Discover what Columnstore Can Really Do for You

Roman Nozdrin

@MariaDB Day Brussels Feb 2025

MariaDB

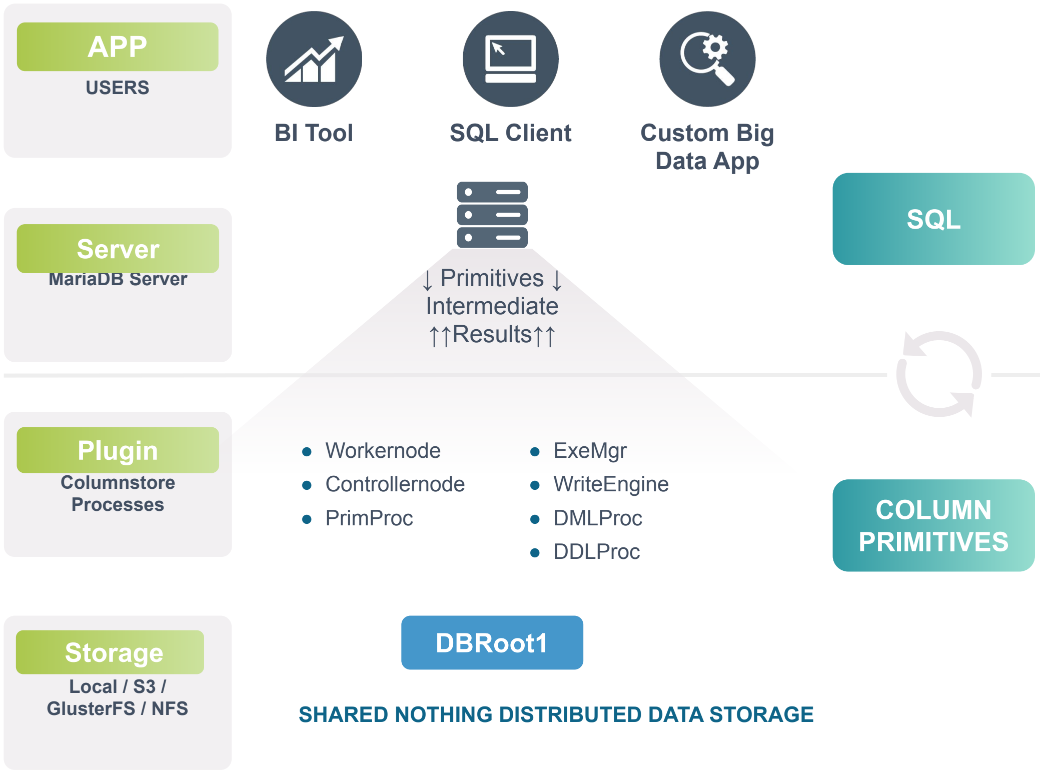
Agenda

- Columnstore overview
- Columnstore use cases
  - Features that enables the use cases

COLUMNSTORE OVERVIEW

MariaDB

MASSIVELY PARALLEL, SHARED NOTHING ARCHITECTURE



APP
USERS

BI Tool

SQL Client

Custom Big Data App

Server
MariaDB Server

↓ Primitives ↓
Intermediate
↑↑Results↑↑

Plugin
Columnstore Processes

- Workernode
- Controllernode
- PrimProc

- ExeMgr
- WriteEngine
- DMLProc
- DDLProc

Storage
Local / S3 / GlusterFS / NFS

DBRoot1

SHARED NOTHING DISTRIBUTED DATA STORAGE

SQL

- Query received and parsed by MariaDB frontend
- Storage Engine Plugin breaks down query in primitive operations and distributes across all nodes

COLUMN PRIMITIVES

- Primitives processed on each node
- One thread working on a range of rows
- Execute column restrictions and projections
- Execute group by/aggregation against local data
- Each Node work on Primitives in parallel and fully distributed
- Each primitive executes in a fraction of a second
- Return intermediate results to MariaDB

MariaDB

COLUMNSTORE USE CASE 1

Replacing closed-source OLAP with MariaDB Columnstore

Background

- Looking to save money & retain more data while maintaining performance compared to the closed-source OLAP

Challenge

- 10TB to 20TB databases
- 3TB+ raw uncompressed daily imports
- On premise closed networks

Features used

- Fast versatile data importing
- Partitioning for tables data
- Distributed report queries execution

DATA IMPORTING

MariaDB

CPIMPORT

Fastest way to ingest data directly into storage; bypasses SQL interface

With `cpimport` data is loaded without impacting the querying capability of the cluster and is available after the data load process is completed

Prerequisite: the table needs to be created beforehand

Example loading data from data file using `cpimport`

```
# cpimport -s ',' -E '"' test table1 table1.csv
```

Example loading data from another application using `cpimport`

```
# zcat t1.csv.gz | cpimport -s ',' -E '"' test t1
```

Example loading data from standard input and `mariadb` client using `cpimport`

```
# mariadb -q -e 'SELECT * FROM table1' -N db2 |
   /usr/bin/cpimport \
  -j501 -s '\t' -f STDIN
```

MODE 3 IMPORTING

Expects files to be prepared for each node and they will be injected as-is

Fastest mode, but more complex

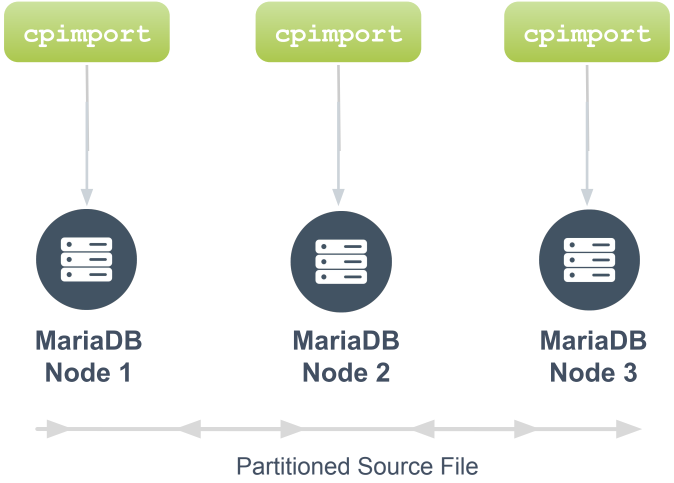
Parallel Distributed Load

Loaded from Each node separately and only

Concurrent Loads can be Executed on Multiple Nodes for the same table

Used to manually load data to a specific node or to all nodes

```
# cpimport -m3 db1 table1 -l /path/table1.tbl
```



REMOTE IMPORTING

`LOAD DATA LOCAL INFILE` can be run from a remote (non-database) machine

`LOAD DATA LOCAL INFILE` needs a user with proper credentials to access the remote database and the `FILE` privilege to execute `LOAD DATA`

`LOAD DATA LOCAL INFILE` has its own enable/disable flag in the MariaDB Server configuration.

Even if `LOAD DATA LOCAL INFILE` is wrapped in a transaction there is a way to ensure that `cpimport` is invoked by setting `columnstore_use_import_for_batchinsert` [`ON|OFF|ALWAYS`]

BULK LOADING FROM S3

Load data directly from S3

- Data is natively read from an S3 bucket by `cpimport`

```
# cpimport test sms -s ","cpimport test sms sms_bulk.csv -s "," -y $S3_ACCESS_KEY_ID
-K $S3_SECRECT_ACCESS_KEY -t mdb01
```

- Or data is read from an S3 bucket with AWS CLI and the output is piped into `cpimport`
- The AWS CLI tool must be installed and configured on the host

```
# aws s3 cp --quiet s3://mdb01/sms_bulk.csv - | cpimport test sms -s ","
```

BULK LOADING FROM S3

Load data directly from S3

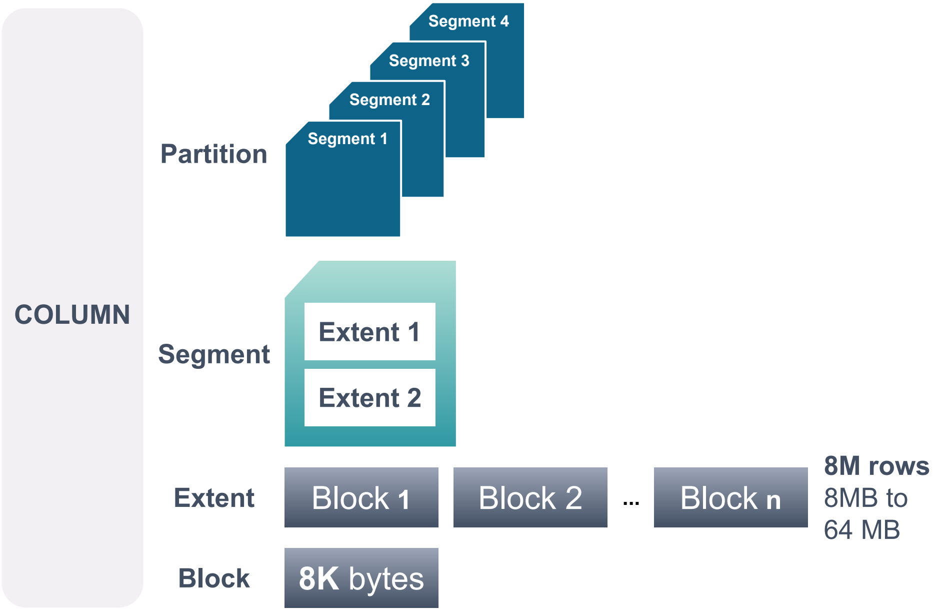
- Data is natively read from an S3 bucket by `UDF using CMAPI`

```
MariaDB [mytest]> CALL columnstore_info.load_from_s3("s3://dleeqadata", "1g/lineitem.tbl", "mytest",
"lineitem", "|", "", "" );
+------------------------------------------------------------------------------------------+
| columnstore_dataload(bucket, filename, dbname, table_name, terminated_by, enclosed_by, escaped_by) |
+------------------------------------------------------------------------------------------+
| {"success": true, "inserted": "6001215", "processed": "6001215"}                                  |
+------------------------------------------------------------------------------------------+
1 row in set (16.243 sec)
```

- See https://jira.mariadb.org/browse/MCOL-5013

PARTITIONING

PARTITION, SEGMENT, EXTENT AND BLOCKS



- Each column stored independently in 8M rows logical measure called an Extent
- An Extent is physically stored as collection of blocks
- A block is 8K Bytes
- String columns > 8 characters store indexes in the main column file and actual values in separate dictionary files
- Collectively, the column files and dictionary files for an extent form a Partition
- Partitions stored in a hierarchical structure organized by segments (i.e. folders)
- ExtentMap - meta store maps file structure/location to database schema as well as information used for partitioning
- By default the data is compressed

PARTITION MANAGEMENT

ColumnStore horizontally partitions extents per 8 million rows

Minimum and maximum values for each extent form a partition schema if data is loaded in semi-order

Partitions can be displayed for a table and column

Partitions can be disabled, enabled, or purged to remove rows corresponding to matched extents

Disabled values are hidden, not deleted

Operations can be performed by extent map minimum, maximum values or by extent id

DISPLAYING PARTITION INFORMATION

Display partitions by a given table and column

```
select calShowPartitions('orders','orderdate');
+----------------------------------------------------------------+
|calShowPartitions('orders','orderdate')                         |
+----------------------------------------------------------------+
| Part# Min        Max       Status
0.0.1 1992-01-01 1998-08-02 Enabled
0.1.2 1998-08-03 2004-05-15 Enabled
0.2.3 2004-05-16 2010-07-24 Enabled |
+----------------------------------------------------------------+
1 row in set (0.05 sec)
```

LEVERAGING PARTITIONS WITH SQL FUNCTIONS

- **idbPartition(column)** -the three part partition id (Directory.Segment.DBRoot)
- **idbPm(column)** -the PM where the physical row resides
- **idbSegmentDir(column)** - the lowest level directory id for the column file containing the physical row
- **idbSegment(column)** - he number of the segment file containing the physical row
- **idbLocalPm()** The PM from which the query was launched. This function will return NULL if the query is launched from a standalone UM

```
select * from 'orders' where idbPartition(orderdata) = '0.2.3';
```

Full list at **https://mariadb.com/kb/en/columnstore-information-functions/**

COLUMNSTORE USE CASE 2

MariaDB

RESEARCH WORKLOAD

Background

- A customer used to run OLAP queries using OLTP engine that took 90 days

Challenge

- Run SQL on 20TB tables reducing 90 to less than 8 hours
- Fast data migration from the existing storage

Features used

- Disk-based SQL operations
- Fast versatile data importing
- Distributed queries execution

DISK-BASED SQL OPERATIONS

DISK-BASED GROUP BY AND JOIN CONFIGURATION

- Enable features with commands

```
sudo mcsSetConfig HashJoin AllowDiskBasedJoin Y
sudo mcsSetConfig RowAggregation AllowDiskBasedAggregation Y
sudo mcsSetConfig SystemConfig SystemTempFileDir $PATH
```

- Optionally set a path for temporary files

```
sudo mcsSetConfig SystemConfig SystemTempFileDir $PATH
```

- Or set the values in /etc/columnstore/Columnstore.xml directly

COLUMNSTORE USE CASE 3

WEB MARKETING SOLUTION

Background

- Online marketing solution based on manually sharded MariaDB cluster

Challenge

- Run analytics SQL preserving their current application patterns with enormous INSERT rate to avoid using ETL from OLTP engine to OLAP

Features used

- INSERT Cache
- Fast DELETE
- Distributed queries execution

INSERT CACHE

INSERT Cache

- Enable in MariaDB server config for columnstore(Ubuntu 24.04)

```
# sudo echo “columnstore-cache-inserts=ON” >> /etc/my.cnf.d/columnstore.cnf
# sudo systemctl restart mariadb
```

- Works for tables created when the feature is active
- 600 record singleton import test (Innodb 2.2s to 2.7s = ~245 TPS)

```
# LocalStorage w/ Cache Inserts - 1.75x to 3x slower
Start:               17:41:39.456573208   0
InnoDB Done:         17:41:41.752143571   2.291949568
Columnstore Done:    17:41:46.314279229   6.854710546

# LocalStorage without Cache Insert - 35x slower
Start:               17:53:13.739659922   0
InnoDB Done:         17:53:16.293950582   2.548612200
Columnstore Done:    17:54:42.321429012   88.578447000
```

FAST DELETE

Fast DELETE

- Enable in Columnstore.xml

```
# sudo mcsSetConfig WriteEngine FastDelete y
# systemctl restart mariadb-columnstore / mcs cluster restart
```

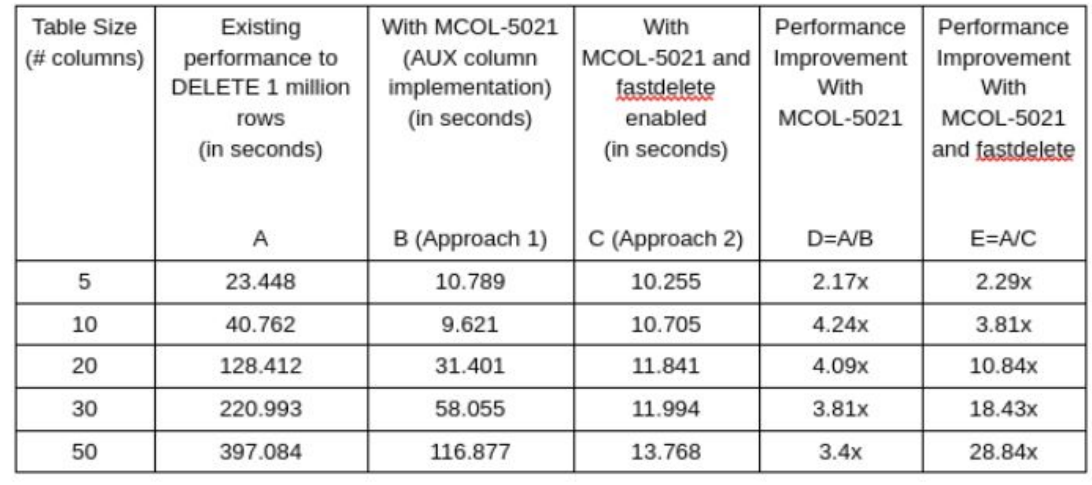


Table Size (# columns)	Existing performance to DELETE 1 million rows (in seconds) A	With MCOL-5021 (AUX column implementation) (in seconds) B (Approach 1)	With MCOL-5021 and fastdelete enabled (in seconds) C (Approach 2)	Performance Improvement With MCOL-5021 D=A/B	Performance Improvement With MCOL-5021 and fastdelete E=A/C
5	23.448	10.789	10.255	2.17x	2.29x
10	40.762	9.621	10.705	4.24x	3.81x
20	128.412	31.401	11.841	4.09x	10.84x
30	220.993	58.055	11.994	3.81x	18.43x
50	397.084	116.877	13.768	3.4x	28.84x

CROSS ENGINE JOIN

CROSS ENGINE JOINS

Cross Engine Joins allow ColumnStore to access and query non-ColumnStore tables in MariaDB Server

Implemented in the ColumnStore engine rather than MariaDB server

Row data can also be updated from columnar using a cross-engine JOIN

Need to correctly set up cross engine join user. This was discussed in ColumnStore Configuration lesson

Common Use Case

Manage dimension tables as InnoDB, and fact tables as ColumnStore

CROSS ENGINE JOIN CONFIGURATION

```
sudo mcsSetConfig CrossEngineSupport Host mcs1
sudo mcsSetConfig CrossEngineSupport Port 3306
sudo mcsSetConfig CrossEngineSupport User cross_engine
sudo mcsSetConfig CrossEngineSupport Password Cr0ss_eng!ne_passwd
```

The password may be encrypted with a key

Generate a key using `cskeys` command-line tool (all nodes should have the same key; it should only be readable to the ColumnStore system user)

Encrypt the password with the `cspasswd` utility before adding it to the configuration

CROSS ENGINE JOIN WHAT IF…

```
CREATE TABLE IF NOT EXISTS INNODB_TABLE ( a DECIMAL(12, 2), b int, INDEX idx_b_a (b, a)) ENGINE=innodb
PARTITION BY KEY(b,a) PARTITIONS 4;

INSERT INTO INNODB_TABLE SELECT ROUND(RAND() * 1000000, 2),ROUND(RAND() * 10000, 0) FROM
seq_1_to_32000000;

select b, sum(a) from INNODB_TABLE group by b;
13.562 sec

select b, sum(a) from
    SAME_MCS_TABLE where 0=1 group by b
 UNION ALL
    select b, sum(a) from INNODB_TABLE where b between 0 AND 2500 group by b  UNION ALL
    select b, sum(a) from INNODB_TABLE where b between 2501 AND 5000 group by b  UNION ALL
    select b, sum(a) from INNODB_TABLE where b between 5001 AND 7500 group by b  UNION ALL
    select b, sum(a) from INNODB_TABLE where b between 7501 AND 10000 group by b;
 11.120 sec
```

CROSS ENGINE JOIN WHAT IF…

```
select s_name, count(*) as numwait
from
(select * from mcs_schema.supplier, mcs_schema.lineitem l1, mcs_schema.orders, mcs_schema.nation
       where
           s_suppkey = l1.l_suppkey and o_orderkey = l1.l_orderkey and s_nationkey = n_nationkey
           and  0=1
   UNION ALL
       select * from innodb_schema.supplier, innodb_schema.lineitem l1, innodb_schema.orders, innodb_schema.nation
       where s_suppkey = l1.l_suppkey and o_orderkey = l1.l_orderkey and l1.l_receiptdate > l1.l_commitdate
           and exists(
               select * from innodb_schema.lineitem l2
               where l2.l_orderkey = l1.l_orderkey and l2.l_suppkey <> l1.l_suppkey
           )
           and not exists (
               select *  from innodb_schema.lineitem l3
               where l3.l_orderkey = l1.l_orderkey and l3.l_suppkey <> l1.l_suppkey and l3.l_receiptdate > l3.l_commitdate
           ) and s_nationkey = n_nationkey and n_name = 'SAUDI ARABIA'
) tmp group by s_name order by numwait desc, s_name limit 100;
```

MariaDB

Thank you