# Introduction to column stores

Justin Swanhart

Percona Live, April 2013


PERCONA LIVE

# INTRODUCTION

# Introduction



- Who am I?
- What do I do?
- Why am I here?

# A quick survey



**How many people have heard the term row store?**

# A quick survey



**How many people have heard the term column store?**

# A quick survey



**How many people have heard the term grocery store?**

**(just kidding)**

# ROW STORES

# What is a row store



- You've probably used one ☺
  - MyISAM
  - InnoDB
  - Memory
  - TokuDB
  - Etc

# What is a row store



- All MySQL storage engines are row stores
  - The storage engine API is *row oriented*
  - This means the API *always* stores and retrieves entire rows from disk

# Materialization



- Row stores live "in a material world"
  - A row is a collection of column *values (attributes)*
  - A row store stores all the values together on disk
    - That is, rows are *materialized on disk*

# Rows are stored in one file or segment on disk



All columns are stored together on disk.

| id | scientist | death_by | movie_name |
|---|---|---|---|
| 1 | Reinhardt | Maximillian | The Black Hole |
| 2 | Tyrell | Roy Batty | Blade Runner |
| 3 | Hammond | Dinosaur | Jurassic Park |
| 4 | Soong | Lore | Star Trek: TNG |
| 5 | Morbius | His mind | Forbidden Planet |
| 6 | Dyson | Skynet | Terminator 2: Judgment Day |

# Best case scenario



- Performs best when a small number of rows are accessed

| id | scientist | death_by | movie_name |
|---|---|---|---|
| 1 | Reinhardt | Maximillian | |
| 2 | Tyrell | Roy Batty | Blade Runner |
| 3 | Hammond | Dinosaur | Jurassic Park |
| 4 | Soong | Lore | Star Trek: TNG |
| 5 | Morbius | Robby | |
| 6 | Dyson | Skynet | Terminator 2: Judgment Day |

**select * from the_table where id = 6**

# BUT…



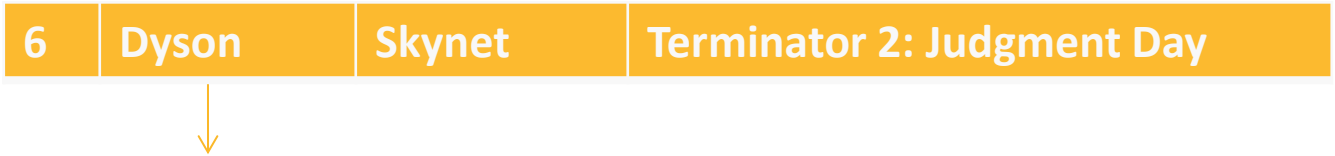
- Row stores not so great for wide rows
- It can be *very* bad if only a small subset of the columns are actually used by most queries
  - Reading the entire row wastes IO

# Row Stores can waste IO

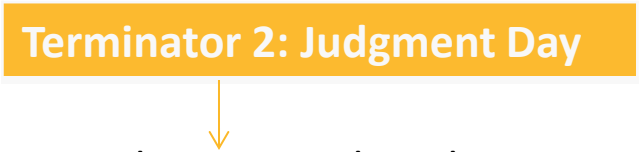


**select movie_name from the_table where id = 6**

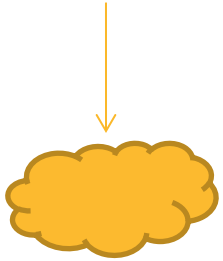
1. Storage engine *gets whole row**:




2. SQL interface extracts only requested portion




3. Results returned to client



* By primary key in this case

# Worst case scenario



- select sum(bigint_column) from table
  - 1000000 rows in table
  - Average row length 1024 bytes
- The select reads one bigint column (8 bytes)
  - But the entire row must be read to get one column
  - Reads **~1GB** of table data for **~8*MB*** of column data

# COLUMN STORES

# Non-material world



- In a column stores rows *are not materialized* during storage
  - An entire row image exists only *in memory*
  - Each row still has some sort of "row id"

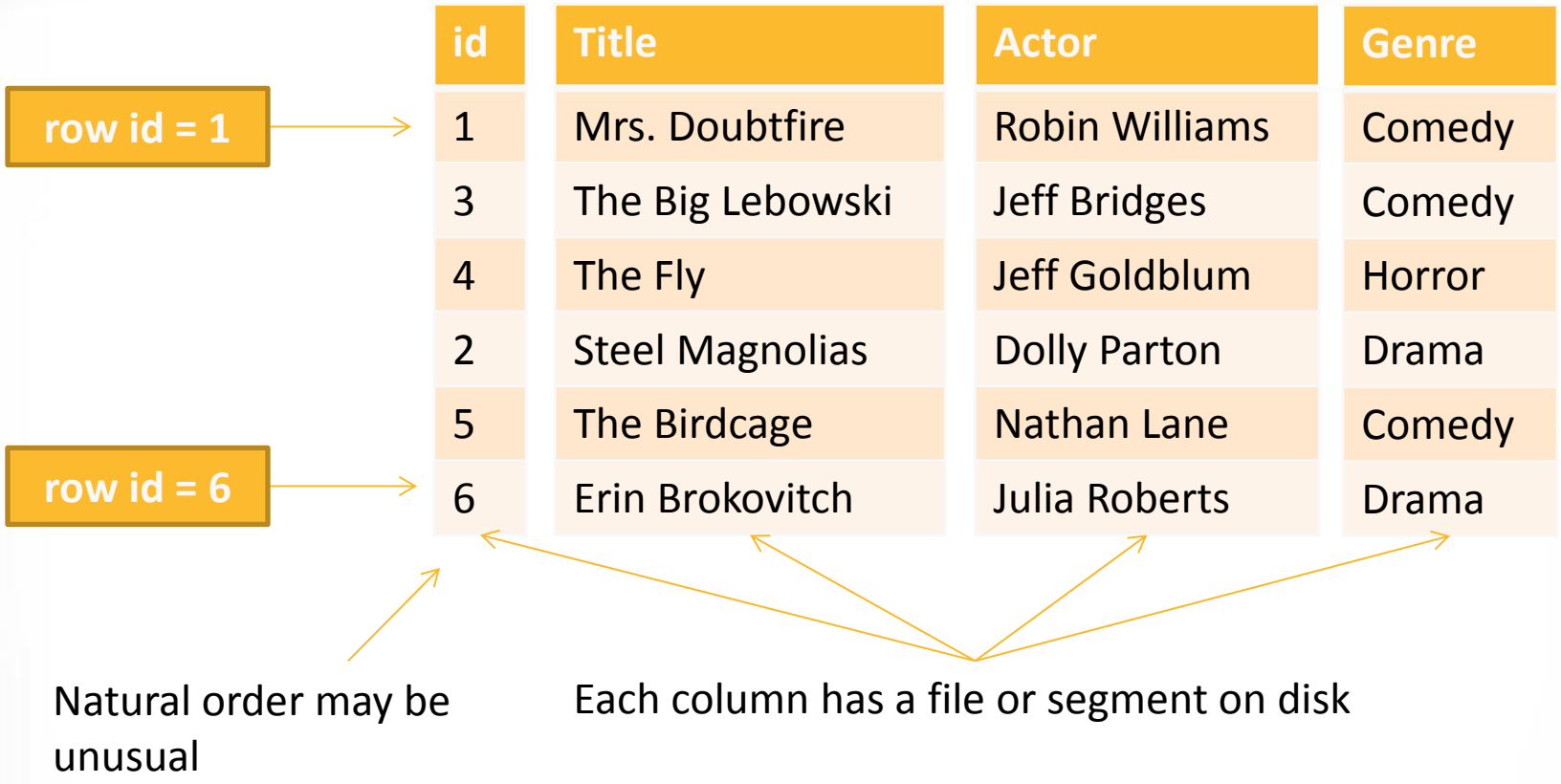
# What is a row?



- A row is a collection of column values *that are associated with one another*
- Associated?
  - Every row has some type of "row id"
    - PK column or GEN_CLUST_INDEX on InnoDB
    - Physical offset in a MyISAM table
    - In some databases it is just a virtual identifier for a row

# Column store stores each COLUMN on disk





# Column store benefit: Compression



- Almost all column stores perform compression
  - Compression further reduces the storage footprint of each column
  - Column data type tailored compression
    - RLE
    - Integer packing
    - Dictionary and lookup string compression
    - Other (depends on column store)

# Column store benefit: Compression



- Effective compression reduces storage cost
- IO reduction yields decreased response times during queries as well
  - Queries may execute an order of magnitude faster compared to queries over the same data set on a row store
- 10:1 to 30:1 compression rations may be seen

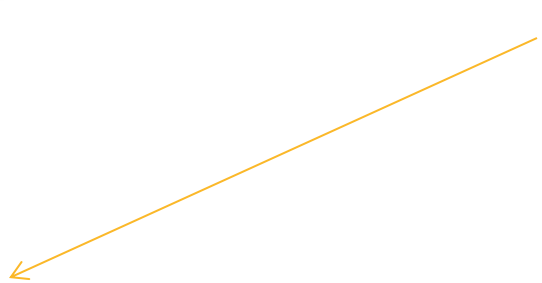
# Best case scenario (IO savings)



- select sum(bigint_column) from table
  - 1000000 rows in table
  - Average row length 1024 bytes
- The select reads one bigint column (8 bytes)
  - Only the single column is read from disk
  - Reads **~8MB** of column data ***instead of*** 1***GB*** of table data
  - Could read even less with compression!

# Less than ideal scenario*






```
select *
  from long_wide_table
 where order_line_id = 321837291;
```


* Possibly worst case.  We'll see in the performance test as this varies by column store.

# Column store downsides:



- Accessing all columns in a table doesn't save anything
  - It could even be more expensive than a row store
  - Not ideal for tables with few columns

# More downsides



- Updating and deleting rows is expensive
  - Some column stores are append only
  - Others strongly discourage writes
  - Some column stores split storage into row and column areas (in Vertica* this is called WOS/ROS)

*Not open source, based on C-Store an MIT project

# OLTP VS OLAP

# Analytics versus OLTP



- Row stores are for OLTP
  - Reading small portions of a table, but often many of the columns
  - Frequent changes to data
  - Small* amount of data (typically working set must fit in ram)
  - "Nested loops" joins are well optimized for OLTP

* < 2TB data sets in general

# Analytics versus OLTP



- Column stores are designed for OLAP
  - Read large portions of a table in terms of rows, but often a small number of columns
  - Batch loading / updates
  - Big data*!
    - Effective compression and reduced query IO make column stores much more attractive for structured big data
    - Machine generated data is especially well suited

* 50TB – 100TB per machine is typically possible, before compression

# Not all column stores are for big data!



- In-memory analytics is popular
  - Some column stores are designed to be used as an in-memory store
  - It is possible to build very large SMP clusters which scale these databases to many machines but this outside of the scope of this talk

# Indexing



- Row stores mainly use tree style indexes
  - When you create an index on a row store you usually create a "B" tree index
  - These indexes leverage binary search
  - Very fast as long as the index fits in memory

# Tree indexes



- A read is necessary to satisfy a write
  - Even with fast IO this is expensive
  - This means that tree indexes work best for in-memory data sets
  - Very large data sets usually end up with indexes which are unmanageably large

# Use less trees: Bitmap indexing



- Some column stores support bitmap indexes
  - Columnar storage lends very well to bitmaps
  - Bitmap indexes are effective for Boolean searches over multiple columns in a table

# Use less trees: Bitmap indexing



- Unfortunately
  - Bitmap indexes are not supported by any MySQL column store
  - Very expensive to update

# OPEN SOURCE COLUMN STORES

# Fastbit bitmap store



- Academic column store
  - Simple text interface
  - Allows bitmap indexes to be compared *whilst still compressed*
  - No joins
  - Append-only* data store

* records can be marked as deleted with library calls, but not removed from disk
https://sdm.lbl.gov/fastbit/

# MySQL column stores - ICE



- Infobright Community Edition (ICE)
  - Some data type limitations
  - Single threaded loader
  - Single threaded query
  - No temp tables, no RENAME table, no DDL
  - LOAD DATA only (append only, no DML)

# MySQL column stores – ICE [2]



- Data stored in "data packs"
- No indexes
  - Knowledge grid stores statistics and information about the data packs
    - Decides which packs must be decompressed
    - Can make very intelligent query decisions
- Hash join only
- No constraints

# MySQL column stores – InfiniDB Community



- InfiniDB community edition
  - Some data type limitations
  - Single threaded query*
  - Single threaded loader*
  - No indexes
  - Wrong results in testing
  - Hash join only

* You are required to bind infinidb to only a single core

# LucidDB



- Java based column store database
- SQL/MED data wrappers
- Built-in data versioning
- Commercial backing ended in October, 2012
  - But source code still available under Apache license

# LucidDB [2]



- Bitmap and tree based indexes
  - Support for foreign key, primary key and unique constraints
- ANSI SQL support
- Multiple join algorithms
- Single-threaded queries

# MonetDB



- Early (started circa 2004) column store
- Open source
- Designed for in-memory working sets
  - Indexes must fit in memory
  - Swap size must be large enough for working set if working set exceeds memory
- Some wrong results during testing

# MonetDB [2]



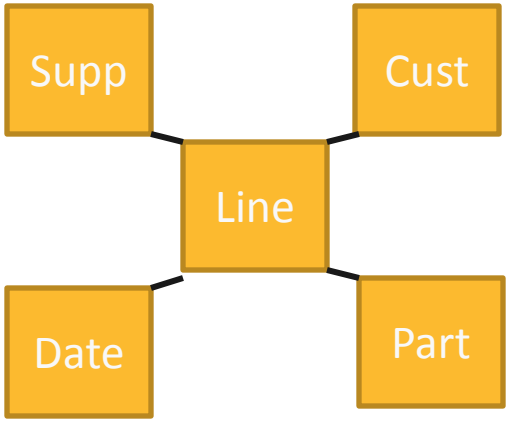
- Data is stored compressed on disk
- In memory structure is array-like
  - Parallel query execution
  - CPU cache optimizations
  - multiple join algorithms
- Primary key, foreign key, unique and NOT NULL constraints

# PERFORMANCE COMPARISON

# Loading Performance (InfiniDB omitted)



| | CUST | DATE | PART | SUPP | LINE | ALL TABLES |
|---|---|---|---|---|---|---|
| MonetDB | 0.83 | 0.26 | 1.980 | 0.23 | 125 | 128s |
| ICE | 1.69 | 0.09 | 2.372 | 0.15 | 128 | 133s |
| LucidDB | 2.49 | 0.31 | 5.915 | 0.96 | 297 | 307s |
| InnoDB | 3.11 | 0.14 | 11.83 | 0.53 | 300 | 316s |




Star Schema Benchmark
Scale Factor = 5
Lineorder rows = 29,999,795

# Point Lookup Query



```
select *
  from lineorder
 where LO_OrderKey = 10000001
   and LO_LineNumber = 3;
```

| | cold | hot |
|---|---|---|
| MonetDB | 0.081 | 0.001 |
| ICE | 0.29 | 0.07 |
| LucidDB | 2.4 | 0.9 |
| InnoDB | 0.01 | 0.001 |

# Simple Aggregate Query



```
select sum(LO_OrdTotalPrice) from lineorder;
```

| | cold | hot |
|---|---|---|
| ICE | 0.02 | 0.001 |
| MonetDB | 0.2 | 0.07 |
| LucidDB | 5.9 | 3.9 |
| InnoDB | 34 | 9.3 |

# Complex Query with no join



```
select LO_CustKey, sum(LO_OrdTotalPrice)
  from lineorder
 group by LO_CustKey
having sum(LO_OrdTotalPrice) >= 11349901235
 order by sum(LO_OrdTotalPrice) desc;
```

| | cold | hot |
|---|---|---|
| MonetDB | 3.9 | 1.5 |
| ICE | 7.19 | 3.14 |
| LucidDB | 17.5 | 13.3 |
| InnoDB | 41.18 | 33.81 |

# Complex Query with join, group by and filter



```
select d_year, p_brand, sum(lo_revenue)
    from lineorder
    join dim_date on lo_orderdate = d_datekey
    join part on lo_partkey = p_partkey
    join supplier on lo_suppkey = s_suppkey
   where p_category = 'MFGR#12'
     and s_region = 'AMERICA'
   group by d_year, p_brand
   order by d_year, p_brand;
```

| | cold | hot |
|---|---|---|
| ICE | 6.9 | 1.83 |
| LucidDB | 17.3 | 15.3 |
| InnoDB | 58.23 | 51.62 |
| MonetDB | ERR | ERR |



Wrong results

# CLOSED SOURCE COLUMN STORES

# Infobright Enterprise Edition – IEE



- Supports most MySQL SQL
- Parallel loader / binary loader
- Supports DML (insert, update, delete)
- Supports DDL (rename, truncate)
- Supports temp tables
- Parallel query
- Support

# Vectorwise



- In memory SMP column store
- Vector execution architecture for parallel pipelined MPP query execution
- Very fast
- Uses ScaleMP to scale to more than one machine

# Vertica



- An HP company
- Originally C-Store
- Write and read optimized storage
- “Projections” for pre-joining and sorting data
- Automatic consistent hashing data distribution(K-safety level)
- Massively parallel processing
- Partitioning support

# Links!



- MonetDB - http://dev.monetdb.org/downloads/sources/Latest/
- ICE - http://www.infobright.org
- InfiniDB - http://www.calpont.com
- LucidDB – http://luciddb.sourceforge.net
- Fastbit - https://sdm.lbl.gov/fastbit/
- Vectorwise – http://www.vectorwise.com
- Vertica – http://www.vertica.com

# Percona Training Advantage



- Check out http://training.percona.com for a list of training events near you
- Request training directly by Justin (me) or any of our other expert trainers by contacting your Percona sales rep today

# Q/A