M—M—E—N—T—H
CONNECTOR	H—C—O—N—N—E—C—T—O—R—H
CORRESPONDING	H—C—O—R—R—E—S—P—O—N—D—I—N—G—H
CREATE	H—C—R—E—A—T—E—H
CROSS	H—C—R—O—S—S—H
CURRENT_CATALOG	H—C—U—R—R—E—N—T—C—A—T—A—L—O—G—H
CURRENT_FEDERATION	H—C—U—R—R—E—N—T—F—E—D—E—R—A—T—I—O—N—H
CURRENT_SCHEMA	H—C—U—R—R—E—N—T—S—C—H—E—M—A—H
CURRENT_TIMESTAMP	H—C—U—R—R—E—N—T—T—I—M—E—S—T—A—M—P—H
CURRENT_USER	H—C—U—R—R—E—N—T—U—S—E—R—H

DATA	H—D—A—T—A—H
DATE	H—D—A—T—E—H
DELETE	H—D—E—L—E—T—E—H
DESC	H—D—E—S—C—H
DETAIL	H—D—E—T—A—I—L—H
DISABLED	H—D—I—S—A—B—L—E—D—H
DISABLE	H—D—I—S—A—B—L—E—H
DISTINCT	H—D—I—S—T—I—N—C—T—H
DROP	H—D—R—O—P—H
ENABLED	H—E—N—A—B—L—E—D—H
ENABLE	H—E—N—A—B—L—E—H
ESCAPE	H—E—S—C—A—P—E—H
EXCEPT	H—E—X—C—E—P—T—H
EXPLAIN	H—E—X—P—L—A—I—N—H
EXPORT	H—E—X—P—O—R—T—H
FALSE	H—F—A—L—S—E—H
FEDERATION	H—F—E—D—E—R—A—T—I—O—N—H
FILE	H—F—I—L—E—H
FILES	H—F—I—L—E—S—H
FIRST	H—F—I—R—S—T—H
FOLDER	H—F—O—L—D—E—R—H
FOLDERS	H—F—O—L—D—E—R—S—H
FOR	H—F—O—R—H

FULL	H—F—U—L—L—H
FROM	H—F—R—O—M—H
GROUP	H—G—R—O—U—P—H
HAVING	H—H—A—V—E—G—H
IMPORT	H—I—M—P—O—R—T—H
IN	H—I—N—H
INDEX	H—I—N—D—E—X—H
INNER	H—I—N—N—E—R—H
INSERT	H—I—N—S—E—R—T—H
INTERSECT	H—I—N—T—E—R—S—E—C—T—H
INTO	H—I—N—T—O—H
IS	H—I—S—H
JOIN	H—J—O—I—N—H
KEY	H—K—E—Y—H
LAST	H—L—A—S—T—H
LEADING	H—L—E—A—D—I—N—G—H
LEFT	H—L—E—F—T—H
LIKE	H—L—I—K—E—H
LIMIT	H—L—I—M—I—T—H
MONOTONIC	H—M—O—N—O—T—O—N—I—C—H
NO	H—N—O—H
NODE	H—N—O—D—E—H
NOT	H—N—O—T—H

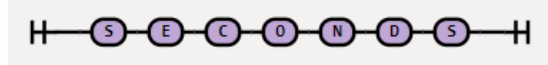
NULL	H N U L L H
NULLS	H N U L L S H
OFFSET	H O F F S E T H
ON	H O N H
OR	H O R H
ORDER	H O R D E R H
OUTER	H O U T E R H
OVER	H O V E R H
OVERLAY	H O V E R L A Y H
PARTITION	H P A R T I T I O N H
PLACING	H P L A C I N G H
PRIMARY	H P R I M A R Y H
REFERENCE	H R E F E R E N C E H
REFRESH	H R E F R E S H H
REPLACE	H R E P L A C E H
RIGHT	H R I G H T H
SCHEMA	H S C H E M A H
SELECT	H S E L E C T H
SET	H S E T H
STATS	H S T A T S H
STATISTICS	H S T A T I S T I C S H
TABLE	H T A B L E H
TO	H T O H

TOP	H O O P H
TRAILING	H T R A I L I N G H
TRUNCATE	H T R U N C A T E H
TRUE	H T R U E H
TYPE	H T Y P E H
UNION	H U N I O N H
UNIQUE	H U N I Q U E H
UPDATE	H U P D A T E H
USING	H U S I N G H
VALUES	H V A L U E S H
WHERE	H W H E R E H
WITH	H W I T H H
BOOLEAN	H B O O L E A N H
INTEGER	H I N T E G E R H
DECIMAL	H D E C I M A L H
FLOAT	H F L O A T H
TIMESTAMP	H T I M E S T A M P H
INTERVAL	H I N T E R V A L H
CHAR	H C H A R H
CHARACTER	H C H A R A C T E R H
VARCHAR	H V A R C H A R H
TEXT	H T E X T H

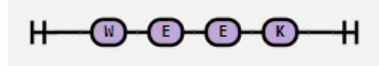
BINARY	H—B—I—N—A—R—Y—H
VARBINARY	H—V—A—R—B—I—N—A—R—Y—H
BLOB	H—B—L—O—B—H
VARYING	H—V—A—R—Y—I—N—G—H
BIT	H—B—I—T—H
BITS	H—B—I—T—S—H
BYTE	H—B—Y—T—E—H
BYTES	H—B—Y—T—E—S—H
CHARS	H—C—H—A—R—S—H
DIGITS	H—D—I—G—I—T—S—H
CASE	H—C—A—S—E—H
CAST	H—C—A—S—T—H
COALESCE	H—C—O—A—L—E—S—C—E—H
CONCAT	H—C—O—N—C—A—T—H
COUNT	H—C—O—U—N—T—H
DECODE	H—D—E—C—O—D—E—H
EXTRACT	H—E—X—T—R—A—C—T—H
EXISTS	H—E—X—I—S—T—S—H
IF	H—I—F—H
IIF	H—I—F—H

IFNULL	H—U—F—N—U—L—L—H
NULLIF	H—N—U—L—L—E—F—H
POSITION	H—P—O—S—I—T—I—O—N—H
SUBSTR	H—S—U—B—S—T—R—H
SUBSTRING	H—S—U—B—S—T—R—I—N—G—H
TRIM	H—T—R—I—M—H
TYPEOF	H—T—Y—P—E—O—F—H
WHEN	H—W—H—E—N—H
THEN	H—T—H—E—N—H
ELSE	H—E—L—S—E—H
END	H—E—N—D—H
DAY	H—D—A—Y—H
DAYS	H—D—A—Y—S—H
HOUR	H—H—O—U—R—H
HOURS	H—H—O—U—R—S—H
MINUTE	H—M—I—N—U—T—E—H
MINUTES	H—M—I—N—U—T—E—S—H
MONTH	H—M—O—N—T—H—H
MONTHS	H—M—O—N—T—H—S—H
QUARTER	H—Q—U—A—R—T—E—R—H
QUARTERS	H—Q—U—A—R—T—E—R—S—H

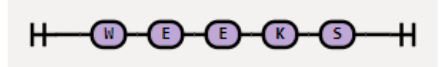
SECONDS



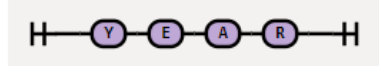
WEEK



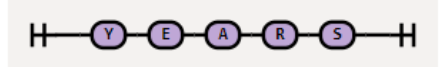
WEEKS



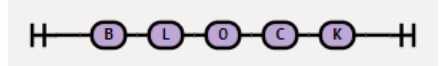
YEAR



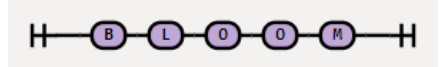
YEARS



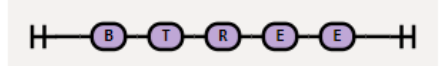
BLOCK



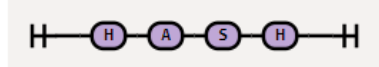
BLOOM



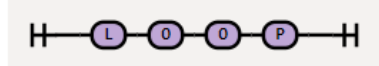
BTREE



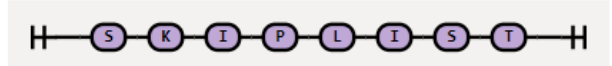
HASH



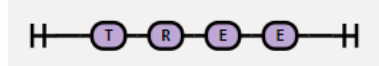
LOOP



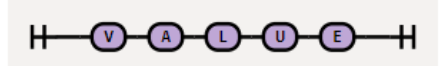
SKIPLIST



TREE



VALUE



DUBLSINGL	
DUBLDUBL	
IDENTIFIER	
PLAIN_IDENTIFIER	
BAKTIK_IDENTIFIER	
DUBLQUOTED_IDENTIFIER	
STRING_LITERAL	
SINGQUOTED_STRING	
DUBLQUOTED_STRING	
FLOAT_LITERAL	
DECIMAL_LITERAL	
INTEGER_LITERAL	
SLASH_COMMENT	
SQL_COMMENT	
BLOCK_COMMENT	