NewSQL, SQL on Hadoop

Prof. Dr. Andreas Thor
Hochschule für Telekommunikation Leipzig (HfTL)
thor@hft-leipzig.de

2nd International ScaDS Summer School on Big Data, July 12, 2016
Agenda

• SQL on Hadoop
  – Motivation: Why MR is not enough?
  – Hadoop-based Frameworks
  – Translating SQL to MapReduce, Optimizing data flows

• NewSQL
  – Motivation: RDBMS and the Cloud
  – Types of NewSQL systems
  – In-Memory Databases, Data Partitioning

• No complete overview of all tools
  • Focus on ideas / techniques
Data analysis / Queries on Big Data

- Simple aggregations, ad-hoc analyses
  - Number of clicks / page views per day / month
  - How many foto uploads on New Year’s Eve 2015? How many tweets during the EURO 2016 final?

- Preprocessing for Data Mining
  - Identify user groups / types
  - Find suspicious / frequent patterns in UGC (user generated content)

- If your data is in Hadoop
  - … use the query capabilities of your NoSQL store!
  - … write a MapReduce / Spark program to analyze it!

- Really?
Data Analysis: Access Frequency Skew

- Empirical analysis from companies reveals access frequency skew
  - Zipf-like distribution: Few files account for a very high number of accesses
  - ~90% of all files accessed only once

SQL-based Data Analysis

• Copy to Relational Database / Data Warehouse?
  – Development overhead for rarely used files
  – Import is inefficient

• High-level language for Hadoop-based data analyses
  – Data analysts do not need to be able to program MapReduce, Spark etc.
  – Efficient re-use of scripts / workflows for similar analysis tasks

• SQL interface for Hadoop needed
  – SQL is declarative, concise
  – People know SQL
  – Interface with existing analysis software
  – Can be combined with MapReduce / Spark
Hadoop Ecosystem (simplified)

Data Type / Algorithm
- SQL
- Graph
- Machine Learning
- ...

Execution Engine
- MapReduce, Spark, Tez

Cluster Management
- Hadoop Yarn

Data Storage
- HDFS
Processing Frameworks for Hadoop

Mark Grover: Processing frameworks for Hadoop, 2015
radar.oreilly.com/2015/02/processing-frameworks-for-hadoop.html

Andreas Thor: NewSQL, SQL on Hadoop
Hadoop-based Data Analysis Frameworks


Andreas Thor: NewSQL, SQL on Hadoop
Apache Hive

- Data Warehouse Infrastructure on Hadoop
  - Hive 2.1 (June 2016) for Hadoop 2.x

- “Hive = MapReduce + SQL”
  - SQL is simple to use
  - MapReduce provides scalability and fault tolerance

- HiveQL = SQL-like query language
  - Extendible with MapReduce scripts and user-defined functions (e.g., in Python)
Hive: Metastore

• Mapping files to logical tables
  - Flexible (de)serialization of tables (CSV, XML, JSON)

Table

Partitions (multiple levels)

HDFS Files (split into hash buckets)

• Table corresponds to HDFS directory: /clicks
  - Subdirectories for partitioning (based on attributes): /clicks/d=20160710
  - Bucketing: Split files into parts

• Advantage: Direct data access, i.e., no transformation / loading into relational format

• Disadvantage: No pre-processing (e.g., indexing)
Hive: Workflow

1. User issues SQL query
2. Hive parses and plans query
3. Query converted to MapReduce
4. MapReduce run by Hadoop

Abadi et. al: SQL-on-Hadoop Tutorial. VLDB 2015
SELECT g1.x, g1.avg, g2.cnt
FROM (SELECT a.x, AVG(a.y) AS avg
FROM a
GROUP BY a.x) g1
JOIN (SELECT b.x, COUNT(b.y) AS cnt
FROM b
GROUP BY b.x) g2
ON (g1.x = g2.x)
ORDER BY g1.avg
Hive: Query Optimization

- Query optimization employs ideas from database research
  - Logical (rule-based) transformations
  - Cost-based optimizations

- Projection / selection pushdown
  - Remove unnecessary attributes / records as early as possible

- Adaptive implementations, e.g., joins
  - Based on statistics (e.g., number of records, min-max values)

http://de.slideshare.net/ragho/hive-user-meeting-august-2009-facebook
Semi-structured JSON data vs. relational data

• JSON data (collection of objects)

```json
```

• Relational: Nested table with multi-valued attributes

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>time</th>
<th>user</th>
<th>camera</th>
<th>info</th>
<th>tags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>width</td>
<td>height</td>
</tr>
<tr>
<td>1</td>
<td>fish.jpg</td>
<td>17:46</td>
<td>bob</td>
<td>nikon</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>trees.jpg</td>
<td>17:57</td>
<td>john</td>
<td>canon</td>
<td>30</td>
<td>250</td>
</tr>
<tr>
<td>3</td>
<td>snow.png</td>
<td>17:56</td>
<td>john</td>
<td>canon</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>4</td>
<td>hawaii.png</td>
<td>17:59</td>
<td>john</td>
<td>nikon</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>hawaii.gif</td>
<td>17:58</td>
<td>bob</td>
<td>canon</td>
<td>320</td>
<td>128</td>
</tr>
<tr>
<td>6</td>
<td>island.gif</td>
<td>17:43</td>
<td>zztop</td>
<td>nikon</td>
<td>640</td>
<td>480</td>
</tr>
</tbody>
</table>

**SQL to MapReduce: Example**

**map**

```javascript
function (doc) {
  if (doc.user == "john") {
    emit(doc.camera,
        doc.info.size);
  }
}
```

**reduce**

```javascript
function (key, values) {
  sum = 0;
  foreach (v:values) sum += v;
  return sum/values.length;
}
```

**Example**

```json
{ id: 1, user: "bob" ... }
{ id: 2, user: "john" ... }
{ id: 3, user: "john" ... }
{ id: 4, user: "john" ... }
{ id: 5, user: "bob" ... }
{ id: 6, user: "zztop" ... }
```

**Shuffle & Sort**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>canon</td>
<td>32091</td>
</tr>
<tr>
<td>canon</td>
<td>1253</td>
</tr>
<tr>
<td>nikon</td>
<td>92834</td>
</tr>
</tbody>
</table>

**Reduce**

<table>
<thead>
<tr>
<th>key</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>canon</td>
<td>[32091, 1253]</td>
</tr>
<tr>
<td>nikon</td>
<td>[92834]</td>
</tr>
</tbody>
</table>

**SELECT**

```sql
SELECT camera, AVG(info.size) FROM Pictures WHERE user="john" GROUP BY camera
```
## SQL to MapReduce

<table>
<thead>
<tr>
<th>SQL</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>Filter in map function</td>
</tr>
<tr>
<td>WHERE user = 'John'</td>
<td>if (user==‘John’) { emit ( ... ); }</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Projection</td>
<td>Map output value</td>
</tr>
<tr>
<td>SELECT camera, size</td>
<td>emit ( ... , {camera, size} );</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Grouping</td>
<td>Map output key = grouping attribute(s)</td>
</tr>
<tr>
<td>GROUP BY camera</td>
<td>emit ( camera, ... );</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Aggregation</td>
<td>Computation in reduce function</td>
</tr>
<tr>
<td>SELECT AVG (size)</td>
<td>average ( [size1, size2, ... ]);</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested Queries</td>
<td>Sequence of MapReduce programs</td>
</tr>
<tr>
<td>FROM (SELECT ... FROM ...) AS T</td>
<td>Output of MR1 (inner query)= input to MR2 (outer q.)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorting</td>
<td>Map output key = sorting attribute(s)</td>
</tr>
<tr>
<td>ORDER BY camera</td>
<td>Requires single reducer or range partitioner</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Join</td>
<td>-- see next slides --</td>
</tr>
<tr>
<td>FROM R JOIN S ON (R.b=S.b)</td>
<td></td>
</tr>
</tbody>
</table>
Repartition Join (for Equi Join)

- Naïve approach
  - Map output: key = join attribute, value = relation + tuple (relevant attributes)
  - reduce: all pairs from different relations
Repartition Join: Extended Key

• Reducer needs to buffer all values per key
  – No specific order of reduce values in list; sequential access to list only

• Key extension (+ adjusted grouping and sorting comparators)
  – Extend map output key by relation name; group by attribute only
  – Sorting so that keys of small relation (S) are before keys of large relation (R)
    → Reduce buffering for S keys only

• Example

<table>
<thead>
<tr>
<th>Naïve</th>
<th>Extended Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, R:a₄)</td>
<td>(2:S, c₂)</td>
</tr>
<tr>
<td>(2, S:c₂)</td>
<td>(2:R, a₉)</td>
</tr>
<tr>
<td>(2, R:a₆)</td>
<td>(2:R, a₆)</td>
</tr>
<tr>
<td>(2, R:a₇)</td>
<td>(2:R, a₇)</td>
</tr>
<tr>
<td>(2, R:a₈)</td>
<td>(2:R, a₈)</td>
</tr>
<tr>
<td>(2, S:c₉)</td>
<td></td>
</tr>
</tbody>
</table>
**Broadcast Join**

- **Repartition Join:** Large map output
  - All tuples are sorted between map and reduce $\rightarrow$ high network traffic

- **Common scenario:** $|R| >> |S|
  - Example: Logfile $\bowtie$ User

- **Join computation in the map phase; no reduce phase**
  - Use small relation ($S$) as additional map input

- **Data transfer**
  - Small relation is sent to all $n$ nodes $\rightarrow n \cdot |S|$  
  - No transfer of $R$: map task consumes local map partition
  - Repartition-Join: $|R| + |S|$
Evaluation

Data size sent through the network

<table>
<thead>
<tr>
<th>#record (Relation S)</th>
<th>Repartition (ext. Key)</th>
<th>Broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3 ( \cdot 10^6 )</td>
<td>145 GB</td>
<td>6 GB</td>
</tr>
<tr>
<td>10 ( \cdot 10^6 )</td>
<td>145 GB</td>
<td>195 GB</td>
</tr>
<tr>
<td>300 ( \cdot 10^6 )</td>
<td>151 GB</td>
<td>6240 GB</td>
</tr>
</tbody>
</table>

- Prefer broadcast for small $S$
- Repartitioning: Benefit of extended key

Blanas et al.: A Comparison of Join Algorithms for Log Processing in MapReduce. SIGMOD 2010
### SQL on Hadoop

<table>
<thead>
<tr>
<th></th>
<th>Apache Hive</th>
<th>Apache Spark SQL</th>
<th>Apache Drill</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation Mode</strong></td>
<td>Batch</td>
<td>Procedural</td>
<td>Interactive</td>
</tr>
<tr>
<td><strong>Scenario</strong></td>
<td>Data-Warehouse-like queries</td>
<td>Complex Data Analysis Algorithms (e.g., Machine Learning)</td>
<td>Interactive Data Discovery (Exploration)</td>
</tr>
<tr>
<td></td>
<td>ETL processing</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td>high</td>
<td>medium</td>
<td>low</td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td>HiveQL (SQL-like)</td>
<td>Mix Spark code (Java / Scale) with SQL</td>
<td>ANSI SQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data Sources</strong></td>
<td>Hadoop</td>
<td>Hadoop, Hive Tables, JDBC</td>
<td>Hadoop, NoSQL (joining different data sources)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Schema</strong></td>
<td>Relational, Pre-defined</td>
<td>Relational, Pre-defined</td>
<td>JSON, On-the-fly („schema-free“)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Translates into</strong></td>
<td>MapReduce &amp; Spark</td>
<td>Spark</td>
<td>--</td>
</tr>
</tbody>
</table>
From SQL on Hadoop to NewSQL
NewSQL: Definition

• “… delivers the scalability and flexibility promised by NoSQL while retaining the support for SQL queries and/or ACID, or to improve performance for appropriate workloads.” (451 group)

• NewSQL: An Alternative to NoSQL and Old SQL for New OLTP Apps (by Michael Stonebraker)
  – SQL as the primary interface
  – ACID support for transactions
  – Non-locking concurrency control
  – High per-node performance
  – Scalable, shared nothing architecture

RBDMS Design Principles

• RBDMS developed for shared-memory and (later) shared-disk architectures
  – Cloud / Data Center: Shared Nothing

• RDBMS store data on hard-drive disks; main memory for caching only
  – Cloud / Data Center: large amount of main memory affordable; solid state disks

• RDBMS implement Recovery using disk-based Logfiles
  – Cloud / Data Center: Fast recovery via data copying through the network possible

• RDBMS support Multi-Threading (on a single core)
  – T2 can be started if T1 is still waiting for data (from disk) → long transactions should not block short transactions → low latency
  – Cloud / Data Center: Multi core nodes, large main memory

Amazon EC2 price history for 1TB main memory
RDBMS Overhead

- “Removing those overheads and running the database in main memory would yield orders of magnitude improvements in database performance”

Useful work
- Retrieve / update data

Index Management

Buffer Management
- Mapping records to pages for block-wise storage on disk
  → Not needed anymore for In-Memory-Databases

Logging
- Write & read log files (write-ahead logging)
- ReDo Recovery (after outage), UnDo Recovery (after transaction failures)
  → ReDo by “Copy from Replica” possible; avoid UnDo cases

Locking & Latching
- Concurrency control (locking protocols), deadlock handling
- Short-term locks in multi-threading (latching)
  → Reduce overhead for Isolated Execution (e.g., no multi-threading)

Harizopoulos, S. et. al., “OLTP: Through the Looking Glass and What We Found There,” SIGMOD, June 2008
Andreas Thor: NewSQL, SQL on Hadoop
HStore: Overview

• Distributed, row-store-based, main memory relational database
  – Cluster of nodes (shared-nothing); multiple sites per node
  – Site = single-threaded daemon on a single CPU → no latching
  – Row-store (B-Tree) in main memory → no buffer management

• Transactions
  – No ad-hoc SQL queries; pre-defined stored Procedures (SP) only
  – Classification of transactions (e.g., “single / multi partition”, “two phase”)
  – Global ordering of transactions → strong consistency
  – ACID
  – Direct data access / transfer (no ODBC)

• Recovery
  – Replica-based recovery → no logging needed

• VoltDB (commercial) ≈ HStore (open source / research prototype)
HStore: Site Architecture

Jones, Abadi, and Madden, "Low overhead concurrency control for partitioned main memory databases," SIGMOD 2010
OLTP transaction in Web Applications

• Focus of web applications: Scalability, scalability, scalability
  – Limited flexibility on transactions is ok

• Observations: Transactions …
  – … often touch data of current user only
  – … modify few records only
  – … are known a-priori, i.e., no ad-hoc queries needed
  – … are comparatively simple
Data Partitioning: Tree Schema

- Most schemas (for web applications) are “tree schemas”
  - One (or more) root tables (e.g., warehouse)
  - Other tables have (multiple) one-to-many relationships to root table

TPC-C Schema

Schema Tree

Andrew Pavlo: NewSQL, 2012
Horizontal Partitioning

- Horizontal partitioning of the root table
  - Child tables are partitioned accordingly
  - Replication of unrelated tables

Goal: Single-Partition Transactions
HStore: Infrastructure

Clients

H-Store

Central Coordinator

Node 1
- Data Partition 1: Primary
- Data Partition 2: Primary

Node 2
- Data Partition 3: Primary
- Data Partition 4: Primary

Node 3
- Data Partition 1: Backup
- Data Partition 4: Backup

Node 4
- Data Partition 3: Backup
- Data Partition 2: Backup
Single Partition Transactions

- Client sends single partition transaction to (node of) primary partition
  - Primary forwards to Secondary (Backup)
  - Execute transactions by node_id + timestamp (nodes are time-synchronized)
- Independent, parallel execution on all partitions
  - Each nodes achieve the same result (commit oder abort)
  - Primary sends back result to client after receiving “acknowledge” from all secondaries → Strong Consistency
  - If node fails → copy partition replica → No ReDo logging
- Transactions are executed sequentially on every node (single-thread) → No Concurrency Control
- “Two phase” transaction
  - Format: “read(s), check for consistency, write(s)”
  - → No UnDo logging necessary

\[
\begin{align*}
x &= \text{read}(a) \\
y &= \text{read}(b) \\
y &\geq 100 \text{?} \\
\text{write}(a, x+100) \\
\text{write}(b, y-100)
\end{align*}
\]
Multi Partition Transactions

• Multi Partition Transaction are controlled by a central Coordinator
  – Multiple coordinators possible but preserving global order of transactions

• Execution
  – Divide Multi Partition Transaction in fragments that are sent to all partitions
  – UnDo buffer for undoing transactions in case of failures (e.g., consistency violations)

• Two-Phase Commit Protocol
  – Coordination protocol to achieve global result (commit / abort) in distributed environment
# NewSQL: Overview

<table>
<thead>
<tr>
<th>Type</th>
<th>New Architectures</th>
<th>New SQL Engines</th>
<th>Middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developed “from scratch”</td>
<td>“Plugin” to existing RDBMS (e.g., MySQL)</td>
<td>Additional layer on top of RDBMS</td>
<td></td>
</tr>
<tr>
<td>Examples</td>
<td>H-Store / VoltDB</td>
<td>MySQL Cluster</td>
<td>Schooner MySQL</td>
</tr>
<tr>
<td></td>
<td>Google Spanner</td>
<td>ScaleDB</td>
<td>ScaleArc</td>
</tr>
<tr>
<td></td>
<td>MemSQL</td>
<td>Tokutek</td>
<td>ScaleBase</td>
</tr>
<tr>
<td></td>
<td>NuoDB</td>
<td>...</td>
<td>dbShards</td>
</tr>
<tr>
<td></td>
<td>Clustrix</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Characteristics</td>
<td>Designed for in-memory (or flash) as primary data store</td>
<td>Reuse components from RDBMS framework</td>
<td>Transparent clustering/sharding for scalability</td>
</tr>
</tbody>
</table>
Summary

• SQL on Hadoop: „Add SQL to NoSQL“
  – Frameworks leveraging (parts of) the Hadoop infrastructure
  – SQL-like queries on (semi-)structured data (files) and NoSQL (OLAP)
  – Techniques: SQL-to-MR-translation, Query optimization, Metadata

• NewSQL: „Add Scalability to RDBMS“
  – New type of RDBMS in a shared-nothing cluster
  – SQL and ACID transactions (OLTP)
  – Techniques: In-Memory, Data Partitioning, Pre-defined SQL statements

Thank you!