ColumnStore
OpenSource Engine for Analytics/BI

Bruno Šimić
Solutions Engineer
Agenda

• About MariaDB

• Data and big data

• Speed, security, performance & scalability, cost saving

• Architecture

• Differences to row-oriented storage

• Windows functions

• Best practices & use cases

• Q&A
“MariaDB was created to preserve openness and community, so that we can push ahead faster with the capabilities for tomorrow’s applications.”

Michael “Monty” Widenius
Founder & CTO of MariaDB

• MariaDB was created in 2009 by the same people behind MySQL
• MariaDB is the fastest growing Open Source RDBMS
• more than 12 million users worldwide trust critical business data to MariaDB
• the database of choice to power applications at companies like booking.com, HP, Virgin Mobile and Wikipedia
• secure, highly reliable and extensible
• trusted by the world’s leading brands
• MariaDB has been chosen by the leading Linux distributions as their default open source database
Data and big data

OLTP vs. OLAP
OLTP

On-line Transaction Processing
- large number of short on-line transactions (INSERT, UPDATE, DELETE)
- very fast query processing, maintaining data integrity in multi-access environments
- effectiveness measured by number of transactions per second
- operational (detailed and current) data
- OLTPs are the original data source

OLAP

On-line Analytical Processing
- characterized by low volume of concurrent transactions
- complex queries, often involving aggregations
- response time is effectiveness measure
- data is aggregated, historical and stored in multi-dimensional schemas
- OLAP data comes from the various OLTP Databases

Rows/Data Size Scope

<table>
<thead>
<tr>
<th>Rows/Data Size Scope</th>
<th>1</th>
<th>100</th>
<th>10,000</th>
<th>1,000,000</th>
<th>100,000,000</th>
<th>10,000,000,000</th>
<th>100,000,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10-100GB</td>
<td>100-1000GB</td>
<td>1-10TB</td>
<td>10-100TB</td>
<td>...PB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MariaDB OLTP

MariaDB ColumnStore OLAP
1. Descriptive Analytics
   What is Happening?
2. Diagnostic Analytics
   Why did it Happen?
3. Predictive Analytics
   What is likely to happen?
4. Prescriptive Analytics
   What should I do about it?
MariaDB ColumnStore

Fast, secure, performant, cost saving
MariaDB
ColumnStore enables real-time business intelligence and analytics

- a columnar storage built by forking InfiniDB
- expands MariaDB functionality to support scalable, real-time big data analytics
- built on the foundation of MariaDB Server
- provides a familiar and single point of access for both transactional and analytic processes
- utilizes a massively parallel distributed data architecture
- enables linear scalability and exceptional performance with real-time response to analytics queries
- Built in redundancy and high availability
- ACID compliant
- designed for big data scaling to process petabytes of data
- released under the GPL
MariaDB ColumnStore is a columnar storage engine for massively parallel distributed query execution and data loading. It supports a vast spectrum of analytic database use cases including real time, batch and algorithmic.

- Increase performance with complex aggregation, joins and windowing functions at the data storage level
- Optimize for real-time and ad hoc database analytics
- No indexes, no materialized views
- Integrate with Spark for advanced analytics
MariaDB ColumnStore accesses all the same security capabilities delivered in MariaDB Server. Secure ColumnStore data with encryption for data in motion, role-based access and audit features.

- Secure data in motion using out-of-the-box encryption
- Tighten security with fine-grained role-based access
- Easy auditability with a zero-downtime activation of query history
MariaDB ColumnStore leverages the I/O benefits of columnar storage, compression, just-in-time projection, and horizontal and vertical partitioning to deliver tremendous performance when analyzing large data sets.

- Massive parallel processing for query intensive environments by distributed scans, hash joins and aggregation
- Support for fully parallel load capabilities
- Faster time to insight with multi-threaded query execution and data streaming at 500K writes per second
- Runs as single or multi node system
- Linear scalable by adding new nodes as data grows
- no manual partitioning
ColumnStore is a pluggable storage engine accessible from the same SQL compliant interface as MariaDB Server. This radically simplifies data management while reducing operating costs.

- Save time writing queries by using MariaDB’s fully compatible SQL user interface
- Easily cross join tables from multiple storage engines for full insights
- Support wide variety of BI tools with ANSI SQL support and language drivers - any that uses ODBC/JDBC or MariaDB/MySQL connectors
MariaDB ColumnStore

Architecture
ColumnStore Setup

User Module: Processes SQL Requests

Performance Module: Multi Threaded
Distributed Query Engine

User Connections

Clients

User Modules

Performance Module 1

Performance Module 2

Performance Module 3

... Performance Module N

Columnar Distributed Data Storage

Local Disks, SAN, EBS, GlusterFS

MariaDB SQL Front End

Distributed Query Engine

 rajadb
**User Modules**

- mysql - The MariaDB server
- ExeMgr - MariaDB’s interface to ColumnStore
- cpimport - high-performance data import

**Query Processing - UM**

- SQL Operations are translated into thousands of Primitives
- Parallel/Distributed SQL
- Extensible with Parallel/Distributed UDFs
- Query is parsed by mysql on UM node
- Parsed query handed over to ExeMgr on UM node
- ExecMgr breaks down the query in primitive operations
Performance Modules

- PrimProc - Primitives Processor
- WriteEngineServ - Database file writing processor
- DMLProc - DML writes processor
- DDLProc - DDL processor

Query Processing - PM

- Primitives are processed on PM
- One thread working on a range of rows
- Typically 1/2 million rows, stored in a few hundred blocks of data
- Execute all column operations required (restriction and projection)
- Execute any group by/aggregation against local data
- Return results to ExeMgr process in User Module
- Each primitive executes in a fraction of a second
- Primitives are run in parallel and fully distributed
8 Steps of query execution (1..4)

1) A request comes in through the Front end interface. MariaDB performs a table operation for all tables needed to fulfill the request and obtains the initial query execution plan from MariaDB Server.

2) Storage engine interface converts the MariaDB table objects to MariaDB ColumnStore objects. These objects are then sent to a User Module.

3) The User Module converts the MariaDB execution plan and optimizes these objects into an MariaDB ColumnStore execution plan. The User Module determines the steps needed to run the query and when they can run.

4) The User Module consults the Extent Map for the locations of the data needed to satisfy the query and performs extent elimination based on the information contained within the Extent Map.
5) The User Module sends commands to one or more Performance Modules to perform block I/O operations.

6) The Performance Module(s) carry out predicate filtering, join processing, initial aggregation of data, and sends data back to the User Module for final result set processing.

7) The User Module performs final result set aggregation and composes the final result set for the query.

8) The User Module returns the result set back for delivery to the user.
Column-oriented storage

Differences to row-oriented storage
Data Storage

Logical Layer

Table

Column1

ColumnN

Extent 1
(8MB~64MB
8 million rows)

Extent N
(8MB~64MB
8 million rows)

Physical Layer

Server

DB Root

Segment
File1 (Extent)

Segment
FileN (Extent)

Blocks (8KB)

- Vertical Partitioning by Column
- Horizontal Partitioning by range of rows
# Row-oriented vs. Column-oriented format

<table>
<thead>
<tr>
<th>Key</th>
<th>Fname</th>
<th>Lname</th>
<th>State</th>
<th>Zip</th>
<th>Phone</th>
<th>Age</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bugs</td>
<td>Bunny</td>
<td>NY</td>
<td>11217</td>
<td>(718) 938-3235</td>
<td>34</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>Yosemite</td>
<td>Sam</td>
<td>CA</td>
<td>95389</td>
<td>(209) 375-6572</td>
<td>52</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>Daffy</td>
<td>Duck</td>
<td>NY</td>
<td>10013</td>
<td>(212) 227-1810</td>
<td>35</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>Elmer</td>
<td>Fudd</td>
<td>ME</td>
<td>04578</td>
<td>(207) 882-7323</td>
<td>43</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>Witch</td>
<td>Hazel</td>
<td>MA</td>
<td>01970</td>
<td>(978) 744-0991</td>
<td>57</td>
<td>F</td>
</tr>
</tbody>
</table>

**Row oriented:**
- rows stored sequentially in a file.

**Column oriented:**
- Each column is stored in a separate file. Each column for a given row is at the same offset.
### Single-Row Operations - Insert

<table>
<thead>
<tr>
<th>Key</th>
<th>Fname</th>
<th>Lname</th>
<th>State</th>
<th>Zip</th>
<th>Phone</th>
<th>Age</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bugs</td>
<td>Bunny</td>
<td>NY</td>
<td>11217</td>
<td>(718) 938-3235</td>
<td>34</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>Yosemite</td>
<td>Sam</td>
<td>CA</td>
<td>95389</td>
<td>(209) 375-6572</td>
<td>52</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>Daffy</td>
<td>Duck</td>
<td>NY</td>
<td>10013</td>
<td>(212) 227-1810</td>
<td>35</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>Elmer</td>
<td>Fudd</td>
<td>ME</td>
<td>04578</td>
<td>(207) 882-7323</td>
<td>43</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>Witch</td>
<td>Hazel</td>
<td>MA</td>
<td>01970</td>
<td>(978) 744-0991</td>
<td>57</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>Marvin</td>
<td>Martian</td>
<td>CA</td>
<td>91602</td>
<td>(818) 761-9964</td>
<td>26</td>
<td>M</td>
</tr>
</tbody>
</table>

**Row oriented:**
new rows appended to the end.

**Column oriented:**
new value added to each file.

Columnar insert not efficient for singleton insertions (OLTP). Batch loads touches row vs. column. Batch load on column-oriented is faster (compression, no indexes).
# Single-Row Operations - Delete

<table>
<thead>
<tr>
<th>Key</th>
<th>Fname</th>
<th>Lname</th>
<th>State</th>
<th>Zip</th>
<th>Phone</th>
<th>Age</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bugs</td>
<td>Bunny</td>
<td>NY</td>
<td>11217</td>
<td>(718) 938-3235</td>
<td>34</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>Yosemite</td>
<td>Sam</td>
<td>CA</td>
<td>95389</td>
<td>(209) 375-6572</td>
<td>52</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>Daffy</td>
<td>Duck</td>
<td>NY</td>
<td>10013</td>
<td>(212) 227-1810</td>
<td>35</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>Elmer</td>
<td>Fudd</td>
<td>ME</td>
<td>04578</td>
<td>(207) 882-7323</td>
<td>43</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>Witch</td>
<td>Hazel</td>
<td>MA</td>
<td>01970</td>
<td>(978) 744-0991</td>
<td>57</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>Marvin</td>
<td>Martian</td>
<td>CA</td>
<td>91602</td>
<td>(818) 761-9964</td>
<td>26</td>
<td>M</td>
</tr>
</tbody>
</table>

**Recommended Partition Drop** to allow dropping columns in bulk.
### Single-Row Operations - Update

<table>
<thead>
<tr>
<th>Key</th>
<th>Fname</th>
<th>Lname</th>
<th>State</th>
<th>Zip</th>
<th>Phone</th>
<th>Age</th>
<th>Sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bugs</td>
<td>Bunny</td>
<td>NY</td>
<td>11217</td>
<td>(718) 938-3235</td>
<td>34</td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>Yosemite</td>
<td>Sam</td>
<td>CA</td>
<td>95389</td>
<td>(209) 375-6572</td>
<td>52</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>Daffy</td>
<td>Duck</td>
<td>NY</td>
<td>10013</td>
<td>(212) 227-1810</td>
<td>35</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>Elmer</td>
<td>Fudd</td>
<td>ME</td>
<td>04578</td>
<td>(207) 882-7323</td>
<td>43</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>Witch</td>
<td>Hazel</td>
<td>MA</td>
<td>01970</td>
<td>(978) 744-0991</td>
<td>57</td>
<td>F</td>
</tr>
</tbody>
</table>

### Row oriented:
Update 100% of rows means change 100% of blocks on disk.

### Column oriented:
Just update the blocks needed to be updated.
Changing the table structure

<table>
<thead>
<tr>
<th>Key</th>
<th>Fname</th>
<th>Lname</th>
<th>State</th>
<th>Zip</th>
<th>Phone</th>
<th>Age</th>
<th>Sex</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bugs</td>
<td>Bunny</td>
<td>NY</td>
<td>11217</td>
<td>(718) 938-3235</td>
<td>34</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>Yosemite</td>
<td>Sam</td>
<td>CA</td>
<td>95389</td>
<td>(209) 375-6572</td>
<td>52</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>Daffy</td>
<td>Duck</td>
<td>NY</td>
<td>10013</td>
<td>(212) 227-1810</td>
<td>35</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>Elmer</td>
<td>Fudd</td>
<td>ME</td>
<td>04578</td>
<td>(207) 882-7323</td>
<td>43</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>Witch</td>
<td>Hazel</td>
<td>MA</td>
<td>01970</td>
<td>(978) 744-0991</td>
<td>57</td>
<td>F</td>
<td>N</td>
</tr>
</tbody>
</table>

Row oriented:
requires rebuilding of the whole table

Column oriented:
Create new file for the new column

Column-oriented is very flexible for adding columns, no need for a full rebuild required with it.
Analytics

Window functions
Window Functions

- **PARTITION BY:**
  One or more columns or Functions
  \[ \text{PARTITION BY server\_date, item\_id} \]
  \[ \text{PARTITION BY MONTH(server\_date)} \]
  Column does not need to be in projection list. If omitted, all the input rows are in one partition.

- **ORDER BY:**
  One or more columns or Functions. Column does not need to be in projection list.

- **Aggregate functions:**
  MAX, MIN, COUNT, SUM, AVG
  STD, STDDEV_SAMP, STDDEV_POP, VAR_SAMP, VAR_POP

- **Ranking**
  ROW_NUMBER, RANK, DENSE_RANK, PERCENT_RANK
  CUME_DIST, NTILE, PERCENTILE, PERCENTILE_CONT
  PERCENTILE_DISC, MEDIAN
Window Functions

- **ROWS [BETWEEN <start> and <end> | <end>]**
  - CURRENT ROW
  - UNBOUNDED PRECEDING
  - UNBOUNDED FOLLOWING
  - <Number of rows> PRECEDING
  - <Number of rows> FOLLOWING

- **RANGE [BETWEEN <start> and <end> | <end>]**
  - UNBOUNDED PRECEDING
  - UNBOUNDED FOLLOWING
  - <value1> PRECEDING
  - <value2> FOLLOWING

- **FIRST/LAST**
  - NTH_VALUE, FIRST_VALUE, LAST_VALUE

- **LEAD/LAG**
  - LAG, LEAD
Window Function
PARTITION BY Example

SELECT item_id, server_date, daily_revenue, AVG(revenue) OVER (PARTITION BY item_id ORDER BY server_date RANGE INTERVAL '1' DAY PRECEDING ) running_avg FROM web_item_sales
### Top N Visitors for each Month - RANK Example

#### SELECT visitor_id, total_time, visit_month, RANK() OVER (PARTITION BY visit_month ORDER BY t1.total_time desc) time_rank

<table>
<thead>
<tr>
<th>Month</th>
<th>visitor_id (visitor identifier)</th>
<th>Total_time</th>
<th>Time_rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>February</td>
<td>John_A_001</td>
<td>670</td>
<td>Rank in February</td>
</tr>
<tr>
<td>February</td>
<td>Tom_J_002</td>
<td>400</td>
<td>Rank in February</td>
</tr>
<tr>
<td>March</td>
<td>Jane_M_009</td>
<td>1000</td>
<td>Rank in March</td>
</tr>
<tr>
<td>March</td>
<td>John_A_001</td>
<td>800</td>
<td>Rank in March</td>
</tr>
<tr>
<td>March</td>
<td>Dan_K_001</td>
<td>200</td>
<td>Rank in March</td>
</tr>
</tbody>
</table>

- Top 1 : time_rank = 1
- Top 2 : time_rank <= 2
- Top N: time_rank <= N
MariaDB ColumnStore

Best practices
Best Practices

General

• Not suited for OLTP, needs big data to process fast (millions of records)
• Micro-batch load allows near real-time behaviour
• Infrequently used columns do not impact other queries
• Columnar suitable for sparse columns (nulls compress nicely)

Query Modeling

• Star-schema optimizations are generally a good idea
• Conservative data typing is important
  – fixed-length vs. dictionary boundary (8 bytes)
  – IP Address vs. IP Number
• Break down compound fields into individual fields
  – Trivializes searching for sub-fields
  – Can avoid dictionary overhead
  – Cost to re-assemble is generally small
Data Ingestion

Cpimport

- Fastest way to load data from CSV file, standard input, binary source file
- Multiple tables in can be loaded in parallel by launching multiple jobs
- Read queries continue without being blocked
- Successful cpimport is auto-committed
- In case of errors, entire load is rolled back

LOAD DATA INFILE

- Traditional way of importing data into any MariaDB storage engine table
- Up to 2 times slower than cpimport for large size imports
- Either success or error operation can be rolled back
High Availability

HA at UM node

• When one UM node goes down, another UM node takes over

HA at PM node

• SAN/AWS EBS - When a PM node goes down, the data volumes attached to the failed PM node gets attached to another PM
• Local Disks - If a PM node goes down, the data on its disks are not available, though queries continue on the remaining data set

HA at Data Storage

• AWS EBS (Elastic Block Store)
• GlusterFS - Multiple copy of data block across storage. If a disk on a PM node fails, another PM node will have access to the copy of the data
MariaDB ColumnStore

Customer use cases
**Business Challenge**

- An organization is generating large amount of operational data
- Multiple terabytes of historical data
- With growth in business and in operational data
  - Analytics query performance degrades
  - Impractical to do analytics

**MariaDB Solution**

- Put past data into MariaDB ColumnStore
- Add new nodes as data grows
- Perform analytics without performance degradation
- Linear Scalability with data growth
## ColumnStore use cases

<table>
<thead>
<tr>
<th>Industry</th>
<th>Category</th>
<th>Use case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaming</td>
<td>Behavior Analytics</td>
<td>Projecting and predicting user behavior based on past and current data</td>
</tr>
<tr>
<td>Advertising</td>
<td>Customer Analytics</td>
<td>Customer behavior data for market segmentation and predictive analytics</td>
</tr>
<tr>
<td>Web, E-commerce</td>
<td>Click Stream Analytics</td>
<td>Web activity analysis, software testing, market research with analytics on data about the clicks areas of web pages while web browsing</td>
</tr>
<tr>
<td>Marketing</td>
<td>Promotional Testing</td>
<td>Using marketing and campaign management data to identify the best criteria to be used for a particular marketing offer</td>
</tr>
<tr>
<td>Social Network</td>
<td>Network Analytics</td>
<td>Relationship analytics among network nodes</td>
</tr>
<tr>
<td>Financial</td>
<td>Fraud Analytics</td>
<td>Monitoring user financial transactions and identifying patterns of behaviour to predict and detect abnormal or fraudulent activity to prevent damage to user and institution</td>
</tr>
<tr>
<td>Aviation</td>
<td>Flight analytics</td>
<td>Proactively project parts replacement, maintenance and airplane retirement based on real-time and historically collected flight parameter data</td>
</tr>
</tbody>
</table>
MariaDB ColumnStore

Links and resources
Building ColumnStore from source

$ git clone https://github.com/mariadb-corporation/mariadb-columnstore-server
$ cd mariadb-columnstore-server
$ git checkout develop
$ git clone https://github.com/mariadb-corporation/mariadb-columnstore-engine
$ cd mariadb-columnstore-engine
$ git checkout develop
$ cmake .
$ make -j
$ sudo make install

$ cmake . -DCMAKE_INSTALL_PREFIX=/usr/local/mariadb/columnstore/mysql/
$ make -j
$ sudo make install
$ sudo /usr/local/mariadb/columnstore/bin/postConfigure
Select the type of System Server install [1=single, 2=multi] (2) > 1
Enter System Name (columnstore-1) >
Select the type of Data Storage [1=internal, 2=external] (1) >
Enter the list (Nx,Ny,Nz) or range (Nx-Nz) of DBRoot IDs assigned to module 'pm1' (1) >
MariaDB Columnstore SNMP-Trap process is enabled, would you like to disable it [y,n] (n) >

https://mariadb.com/kb/en/mariadb/columnstore-getting-started/ (Installation how-to)
Learn more about MariaDB ColumnStore

• Product Page

• MariaDB ColumnStore Download

• MariaDB Columnstore Knowledge Base

• MariaDB ColumnStore on GitHub:
  – [https://github.com/mariadb-corporation/mariadb-columnstore-server](https://github.com/mariadb-corporation/mariadb-columnstore-server) (Server)

Thank you

Bruno Šimić
bruno@mariadb.com