Presenters

Alexander Bohm - SAP AG.
Niloy Mukherjee - LinkedIn Corporation
Rajkumar Sen - Oracle Corporation

Alfons Kemper - TU Munich (Guest Speaker)

Jens Dittrich - U. of Saarland
Ippokratis Pandis - Amazon Web Services
A Brief History of Analytics
Analytics 1.0: Till mid-2000s

• Long era of Decision Support Systems
• Isolated data warehouses
  • Curating data from Online Transaction Processing (OLTP) sources
  • Applying business intelligence on data at rest
• Data sets small in volume, static in velocity
• Standalone descriptive analytics and reporting
  • Not ad-hoc
  • Not real-time
  • Long running complex queries using pre-determined set of auxiliary structures
Analytics 1.0: Till mid-2000s

- Nightly Data Extraction, Transfer, and Loading Between Systems (ETL)
  - Transformation from OLTP-friendly to OLAP-friendly format; indexing; pre-aggregation
Analytics 2.0: in Relational Space

• Era of scale
  • Data volumes start increasing
  • Storage scales in capacity, memory becomes affordable
  • Processors scale in number of cores

• Real-time data processing over large volumes of data
  • Ad-hoc query processing

• Resurgence of columnar RDBMS
  • Compression-friendly format to handle large data volumes
  • Compute-friendly format for fast ad-hoc query processing
Analytics 2.0: in Relational Space

- Emergence of parallel processing DW systems
  - Multi high-end server systems with hundreds of cores and terabytes of storage
  - Integrated with specialized software for fast ad-hoc analytics mainly on data at rest
Analytics 2.0: Unstructured data deluge

• Explosion in unstructured data volumes
  • In petabytes scale

• Dramatic rise of ubiquitous applications and data sources
  • Internet, social, and mobile
  • Google, Facebook, LinkedIn, Twitter

• Commoditization of hardware and software
  • Massively distributed data stores built of commodity hardware
Analytics 2.0: Unstructured data deluge

• Introduces a co-existing analytics ecosystem
  • Hadoop / MapReduce framework for batch analytics on data at rest
  • Designed for scale in throughput and capacity
  • On-going efforts on SQL standardization, real-time ad-hoc analytics

• Introduces a co-existing online platform
  • NoSQL distributed key-value stores
  • DynamoDB, Cassandra, etc.
  • Low latency high throughput key-value style transactions

• Data Extraction, Transfer, and Loading Between online and offline Systems (ETL)
Common Pain Points of both Ecosystems

• Costs and manageability
  • Two (different?) data management systems
  • Two times licensing cost
  • Two times the hardware
  • Two times the administration

• No Data Freshness for Reporting
  • Data is curated infrequently
  • Reporting during the day always sees stale data
Common Pain Points of both Ecosystems

• ETL Processes are a Pain
  • Complex to design and maintain
  • Need to complete in daily maintenance window (e.g. <8 hours)
  • Data preparation necessary for reporting
  • Slow reaction to changing business demands

• As businesses becomes more agile
  • Lose significant competitive edge unless business decisions made on online data
Need of the Hour: Analytics 3.0

- **Batch**
  - ad hoc queries
  - monthly active users
  - relevance for ads

- **Online Non-Transactional**
  - ad impressions count
  - hash tag trends

- **Online Transactional**
  - deterministic workflows
  - fast out Tweets
  - search for Tweets

- **< 500 ms**
  - latency sensitive
Use Case 1: Retail

- Inventory, Customers, Products, Pricing and Purchases live in OLTP store
- Customer behavior, Clickstream data go to NoSQL, HDFS storage

Operational Analytics use cases:
- Optimized pricing, discounts
- Personalized recommendations to preferred customers
- Identify high-demand items and do pro-active inventory refill
Use Case 2: Content Delivery Networks (CDN)

- Global distributed network of web-servers to distribute content
  - e.g. Comcast CDN, Akamai CDN
  - Customer usage data logged into operational store for billing etc.
  - Business need to monitor system in real-time

Operational Analytics use cases:
- Denial-of-Service attack alerts
- Find locations with spikes to re-allocate bandwidth
- Find top-k customers in terms of increase of traffic
  - *Throttle some of them!!*
Use Case 3: Financial Services

- Real-time activity detection and action can save billions

- Regulatory Analytics
  - Identify money-laundering cases in real-time
  - Save fines worth billions of dollars

- Risk Analytics
  - Identify suspicious transactions in real-time
  - Automated fraud prevention
Use Case 4: Energy and Utilities

Smart Grid initiatives demand real-time operational analytics

• Operations
  • Optimizing power plant production
  • Detect transmission leaks in real-time
  • Predictive maintenance (using drones to monitor transmission assets !!)

• Business
  • Real-time dashboards on smart meter readings
  • Real-time energy trading in wholesale markets
Status-Quo

• Last few years have witnessed the evolution of real-time operational analytics database management systems
  • breakthrough performance for analytic workloads not only in pure analytic but also in mixed (analytic + online) environments

• Mostly, in relational space...

• Cloud-scale deployments becoming more commonplace
Agenda

• Technology Trends towards Operational Analytics DBMS

• Technical Deep-dive
  • SAP HANA
  • Oracle Database In-memory
  • MemSQL
  • Cloudera Impala with Kudu

• Academic Perspective

• Open Challenges
An operational analytics system

- A fully-equipped operational store
  - Fast SQL/NoSQL ACID transactions
  - Source of truth system.
- Runs analytical queries in real-time over the fresh transactional data
  - Analytic-friendly storage format
  - Support for complex SQL
  - Smart optimizer, fast execution
- Is scalable, available, and fault tolerant
- Exploits modern hardware (advanced processors, main-memory, SSD)
Trends in Storage
Pure Columnar Format

- **Analytics** run faster on column format
  - Example: Report on sales totals by region
  - Fast accessing few columns, many rows

- Columnar organization provides better compression
  - Maximal capacity utilization on large data volumes

- Columnar organization main-memory/SSD optimized
  - For example, vector processing (SIMD)
Pure Columnar Format

- Geared towards traditional data warehousing and pure OLAP practices
  - Data needs to be first curated in usually dedicated data warehouses

- Not really suited for real-time analytics in mainstream production environments
  - Data arrives in production environments through OLTP
  - Columnar format not suitable for traditional OLTP
Operational Columnar Format

• Read-optimized immutable pure columnar storage snapshot
  • Main-memory/SSD optimized

• Write-optimized journaling of operational changes
  • Delta stores in columnar format
  • Traditional row stores

• Periodical merge of deltas into base snapshots
  • LSM
  • MVCC
Immediate Consistency

- Analytic workloads deliver consistent results in real-time
  - No near real-time/stale consistency invariants

- Analytic query processing consults columnar snapshots and delta journals to construct consistent set of results

- Journaling and merge mechanisms very ‘significant’
  - To handle operation workload
  - To provide high analytics performance on changing data
Main-memory Column Stores

DRAM getting cheaper over the years

Exponential increase in capacity:price

Cheaper to place more and more data closer to compute
Main-memory/SSD Optimized Row Stores

- Main-memory optimized column-store provides fast analytics

- Main-memory/SSD optimized row/delta stores provide extremely low latency and high throughput for operational workloads
  - Lock-free skip lists
    - Main-memory based structures
  - Latch-free BW trees
    - Append of delta updates
    - Delta merge
    - Employs CAS
Trends in Compute
Query Execution

Faster Operators

• Vectorized Processing
  • Single Instruction Multiple Data (SIMD)

• Operators on encoded data
  • Faster Dictionary Joins
  • In-memory aggregations using encoding

• Modern data structures
  • Segment Trees for window function evaluation
  • Cuckoo-hashing
Query Execution

Parallel Query Execution

• Operator Parallelism
  • Multi-core radix sort, merge sort, hash joins, aggregation, window function
  • Multi-core main-memory partitioning

• NUMA Awareness
  • Avoid cross-socket memory transfer

• Cache Friendliness
Query Optimization

In-memory costing

• Cost based selection of access paths
  • Selection of appropriate access path based on workload
  • Critical for mixed workloads

• Evaluation of predicate pushdown
  • Pushing down predicates to storage engine for selective columnar access

• Bloom filters
  • Joins replaced by in-memory columnar scan
Query Optimization

Network Cost & Plan Generation

• Cost-based Query Rewrites
  • Aware of network cost of operators

• Generate bushy plans
  • A must for snowstorm (multiple snowflake) schemas

• Fast optimization of real-time analytical queries
  • Enhanced pruning techniques
  • Optimal operator/join order based on distribution and local execution cost
Code Generation

Paradigm shift towards Just-In-Time (JIT) compilation

• Generate code in high level language C/C++
  • Run language compiler to generate executable code

• LLVM
  • Translate query into native machine code using LLVM compiler framework
  • Code & data locality
  • Predictable branches
Code Generation

- HHVM: Facebook’s Just-In-Time (JIT) compilation for PHP
Trends in Scalability
Distributed Storage

- Scale out in both throughput for main-memory and disk based systems
- Scale out in capacity for main-memory based systems
  - Main memory still a capacity bound resource
- Application-transparent distribution/rebalancing
  - Consistent Hashing
  - Rendezvous Hashing
Distributed Query Execution

• Distributed Query Plan
  • Generate optimal distributed plan factoring execution and network costs
    • e.g. Exploit co-located/location-aware joins
  • Generate one global plan in a single node and push the plan to other nodes
    • No local re-optimization, no local decisions
  • Generate global plan with possibility of node-level re-optimizations
    • Query plan is a set of SQL-like statements
    • Layers of primitive SQL operations glued together by SQL-like constructs

• Distributed statistics collection
Distributed Query Execution

• Fast Joins over high-speed networks (Infiniband)
  • Implementing joins to use Remote Direct Memory Access (RDMA)
  • Communication multiplexing, low-latency network scheduling

• Reducing network communication
  • Locally sensitive data shuffling
  • Novel distributed join algorithms: Track Join, Flow Join

• Scheduling
  • Gang scheduling distributed operators
Resource Management

- Cluster Resource Manager is needed to serve
  - Mixed OLTP and analytical workloads
  - OLTP itself could have short and long running transactions
  - Concurrent analytical queries requiring distributed resources

- Resource Manager
  - Job Managers
    - Admit, manage and schedule queries
    - Global resources are distributed across Job Managers
  - Resources are transferable across job managers

- QPS quota restrictions for analytic workloads
- Service based partitioning of cluster
High Availability and Fault Tolerance

- Redundancy of storage for availability
  - Highly relevant for main-memory based systems

- Replication types
  - Single master replication
    - Transactional replication - Two-phase commit
    - State machine based replication - Paxos/Raft
  - Multi-master replication
SAP HANA
In-Memory DBMS: Modern Hardware / Technological Context

- Multi-Core CPUs
  - Clock speed does not increase
  - More CPU cores
  - Small cache in CPU
- Large Memory
  - 1 TB RAM widely available
  - Slow compared to CPU
- Disk
  - “Unlimited” Size
  - Increasing Latency gap
HANA Table Types

- In-memory row-based storage
  - Fast OLTP operations (SELECT SINGLE, small K/FK joins, ...)
  - No compression
- In-memory columnar storage
  - Fast OLAP operations (massive scans, aggregation, ...)
  - Read-optimized, immutable Main Store
  - Write-optimized Differential Store (Delta)
  - Slow on OLTP operations
- Focus is on columnar storage
Main: Fast Queries

- Read-Optimized, Immutable Data Store
- Dictionary Compression
  - All data in columnar tables is dictionary compressed
  - Domain coding (aka dictionary compression)
  - Redundant literals only stored once (dictionary)
  - Dictionary is prefix-compressed
  - Dictionary is sorted (only for main)
- Reduces Storage Consumption Significantly
  - Up to factor 5 for typical SAP data schemes not uncommon
Delta: Update Support

- Write-Enabled Table Fragment Taking Updates
  - Only update operation on main is deleting rows
  - UPDATEs modelled as DELETE+INSERT
- Dictionary not Sorted
  - No need to recode index vector upon delete/insert
- Additional B-Tree for Efficient Lookup
  - Allows to quickly retrieve VID for value
  - Essential for fast unique checks upon insert
  - Can be used for range queries
Delta Merge

- Consolidation of Delta and Main into new Main
  - Improves query performance (especially for analytics)
  - Reduces memory footprint (no B-Tree necessary)
- Automatically Done by the System Based on Cost-Based Decision Function
  - Considers delta:main ratio, size in RAM and disk, system workload
  - Done on a per table-basis (actually: partition-based), parallelized on column-level
Indexes

- **Single-Value Indexes are Inverted Lists**
  - Mapping of valueIDs to rowIDs (actually UDIVs)
  - Lookup sequence:
    - Lookup valueID for current value in dictionary
    - Retrieve matching rows from index
  - Very little overhead compared to other indexes
  - Also beneficial for queries with limited selectivity

- **Multi-Value Indexes**
  - Handled by creating concat attributes first
  - Significant memory footprint
  - Prefer multiple single-column attributes instead
  - Often hurts for primary key
  - Hash-based alternative available
Optimize Compression for the Main Table Fragments

- **Beyond Dictionary Compression**
  - Sparse coding, cluster coding, indirect coding, run-length coding, ...
  - Best compression scheme to use
  - Depends on data distribution and ordering

- **Example: Sparse-Coding**
  - Remove the most frequent value
  - Store the removed positions
  - Additionally: Shorten the positions bit vector if the
    - most frequent value is also at the beginning
Challenges for Achieving High Compression Rates

- Most Compression Techniques Require Data to be Sorted Appropriately
  - Move all appearances of the most frequent value to the top (Prefix-Coding)
  - Form blocks of as few as possible values
- Issues
  - Sorting of one column depends on the sorting of another column
  - Solution space grows exponentially with the number of columns
  - Finding the optimal solution is infeasible (NP-complete)
- Solution
  - Greedy heuristics for a good approximation
  - Find a compromise between reduced run time and reduced memory consumption
Transaction Management and Synchronization

- Multi-Version Concurrency Control (MVCC)
  - Serializable as of snapshot Isolation
  - Record-Level Visibility Information
  - Used to delete from immutable main

- Record Locking for Updates
  - Conflict Detection
  - No Lock Escalation / MGL
Parallelization at All Levels

- Query Execution
  - Multiple User Sessions
  - Inter-Query Parallelism
  - Concurrent Operations Within a Query
  - Intra-Query Parallelism/Inter-Operator Parallelism
  - Multiple Threads for one Operator (e.g. aggregation)
  - Intra-Operator Parallelism

- Hardware
  - Multi-threading at processor core level
  - Vector processing (SIMD)
Single Instruction Multiple Data (SIMD)

- Scalar processing
  - Traditional mode
  - One instruction produces one result

- SIMD processing
  - With Intel® SSE / AVX / AVX2
  - One instruction produces multiple results
Query Execution Engines

- Vector Pipelining Engine
- Table Filter (qo3)
- Join Engine
- OLAP Engine
Vector Pipelining Engine

- Initially coming from the \((p*\text{time})\) row store
- Row-oriented and literal-based
  - Does not operate on value IDs
- Based on vector pipelining
  - Processes blocks of tuples at a time
- Compensates for missing functionality in other engines
  - Often used as “glue-code“ between other engines
Single Table Search aka Table Filter aka QO3

- Handles single-table search / filter operations on columnar data
  - Processes single table queries without aggregations
  - Also leveraged by other engines (JE, OLAP)
- Decides on right order to process selection predicates
  - Based on cardinality estimations
  - Uses top k statistics and dictionary information
  - Includes optimizations for in-list simplification (eliminate redundant predicates)
  - Decides on representation of intermediate results (bitvector vs vector<udivs>)
  - Estimation and execution phase interwoven
- Special case: OLTP fast stack
  - Bypasses regular qo3 operations for trivial OLTP queries (primary-key based selects)
  - Very limited functionality
Join Engine

- Default Column Store workhorse operating on dictionary-encoded data
- Processes everything too complex for QO3
  - Joins
  - Sorting
  - Aggregations
  - Offset/Limit
- Based on full semi-join reduction
  - Also fast for distributed processing
  - Cannot handle arbitrary join types (e.g. anti or theta joins)
  - Details on next slide
- Incorporates OLTP fast execution stack to run single threaded
JE Processing Workflow (Tables A and B)
Analytical aka OLAP Engine

- Initially Custom-tailored for SAP Business Warehouse (BW)
- Designed for Star- and Snowflake Schemas
- No general purpose engine, limited join support
- Core feature: Parallel aggregation
Algorithm Example

- Very Similar to Map and Reduce
- Map Step Processes Fact Table and Generates Local Results
- Reduce Step Joins Local Results
- No Synchronization Required
- Scales Linearly with Available CPUs
Performance Results for Parallel Aggregation

- Micro Benchmark
  - Cubesize: 228 213 237 rows
  - Resultsize: 4 469 335 rows
  - 32 physical CPUs (64 HT)
References

Row Format vs. Column Format

- **Transactions** run faster on row format
  - Example: Query or Insert a sales order
  - Fast processing few rows, many columns

- **Analytics** run faster on column format
  - Example: Report on sales totals by region
  - Fast accessing few columns, many rows
Dual Format Database

• BOTH row and column formats for same table

• Simultaneously active and consistent

• OLTP uses existing row format
  • Persisted in underlying storage with durability
  • Modified and accessed through in-memory buffer cache

• Analytics uses new In-Memory Column format
  • Maintained purely in-memory
Dual Format Database

- Highly optimized main memory utilization for *mixed workloads*
  - Buffer cache optimized for hit rates
  - Alleviating caching requirements for analytic indexes
  - Several compression levels for the column format

- **DOES NOT** require the entire database to be in-memory
  - Can be enabled at an Oracle table/partition level
  - Within an object, a smaller subset of columns can be selected to be maintained in-memory.
Oracle Real Application Cluster (RAC)

Allows a cluster of database compute servers (instances) to operate on multiple tenant Oracle databases persisted in shared storage abstracting each tenant as a single entity.
Distributed Architecture

Oracle Real Application Cluster (RAC)

Allows a cluster of database compute servers (instances) to operate on multiple tenant Oracle databases persisted in shared storage abstracting each tenant as a single entity
In-memory Column Store

- Shared-nothing NUMA-aware container of IMCUs

- In-Memory object
  - Collection of IMCUs across all instances

- A globally consistent instance-local ‘home location’ service
  - Seamless interfacing with row-format based layers
In-memory Compression Unit

- Columnar representation ‘populated’ from a large number of RDBMS table pages (Data Blocks)
- Contains contiguous compressed runs for each column (column compression units)
- A read-only snapshot as of a certain point in time
- Accompanied by a Snapshot Metadata Unit (SMU)
  - tracks changes due to DMLs
- Undergoes heuristics based online ‘repopulation’ as changes accumulate on the SMU
Vector Processing

- Each CPU core scans local in-memory columns
- Scans use super fast SIMD vector instructions
- Originally designed for graphics & science
- **Billions of rows/sec** scan rate per CPU core
  - Row format is millions/sec
Operation Pushdown - Predicates

- When possible, push operations down to In-Memory scan
  - Greatly reduces # rows flowing up through the plan
  - Similar to Exadata smart scan

- For example:
  - Predicate Evaluation (for qualifying predicates - equality, range, etc.):
    - Inline predicate evaluation within the scan
    - Each IMCU scan only returns qualifying rows instead of all rows
Example: Find total sales in outlet stores

- **Bloom Filter:**
  - Compact bit vector for set membership testing
- **Bloom filter pushdown:**
  - Filtering pushed down to IMCU scan
  - Returns only rows that are likely to be join candidates
- **Joins tables** 10x faster
In-memory Storage Indexes

- **Min-Max Pruning**
  - Min/Max values serve as storage index
  - Check predicate against min/max values
  - Skip entire IMCU if predicate not satisfied

- Can prune for many predicates including equality, range, inlist, etc.

- Eliminates processing unnecessary IMCUs

**Example:** Find stores with sales greater than $10,000
In-memory Storage Indexes

• Avoid evaluating predicates against every column value
  • Check range predicate against min/max values
    • As before, skip IMCUs where min/max disqualifies predicate
  • If min/max indicates all rows will qualify, no need to evaluate predicates on column values

Example: Find stores with sales between $8000 and $14000

- Min $4000
  Max $7000
  Skip IMCU

- Min $8000
  Max $13000
  Skip Evaluation

- Min $13000
  Max $15000
  Needs evaluation
Aggregations

Example: Report sales of footwear in outlet stores

- Dynamically creates in-memory report outline
- Fast fact scan fills in in-memory vectors
- Reports ran faster
  - without predefined cubes
Complex OLTP slowed by Analytic Indexes

- Most Indexes in complex OLTP (e.g. ERP) databases are only used for analytic queries
- Inserting one row into a table requires updating 10-20 analytic indexes: Slow!
- Indexes only speed up predictable queries & reports
Column Store Replaces Analytic Indexes

- Fast analytics on **any** columns
  - Better for unpredictable analytics
  - Less tuning & administration

- Column Store not persistent so update cost is much lower
  - OLTP and batch run faster
IMCU Repopulation

DML Operations on Rows and Corresponding IMCU
IMCU Repopulation

DML Operations on Rows and Corresponding IMCU
IMCU Repopulation based on Scan and Modification Thresholds
Application-transparent IMCU Distribution

• Serialized Consensus Generation
  • Elect leader node for a table/partition
  • Leader node generates and broadcasts consistent IMCU population contexts
  • Followers accept and wait for Leader to downgrade
  • Leader downgrades to Follower
Application-transparent IMCU Distribution

- Decentralized IMCU Population per Follower Node
  - Decode IMCU population contexts
  - Uses rendezvous hashing to determine home locations (cluster node + NUMA node)
  - If home location matches node id
    - Physical population of IMCU
    - Registration in in-memory home location index
  - Otherwise
    - Registration in in-memory home location index

- Eventual global consistency of home location index across cluster
Distribution Keys

- **Distribution by Row-id Range**
  - The first rowid of the IMCU context
  - Non-partitioned tables

- **Distribution by Partition**
  - Applicable for partitions as well as top-level partition for composite partitioned tables
  - Partitioning key associated with the IMCU context

- **Distribution by Sub-Partition**
  - Applicable for composite sub-partitions
  - Sub-Partitioning key of the IMCU context
Distributed Query Scale-out/Scale-up

- Query coordinated on a random instance
  - Uses local home location index to generate IMCU-aligned work-sets (granules)

- Compiles granules based on <instance, NUMA node> and assigns granules to affined distributors
Distributed Query Scale-out/Scale-up

- Local parallel execution processes de-queue work-sets from their affined distributors
  - Perform fully local NUMA-affined IMCU scans

- Seamless distribution awareness to SQL execution engine ensures
  - No plan changes
  - No query recompilations or rewrites
In-memory Aware Query Optimizer

- Enhanced Cost Model
  - New cost formula for in-memory access
  - Works on combination of disk pages + IMCUs
  - Storage index based pruning cost
  - Decompression cost
  - Predicate evaluation cost

- Hybrid access path
  - For partially populated tables
  - Favor index scans over full table scans and vice versa
    - Depending on percentage of table in memory
In-memory Aware Query Optimizer

- Predicate pushdown
  - Figures out predicates that can be evaluated on compressed formats
  - Splits and pushes them down the query plan to in-memory scans

- Degree-of-parallelism
  - Derive DOP based on cost model
  - Consider partially populated tables
  - At-least one parallel execution process per node
  - NUMA-local access best effort
In-memory Aware Partition-wise Joins

Lineitem table partitioned on l_orderkey
sub-partitioned on l_custkey

Orders table partitioned on o_orderkey

Joins pushed down by partitioning key

Distributed by sub-partitioning key

partition wise join (disk scans + no IPC)
broadcast joins (in-memory scans + IPC)

Cost model to select optimal plan

Distributed by partitioning key

partition wise join (in-memory scans + no IPC)
Fault Tolerance

- Similar to storage mirroring
- Duplicate in-memory columns on another node
  - Enabled per table/partition
  - E.g. only recent data
  - Application transparent
- Downtime eliminated by using duplicate after failure
Analytic Throughput in Mixed Workload

- 200 clients
  - Each executing DMLs + analytic scans on a single table
  - For 8 hours
- Single node:
  - 101 columns wide table (initial size 12G)
- 8 nodes:
  - 8x scaled version of the same table
- Sustained analytic query throughput scale out
- OLTP latency sustained: sub-millisecond
Information Lifecycle Management

- Size not limited by memory
- Data transparently accessed across tiers
- Each tier has specialized algorithms & compression
Easy Configuration

- Configure Memory Capacity
  - `inmemory_size = XXX GB`

- Configure tables or partitions to be in memory
  - `alter table | partition ... inmemory;`

- Later drop analytic indexes to speed up OLTP
MemSQL: Distributed Architecture

- Two-tiered architecture
- Aggregators
  - Cluster Monitoring/Failover
  - Query Re-routing
  - Metadata management
- Leaves
  - Store data
  - Process SQL queries
MemSQL: Storage Types

- In-memory store for extremely fast and concurrent OLTP
  - Operational analytics when data set fits in memory
  - Very fast writes, low-latency queries

- Disk based operational/online column store
  - Online data ingestion via inserts, updates
  - Low-latency queries

- Join row-store and column store data
MemSQL: Skiplist Indexes for rowstore
MemSQL: Skiplist Indexes for rowstore

• Ordered data structure
  • Expected $O(\log(n))$ lookup, insert and delete

• Optimized for memory

• Implemented lock free
  • Support highly concurrent workloads on multi-cores

• Few instructions to insert, delete, search
MemSQL: Skiplist Indexes for rowstore

Concurrency control

- No latches
- Row-locks are implemented with 4-byte futexes
- Read Committed and SNAPSHOT isolation
MemSQL: Operational column store

- Clustered Column Store Index
  - Primary table storage
  - Needs clustered column key(s)
  - Data is stored in the key column order

- Row Segment
  - Represents logical set of rows in the column store index
  - Stores row count, delete bitmask
MemSQL: Operational column store

Column Segment
- Part of row segment
- One column segment per column
- Values in column segments are stored in same logical order across column segments
- Typically contains tens of thousands of rows
- Basic Unit of Storage
- Metadata maintained in-memory via row-store table
  - Deleted row masks, Min/Max

<table>
<thead>
<tr>
<th>Price 4-15</th>
<th>ProductId 1-15</th>
<th>Color Black-White</th>
<th>Qty 2-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>15</td>
<td>Red x 2</td>
<td>2 x 5</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>Black</td>
<td></td>
</tr>
<tr>
<td>10 x 2</td>
<td>6</td>
<td>Red</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>White</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MemSQL: Operational column store

Sorted Row Segment Group
- Set of row segments that are sorted together on the key columns
- No row segments with overlapping value ranges for key columns

<table>
<thead>
<tr>
<th>Price 4-15</th>
<th>ProductId 1-15</th>
<th>Color Black-White</th>
<th>Qty 2-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>15</td>
<td>Red x 2</td>
<td>2 x 5</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>Black</td>
<td></td>
</tr>
<tr>
<td>10 x 2</td>
<td>6</td>
<td>Red</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>White</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Price 20-25</th>
<th>ProductId 2-12</th>
<th>Color Black-White</th>
<th>Qty 2-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 x 4</td>
<td>3</td>
<td>Black</td>
<td>2 x 5</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>Red x 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>White</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MemSQL: Operational column store

Writes to the column store
• Continually allow inserts, updates and deletes while servicing queries
• Use row-store for fast small-batch writes
• Row-store backed row-segments
**MemSQL: Operational column store**

- **Reads**
  - Reads against column store are served with SNAPSHOT isolation

- **Inserts & Loads**
  - Small batch inserts goes via the row-store backed row segment
  - Large batch inserts & Loads directly go to the column store and creates a new column-store backed row segment

- **Deletes**
  - Marks row as deleted in metadata, data remains in column segments
  - Segments with all deleted rows are removed and others are compacted

- **Updates**
  - Implemented as delete followed by an insert
MemSQL: Operational column store

Managing Row Segment Groups

• Loads and Inserts could potentially create a large number of unsorted row segment groups
• Background Merger keeps merging sorted row segment groups
  • Log Structured Merge
  • Optimistic concurrency control
MemSQL: Distributed Query Execution

**Aggregator Tier:**
Distributed Optimizer
Transforms to Partial SQL

**Leaf Tier:**
Executes SQL Over Local Data in Parallel

1: Raw SQL
2: Partial SQL
3: Data Shuffling
4: Partial Results
5: Final results
MemSQL: Distributed Query Execution

• Distributed Query Plan created on Aggregator
  • Query Plan is set of SQL-like statements
  • Aggregator sends plan to all leaves

• Layers of primitive operations glued together
  • Full SQL on leaves
  • Leaves could potentially re-optimize SQL
  • RESULT and REMOTE tables for distributed operations
  • Distributed Execution Primitives: e.g. Reshuffle, Broadcast
MemSQL: Distributed Query Execution

```
SELECT SUM(l_extendedprice)
FROM lineitem, part
WHERE l_partkey = p_partkey
AND l_shipdate >= Date('1993-01-01')
AND l_shipdate < Date('1993-01-01')
+ INTERVAL '13' month;
```

```
CREATE RESULT TABLE arrange_N
PARTITION BY (l_partkey)
AS
SELECT ...
```

```
SELECT SUM(l_extendedprice) AS sum_price
FROM REMOTE(arrange_N(N))
JOIN 'part'
WHERE arrange_N.l_partkey = part.p_partkey
```

```
SELECT SUM(l_extendedprice) AS sum_price
FROM lineitem
WHERE l_partkey = p_partkey
```

```
Local SQL Execution
```

```
Merge Results
```

```
Reshuffle Data
```

```
Result Table
```

```
Distributed Optimizer
```

```
mysql_query()
```

```
mysql_use_result()
```

```
App Tier
```

```
Agg Tier
```

```
Leaf Tier
```
RESULT Tables

- Shared, cached results of SQL queries
- Similar to RDD in Spark
- Shares scans/computations across readers
- Supports streaming semantics
MemSQL: Query Optimizer

- Industry strength modular query optimizer for real-time analytics
  - Rewriter
  - Enumerator
  - Planner
  - Designed and developed to generate optimal distributed plans FAST

- Query rewrites based on execution and network costs
- Generation of bushy plans
- Extremely fast plan enumeration

More details in Industrial Track Presentation
MemSQL: Code Generation

• Queries are natively compiled into machine code using LLVM

• SQL Execution Plans are converted to MemSQL Plan Language (MPL) Abstract Trees

• MPL Plans are flattened into MemSQL Byte Code (MBC)

• MBC Plans are transformed to LLVM bitcode for compilation to x86_64 machine code.
MemSQL: Code Generation
MemSQL: Row-Store Durability

- Indexes are not materialized on disk
  - Indexes reconstructed on the fly during recovery
- Need to log PK, data
- Take full database snapshots periodically

Recovery
- Replay latest snapshot and every log file since
- No partially written state to disk, so no undos
MemSQL: Column-Store Durability

- Metadata is stored in row-store table
- Column store segments synchronously written to disk

Recovery
- Replay metadata
Impala - Query Execution Engine over HDFS
Query Planning - Distributed Plans

Single-Node Plan:
- TopN
  - Agg
    - HashJoin
      - Scan: t1
  - HashJoin
    - Scan: t2
    - Scan: t3

Distributed Plan:
- TopN
  - MergeAgg
    - Pre-Agg
      - HashJoin
        - Scan: t1
        - Scan: t2
        - Scan: t3
    - at HDFS DN
      - at HBase RS
      - at coordinator
  - Merge
    - TopN
      - hash t1.custid
      - hash t1.id1
      - hash t2.id
Query Execution

- Written in C++ for minimal cycle and memory overhead
- Leverages decades of parallel DB research
- Partitioned parallelism
- Pipelined relational operators
- Batch-at-a-time runtime
- Focussed on speed and efficiency
- Intrinsics/machine code for text parsing, hashing, etc.
- Runtime code generation with LLVM
Runtime Code Generation

- Uses llvm to jit-compile the runtime-intensive parts of a query
- Effect the same as custom-coding a query:
  - Remove branches, unroll loops
  - Propagate constants, offsets, pointers, etc.
  - Inline function calls
- Optimized execution for modern CPUs (instruction pipelines)
Example

```cpp
IntVal my_func(const IntVal& v1, const IntVal& v2) {
    return IntVal{v1.val * 7 / v2.val};
}

SELECT my_func(col1 + 10, col2) FROM ...
```

(interpreted) $(\text{col1} + 10) \times 7 / \text{col2}$

(codegen’ed)
Performance with Code generation

10 node cluster (12 disks / 48GB RAM / 8 cores per node)
~40 GB / ~60M row Avro dataset
Resource Management in Impala

- Admission control and Yarn-based RM cater to different workloads
  - Use admission control for:
    - Low-latency, high-throughput workloads
    - Mostly running Impala, or resource partitioning is feasible
  - Use Llama/Yarn for:
    - Mixed workloads (Impala, MR, Spark, ...) and resource partitioning is impractical
    - Latency and throughput SLAs are relatively relaxed
Kudu - Operational Columnar Storage

Flowchart:
- **Incoming Data (Messaging System)**
  - HBase
  - Parquet File
- **Impala on HDFS**
  - Historic Data
    - Most Recent Partition
    - New Partition
  - Reporting Request
- **Impala on Kudu**
  - Incoming Data (Messaging System)
  - Reporting Request
Kudu - Basic Design

- Typed storage
- Basic Construct: *Tables*
- Tables broken down into *Tablets* (roughly equivalent to partitions)
- N-way replication
- Maintains consistency through a Paxos-like quorum model (Raft)
- Architecture supports geographically disparate, active/active systems
Kudu - Basic Design

- Inserts buffered in an in-memory store (like HBase’s memstore)
- Flushed to disk
- Columnar layout, similar to Apache Parquet
- Updates use MVCC (updates tagged with timestamp, not *in-place*)
- Allow “SELECT AS OF <timestamp>” queries and consistent cross-tablet scans
- Near-optimal read path for “current time” scans
- No per row branches, fast vectorized decoding and predicate evaluation
- Performance worsens based on number of recent updates
Academic Perspective
Open Challenges