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

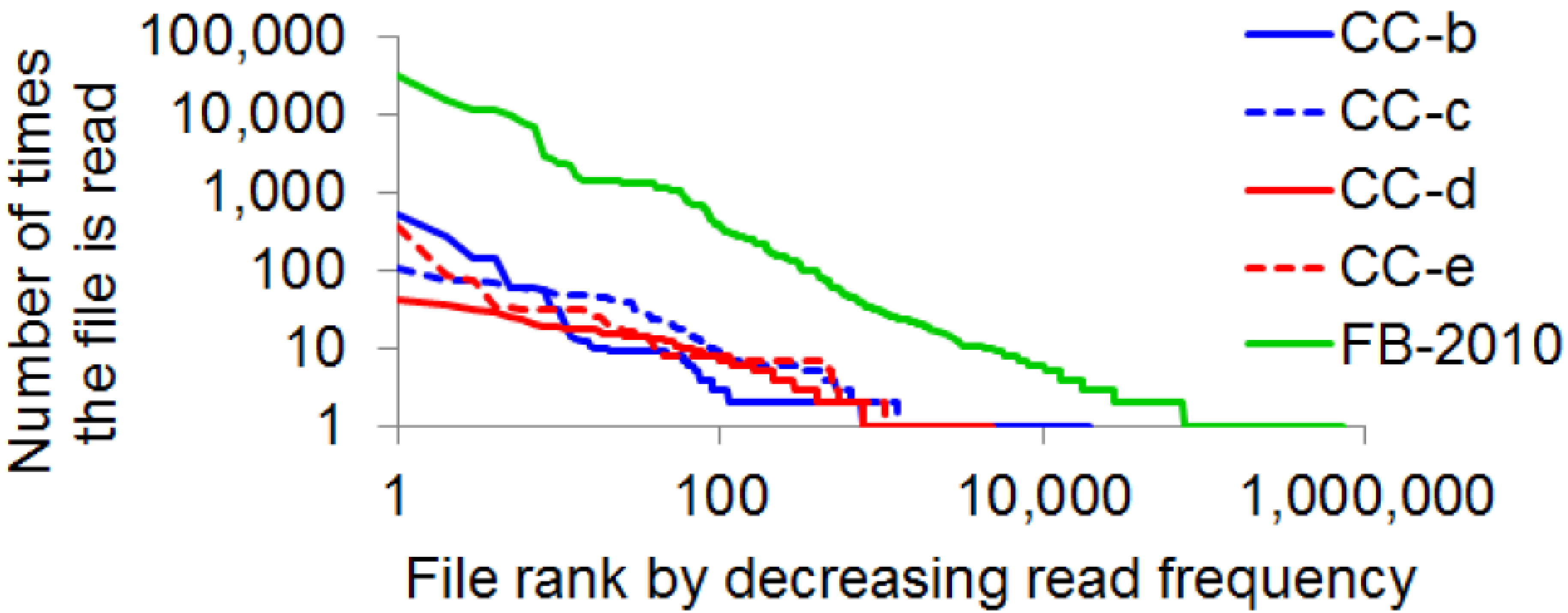
Data analysis / Queries on Big Data

- 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



Chen et. al: Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. VLDB 2012

SQL-based Data Analysis

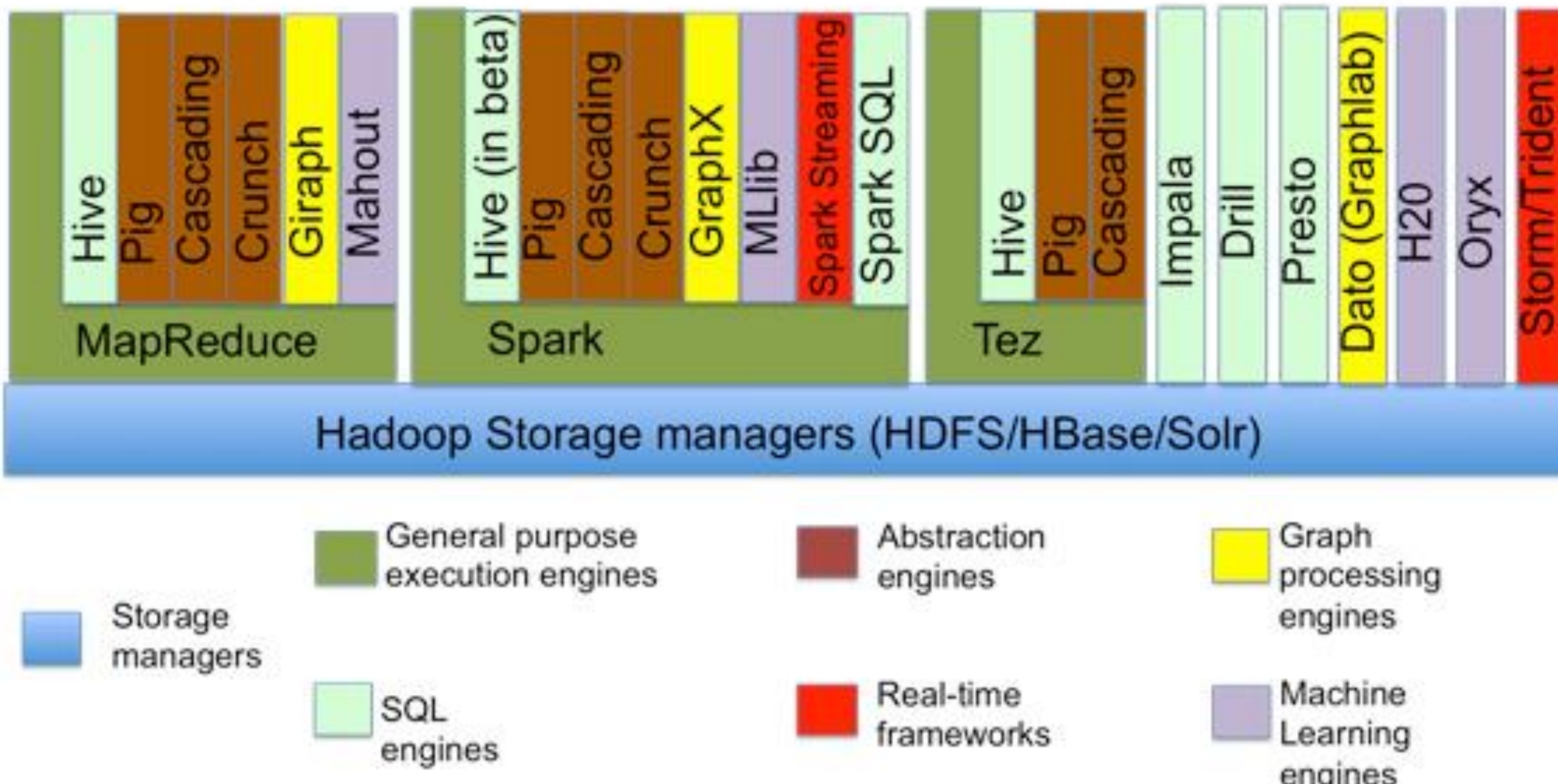
- Copy to Relational Database / Data Warehouse?
  - Development overhead for rarely used files
  - Import is inefficient

- 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)

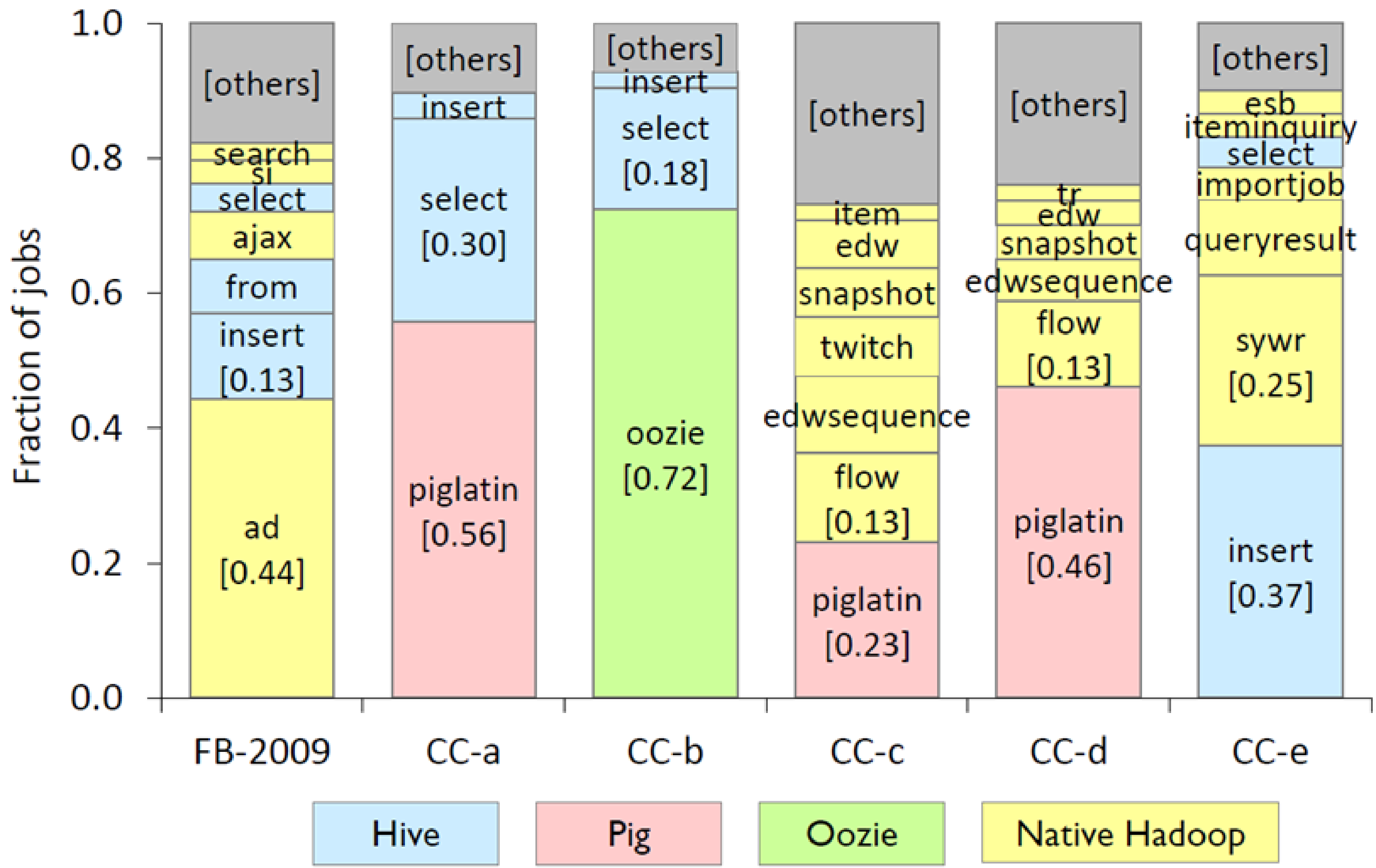
Data Type / Algorithm	SQL | Graph | Machine Learning | …
Execution Engine	MapReduce, Spark, Tez
Cluster Management	Hadoop Yarn
Data Storage	HDFS

Processing Frameworks for Hadoop



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

Hadoop-based Data Analysis Frameworks



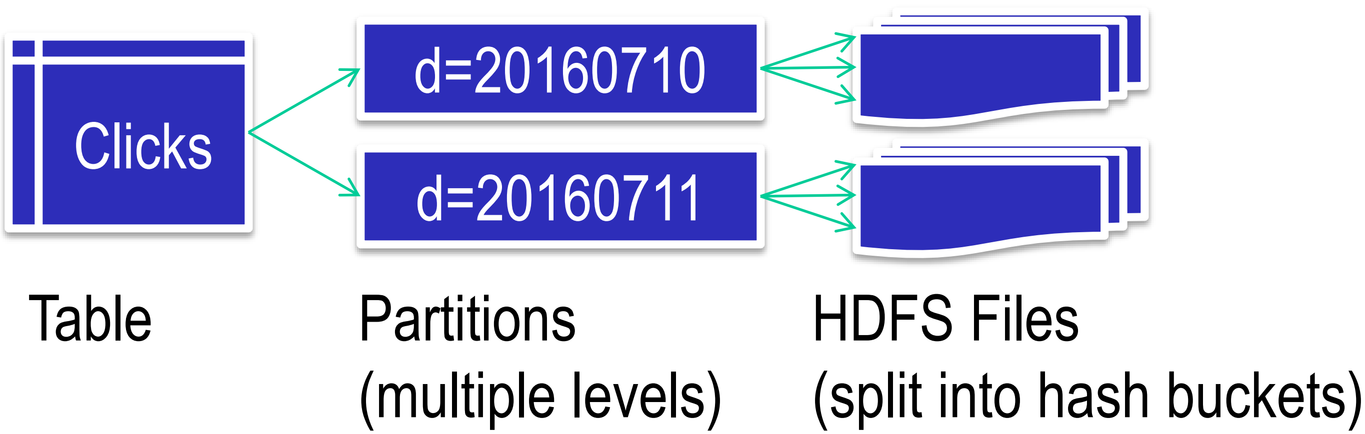
Quelle: Chen et. al: Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. VLDB 2012

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)

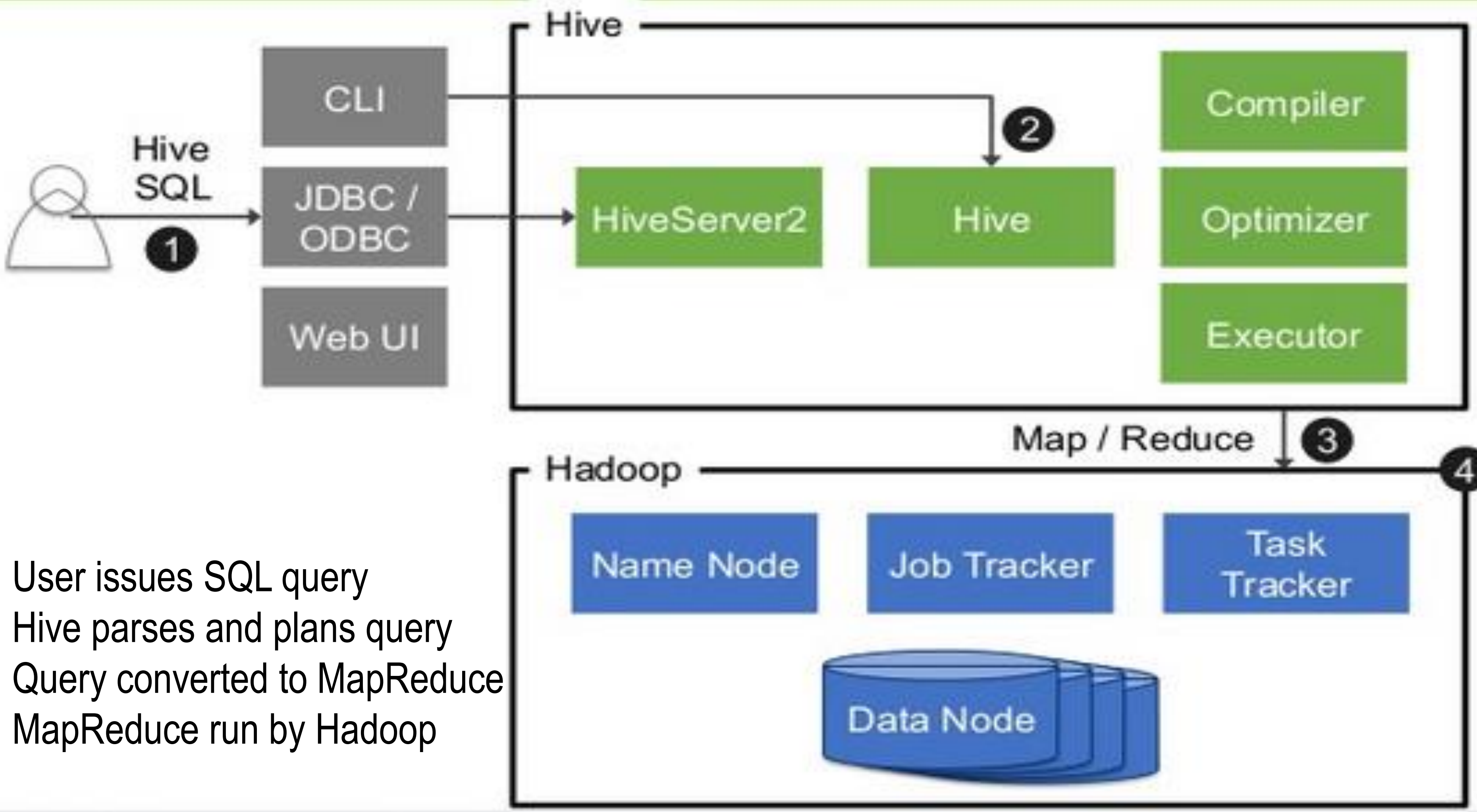
Hive: Metastore

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



- Table 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

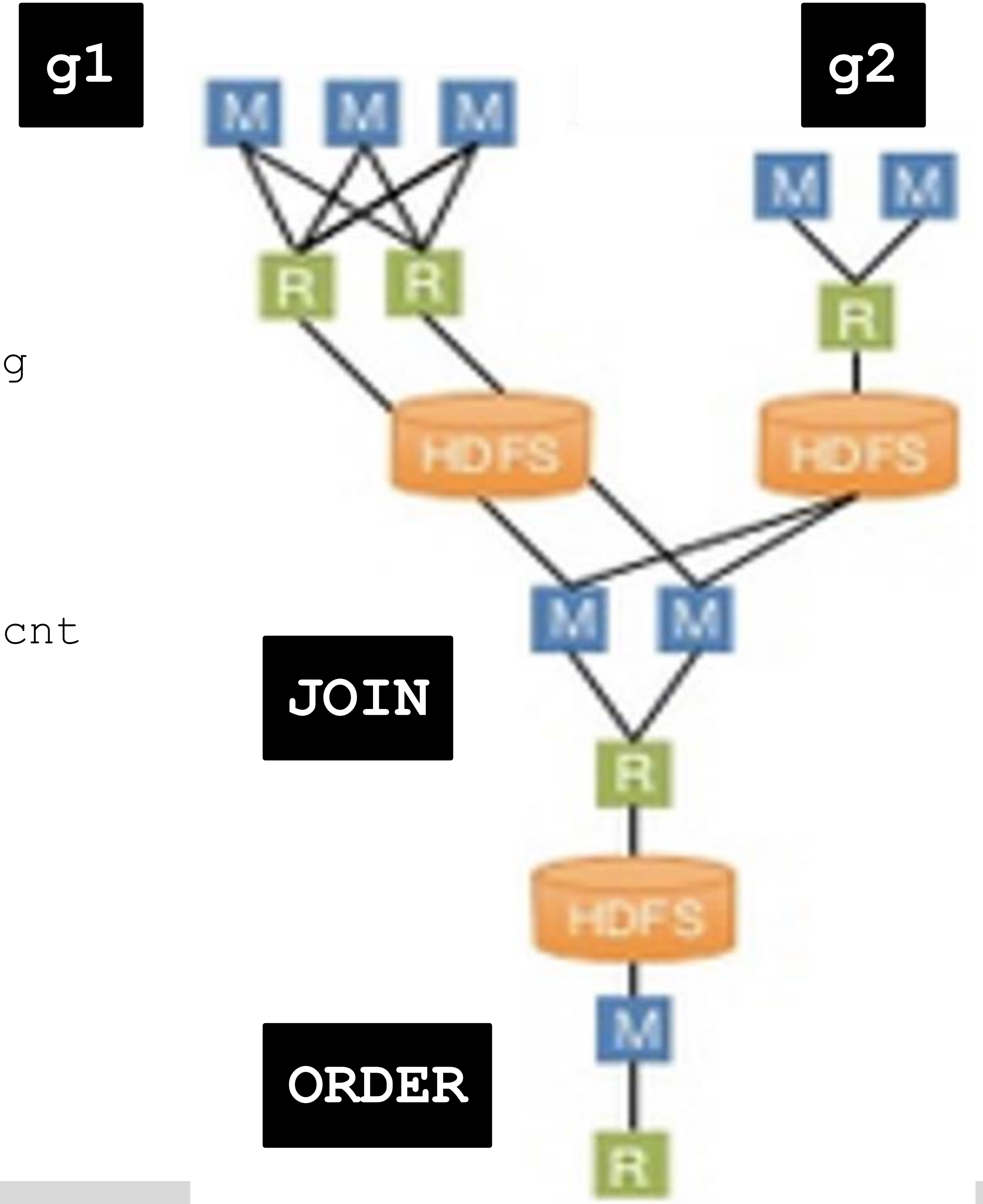


1. User issues SQL query
2. Hive parses and plans query
3. Query converted to MapReduce
4. MapReduce run by Hadoop

Abadi et. al: SQL-on-Hadoop Tutorial. VLDB 2015

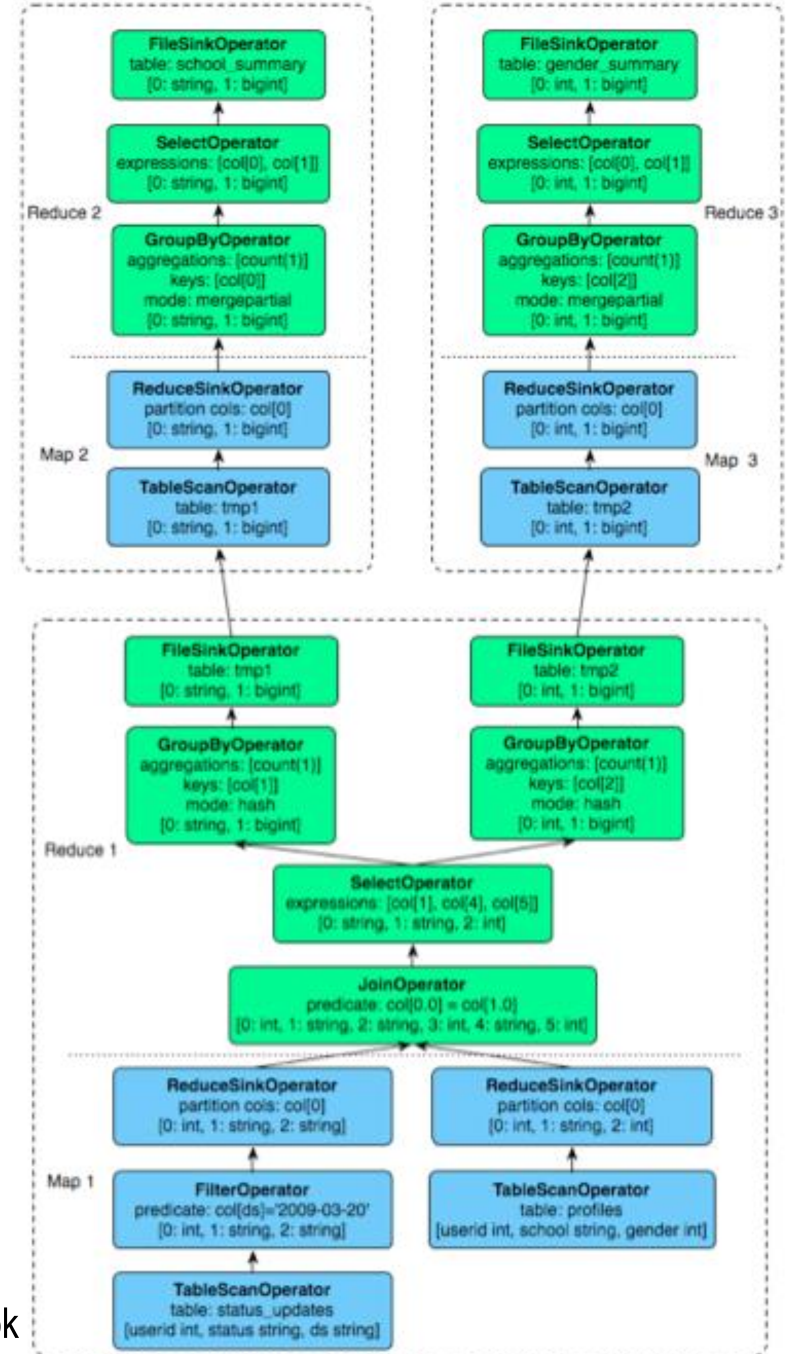
Hive: Query

```
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)

```
{“_id”:”1”, "name":"fish.jpg",”time":”17:46","user":"bob“,"camera":"nikon",
           "info":{"width":100,"height":200,"size":12345},"tags":["tuna","shark"]}
{“_id”:”2”, "name":"trees.jpg",“time":”17:57”,"user":"john”,"camera":"canon",
           "info":{"width":30,"height":250,"size":32091},"tags":["oak"]}
....
```

- Relational: Nested table with multi-valued attributes

id	name	time	user	camera	info			tags
					width	height	size	
1	fish.jpg	17:46	bob	nikon	100	200	12345	[tuna, shark]
2	trees.jpg	17:57	john	canon	30	250	32091	[oak]
3	snow.png	17:56	john	canon	64	64	1253	[tahoe, powder]
4	hawaii.png	17:59	john	nikon	128	64	92834	[maui, tuna]
5	hawaii.gif	17:58	bob	canon	320	128	49287	[maui]
6	island.gif	17:43	zztop	nikon	640	480	50398	[maui]

Source: http://labs.mudynamics.com/wp-content/uploads/2009/04/icouch.html

SQL to MapReduce: Example

```
SELECT camera, AVG(info.size)
FROM Pictures
WHERE user="john"
GROUP BY camera
```

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

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

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

map

key	value
canon	32091
canon	1253
nikon	92834

shuffle + sort

key	values
canon	[32091, 1253]
nikon	[92834]

reduce

(canon,16672)
(nikon,92835)

SQL to MapReduce

SQL	MapReduce
Selection WHERE user = ‘John‘	Filter in map function if (user==‘John‘) { emit (…); }
Projection SELECT camera, size	Map output value emit (…, {camera, size});
Grouping GROUP BY camera	Map output key = grouping attribute(s) emit (camera, …);
Aggregation SELECT AVG (size)	Computation in reduce function average ([size1, size2, …]);
Nested Queries FROM (SELECT … FROM …) AS T	Sequence of MapReduce programs Output of MR1 (inner query)= input to MR2 (outer q.)
Sorting ORDER BY camera	Map output key = sorting attribute(s) Requires single reducer or range partitioner
Join FROM R JOIN S ON (R.b=S.b)	-- see next slides --

Repartition Join (for Equi Join)

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

R

a	b
a_1	3
a_2	3
a_3	3
a_4	2
a_5	3
a_6	2
a_7	2
a_8	2

S

b	c
1	c_1
2	c_2
3	c_3
4	c_4

map

(3, R:a_1)
(3, R:a_2)
(3, R:a_3)
(2, R:a_4)

(3, R:a_5)
(2, R:a_6)
(2, R:a_7)
(2, R:a_8)

(1, S:c_1)
(2, S:c_2)
(3, S:c_3)
(4, S:c_4)

shuffle + sort

(1, [S:c_1])

(2, [R:a_4, S:c_2, R:a_6, R:a_7, R:a_8])

(3, [R:a_1, R:a_2, R:a_3, S:c_3, R:a_5])

(4, [S:c_4])

reduce

a	b	c
a_4	2	c_2
a_6	2	c_2
a_7	2	c_2
a_8	2	c_2
a_1	3	c_3
a_2	3	c_3
a_3	3	c_3
a_5	3	c_3

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) $\rightarrow$ Reduce buffering for S keys only

- Example

Naïve	**Extended Key**
(2, R:a_4)	(2:S, c_2)
(2, S:c_2)	(2:S, c_9) (2:R, a_4)
(2, R:a_6)	(2:R, a_6)
(2, R:a_7)	(2:R, a_7)
(2, R:a_8)	(2:R, a_8)
(2, S:c_9)	

Broadcast Join

- Repartition Join: Large map output
  - All tuples are sorted between map and reduce → high network traffic
- Common scenario: |R| >> |S|
  - Example: Logfile ⋈ 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 → $n \cdot |S|$
  - No transfer of R: map task consumes local map partition
  - Repartition-Join: $|R|+|S|$

R

a	b
a_1	3
a_2	3
a_3	3
a_4	2
a_5	3
a_6	2
a_7	2
a_8	2

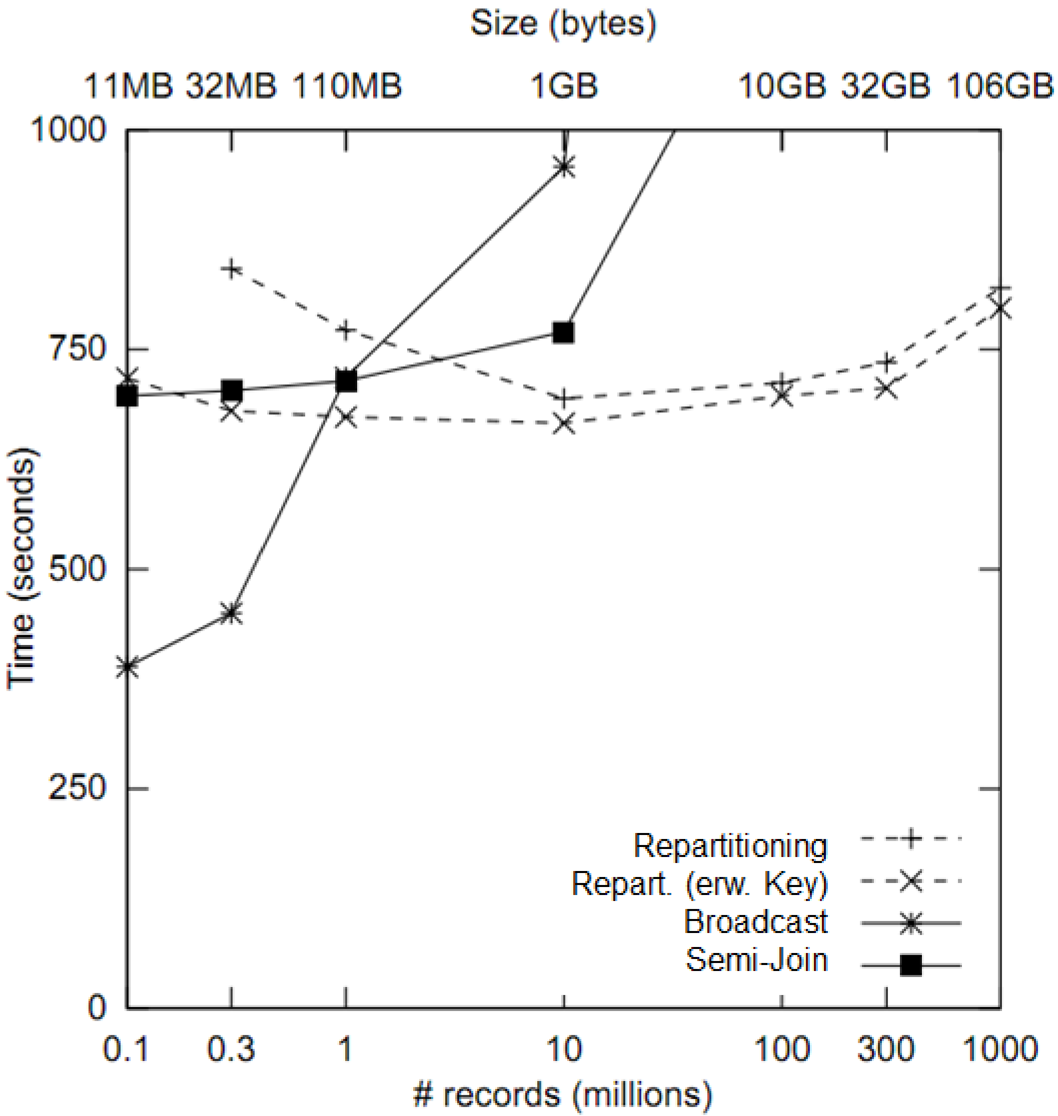
map

a	b	c
a_1	3	c_3
a_2	3	c_3
a_3	3	c_3
a_4	2	c_2
a_5	3	c_3
a_6	2	c_2
a_7	2	c_2
a_8	2	c_2

S

b	c
1	c_1
2	c_2
3	c_3
4	c_4

Evaluation



Data size sent through the network

#record (Relation S)	Repartition (ext. Key)	Broad-cast
$0.3 \cdot 10^6$	145 GB	6 GB
$10 \cdot 10^6$	145 GB	195 GB
$300 \cdot 10^6$	151 GB	6240 GB

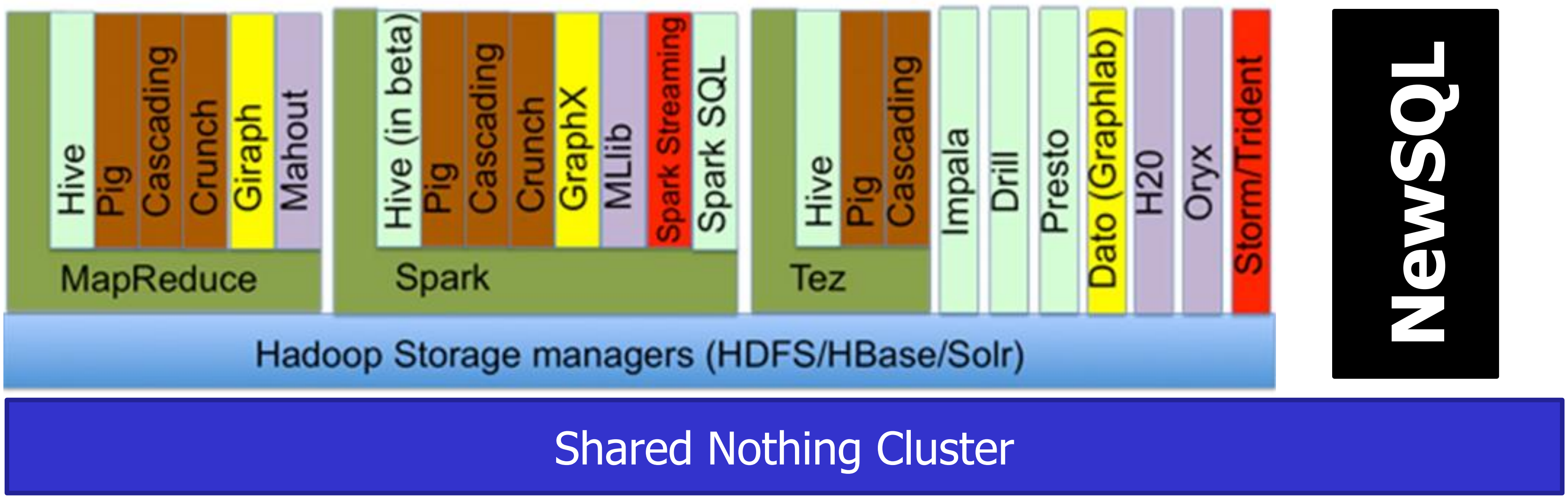
- Prefer broadcast for small S
- Repartitioning: Benefit of extended key

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

SQL on Hadoop

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

From SQL on Hadoop to NewSQL



NewSQL: Definition

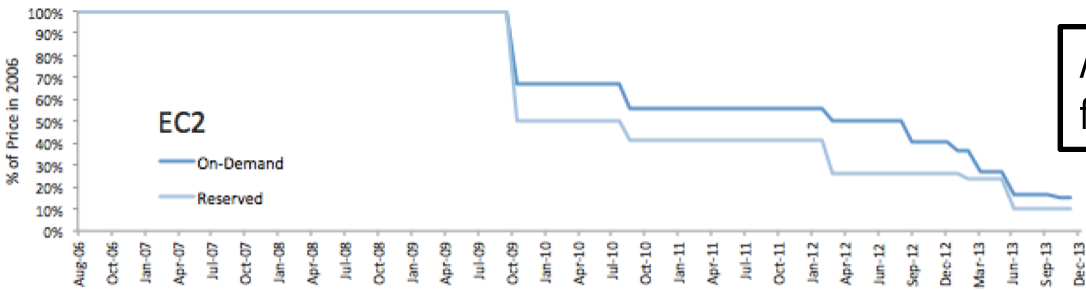
- "… delivers the scalability and flexibility promised by NoSQL while retaining the support for SQL queries and/or ACID, or to improve performance for appropriate workloads." (451 group)

- 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

Matt Asslet, 451 Group, 2011: https://www.451research.com/report-short?entityId=66963
Michael Stonebraker, 2011: http://cacm.acm.org/blogs/blog-cacm/109710

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



Amazon EC2 price history for 1TB main memory

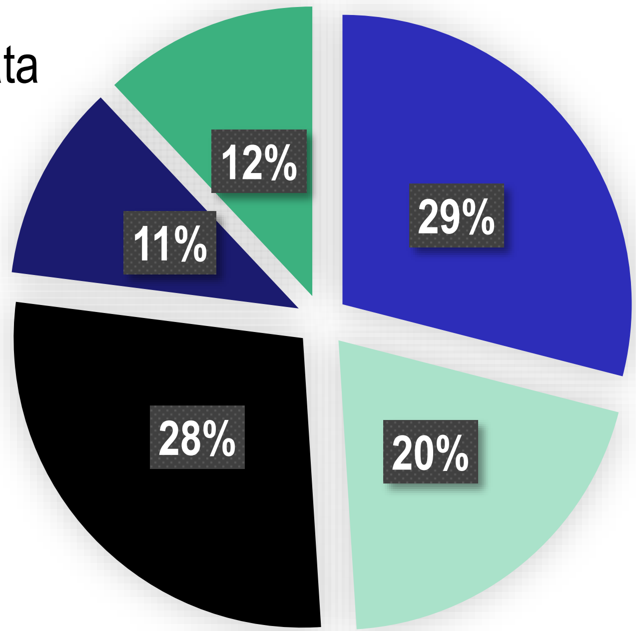
- RDBMS implement Recovery using disk-based Logfiles
  - Cloud / Data Center: Fast recovery via data copying through the network possible
- RDBMS support Multi-Threading (on a single core)
  - T2 can be started if T1 is still waiting for data (from disk) → long transactions should not block short transactions → low latency
  - Cloud / Data Center: Multi core nodes, large main memory

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

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)

Logging
- Write & read log files (write-ahead logging)
- ReDo Recovery (after outage), UnDo Recovery (after transaction failures)

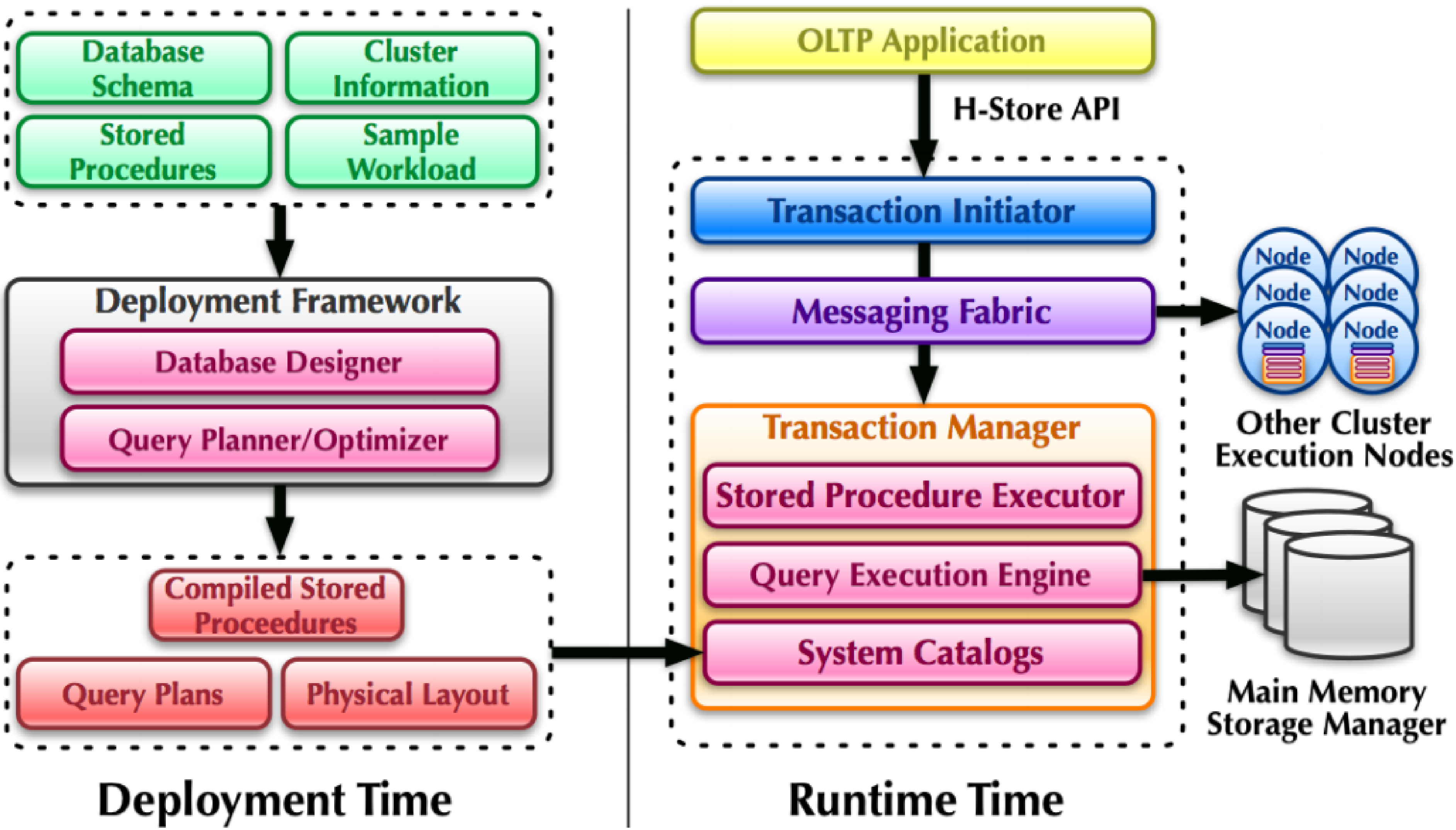
→ ReDo by "Copy from Replica" possible; avoid UnDo cases

Harizopoulos, S. et. al., "OLTP: Through the Looking Glass and What We Found There," SIGMOD, June 2008.

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 $\rightarrow$ no latching
  - Row-store (B-Tree) in main memory $\rightarrow$ 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 $\rightarrow$ strong consistency
  - ACID
  - Direct data access / transfer (no ODBC)
- Recovery
  - Replica-based recovery $\rightarrow$ no logging needed
- VoltDB (commercial) $\approx$ HStore (open source / research prototype)

HStore: Site Architecture



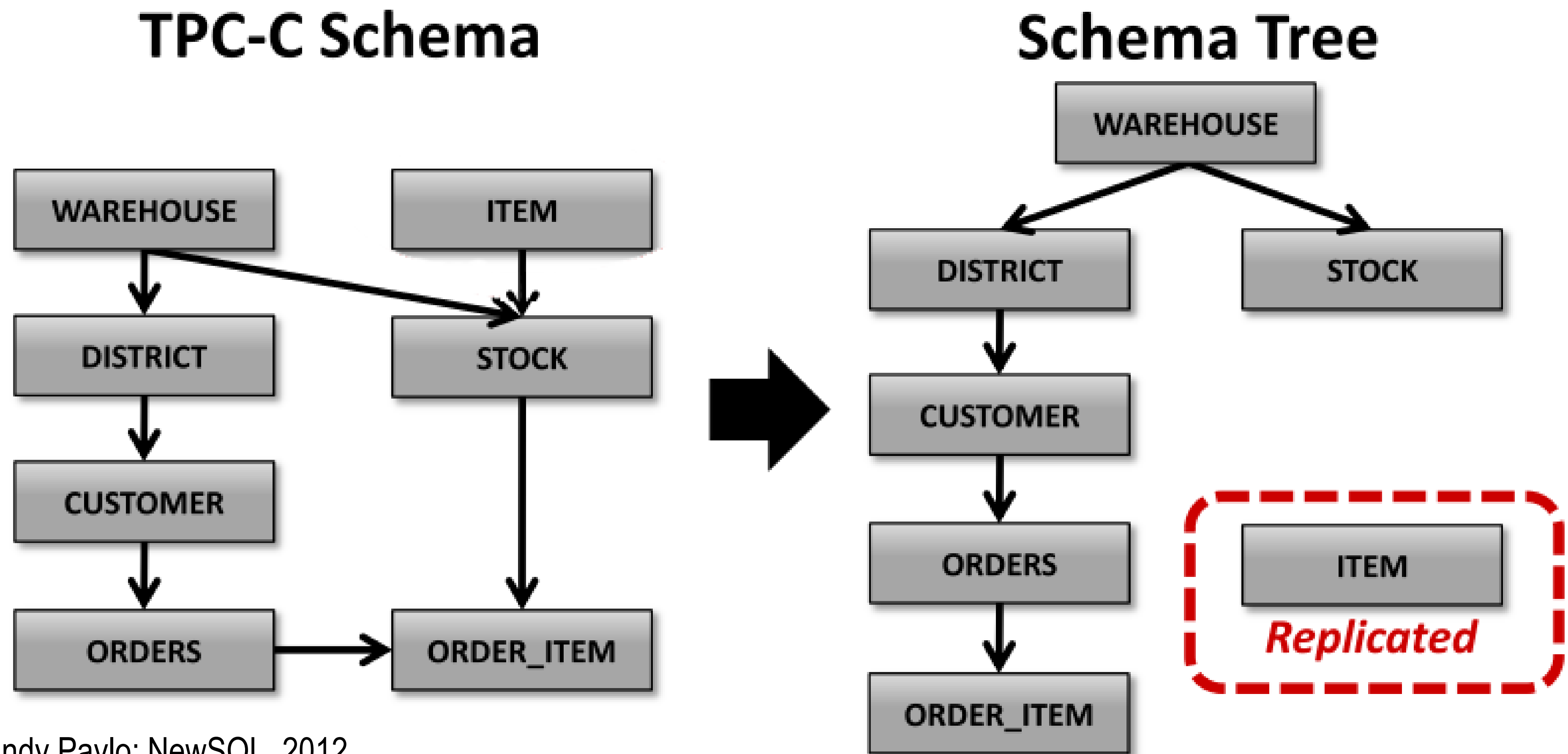
Jones, Abadi, and Madden, "Low overhead concurrency control for partitioned main memory databases,“ SIGMOD 2010

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-may relationships to root table

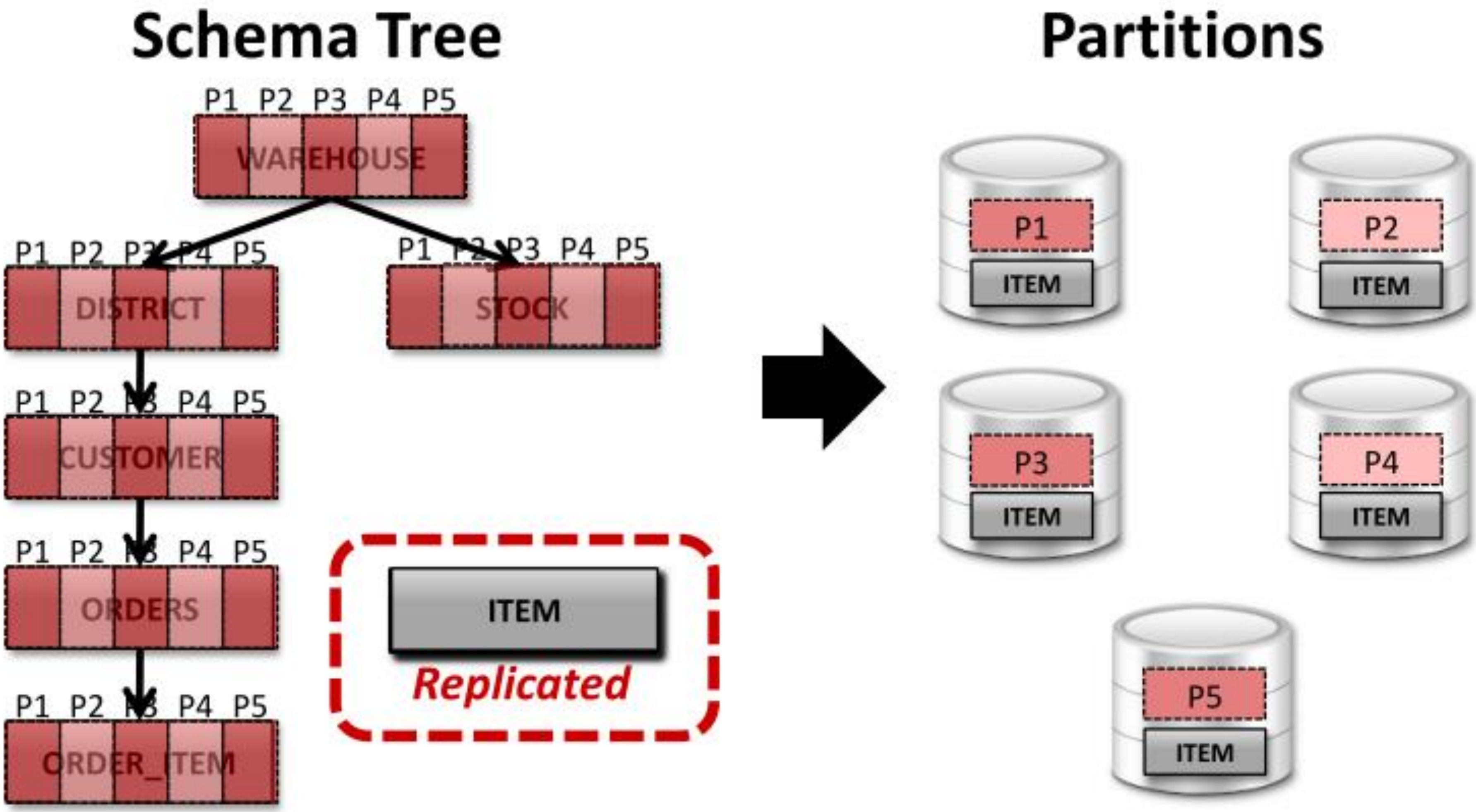


Andy Pavlo: NewSQL, 2012

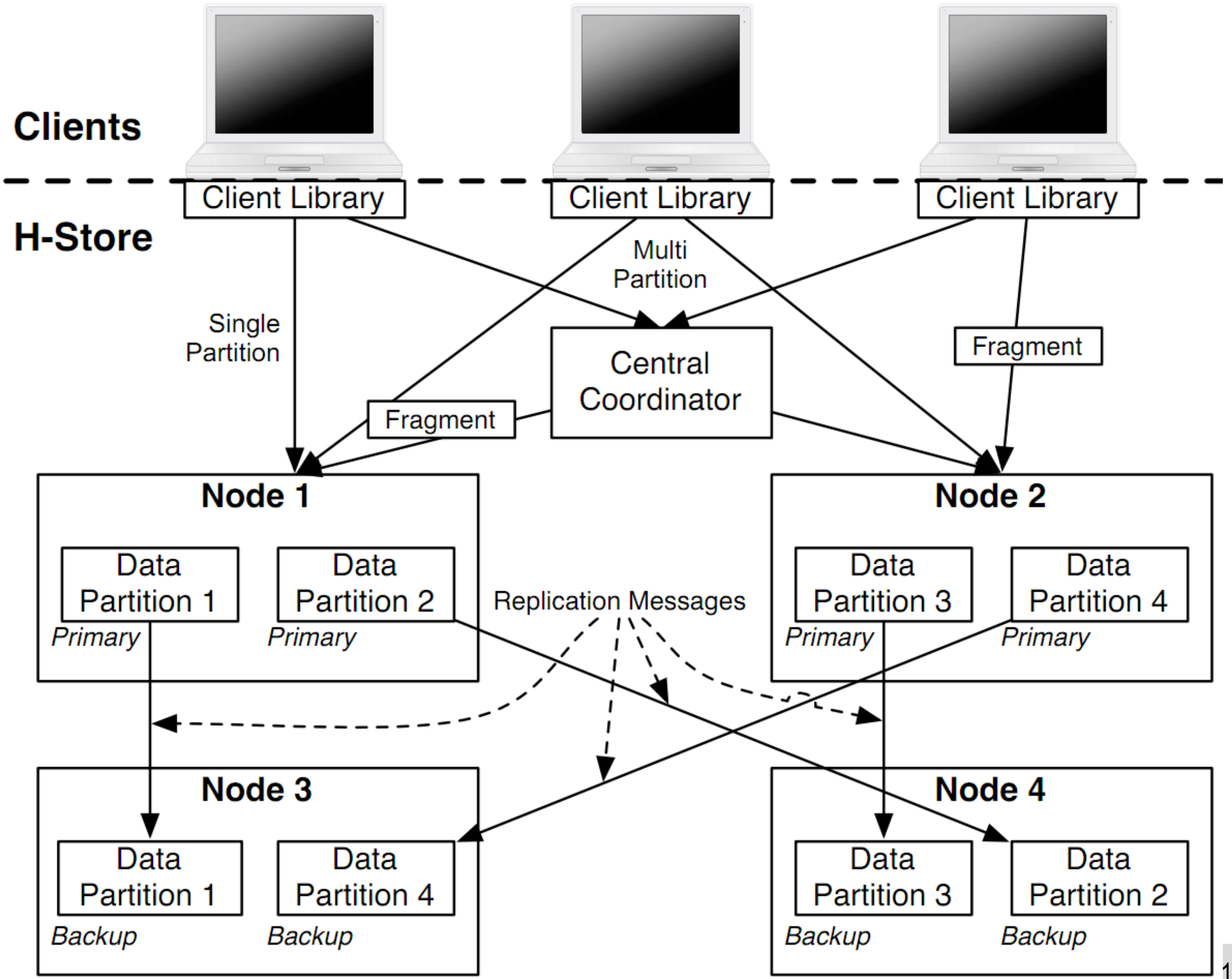
Horizontal Partitioning

Goal: Single-Partition Transactions

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



HStore: Infrastructure



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

```
x=read(a)
y=read(b)

y ≥ 100 ?

write(a, x+100)
write(b, y-100)
```

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

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

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!