CumuloNimbo
Final Review Meeting
Brussels, Belgium
November 27, 2013

FP7-257993

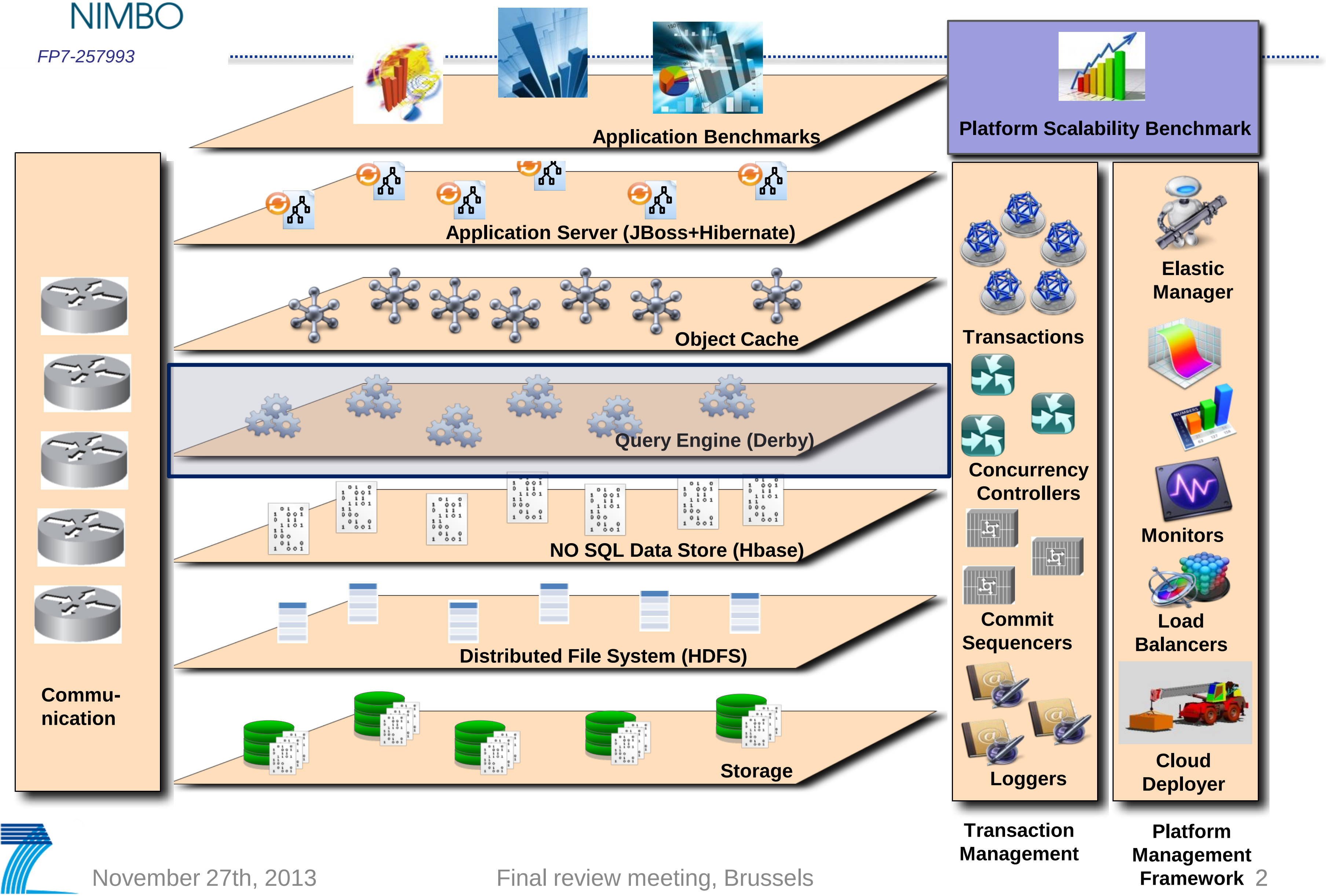
Query Engine

José Pereira, U. Minho

SEVENTH FRAMEWORK PROGRAMME

Positioning with Respect to the Project

FP7-257993



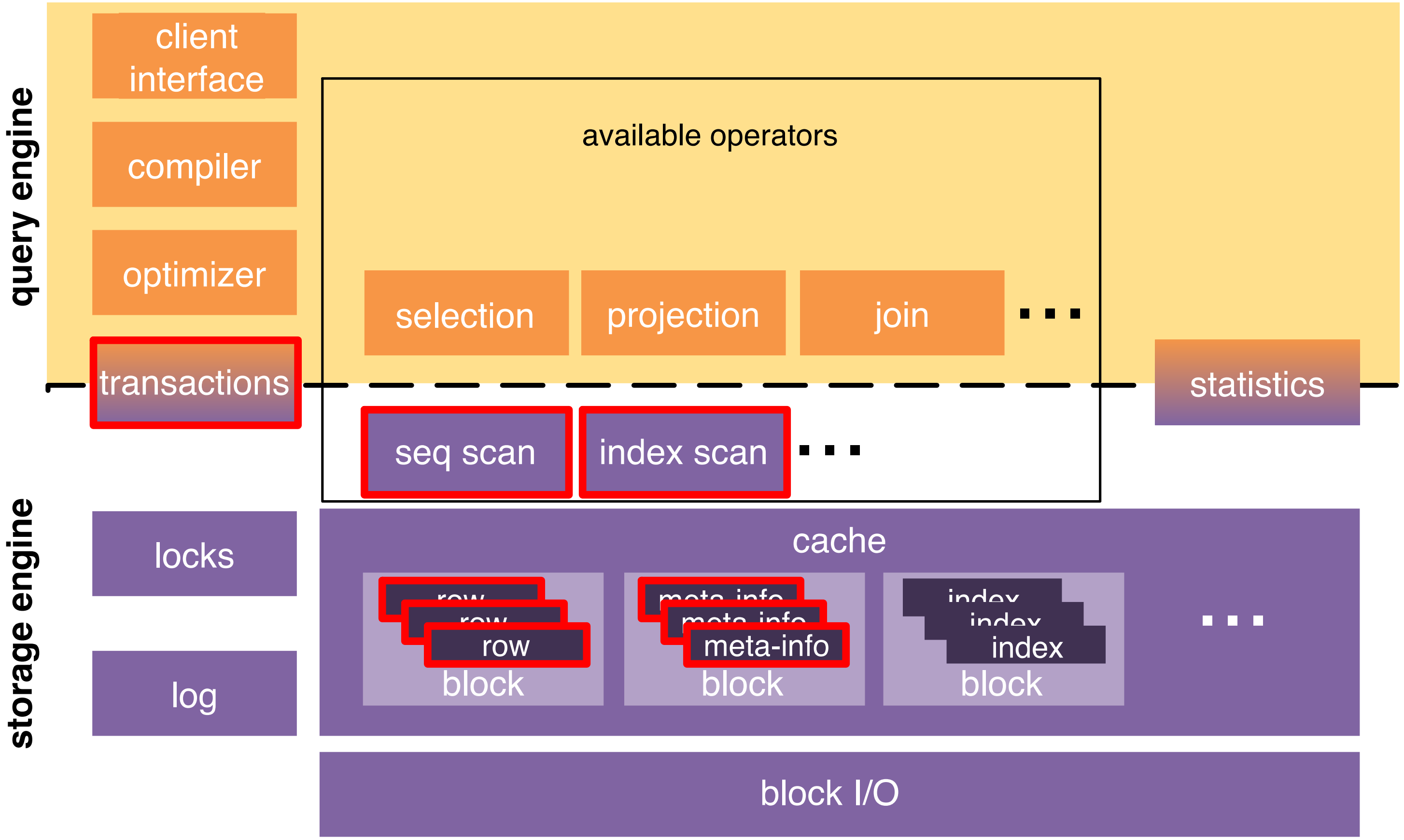
Advancement with Respect to SOTA

- Attaining scalable SQL processing on top of a NoSQL data store
  - A distributed query engine architecture that does not introduce coordination bottlenecks
- Distillation of a query engine functionality out of an open source RDBMS
  - Separation of concerns: execution, transactions, and storage

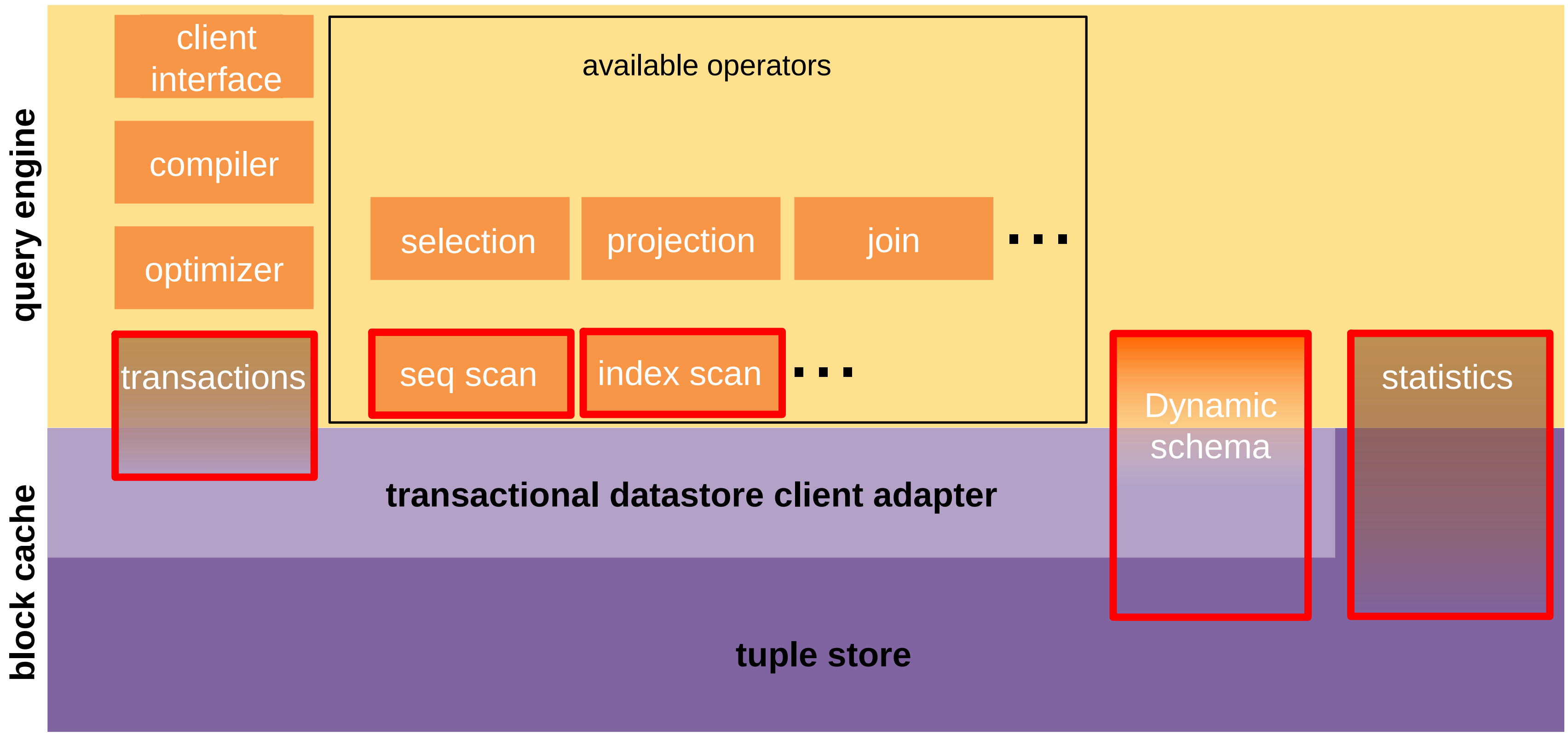
Overview

- General approach to the Query Engine
- Selection of the Query Engine codebase
- Data manipulation (DML) implementation and performance
  - SELECT, INSERT, UPDATE, DELETE
- Data definition (DDL) implementation
  - CREATE/ALTER TABLE, ...
- Monitoring and platform integration

Generic SQL DBMS architecture



Query Engine architecture



Query Engine selection criteria

- Issues
  - Generation of adequate plans
  - Plan selection cost
  - Step execution overhead
  - Optimizer customizability
  - Minimal code intrusion
  - Component interoperability (e.g., communication)
- Non-issues
  - Isolation (replaced)
  - Implementations of indexes (replaced)
  - Effectiveness of caching and block I/O (replaced)

Query Engine selection: Apache Derby

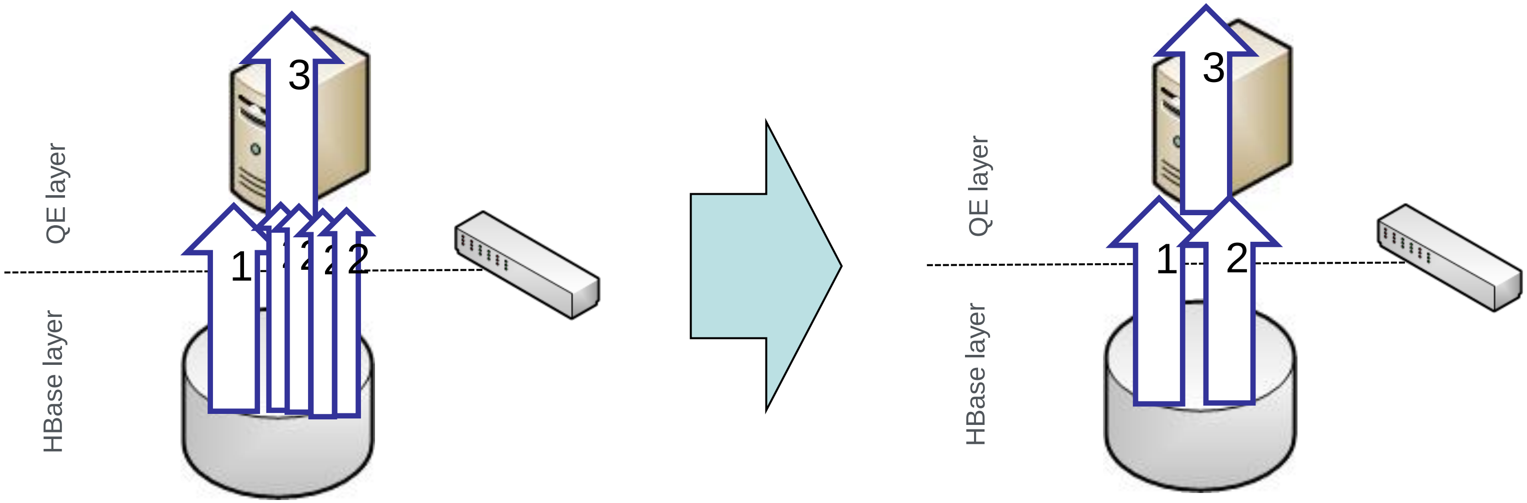
- Selection of relevant queries (sources: SPECj and TPC-E)
- Analysis of the query plans generated:
  - plan itself
  - selected operators
  - compilation and optimization time
- For the target workload, the derived plan is similar to PostgreSQL and differs slightly from MonetDB
- Apache Derby is slower when planning queries, but query caching mitigates this issue.
- Apache Derby facilitates integration with other components

Implementation issues

- Mapping of relational to HBase column-oriented model
  - Usage of a single column family in each table, indexed on the primary key
  - Data types and conversions:
    - Order preservation of the primary key byte encoding
    - Relational NULL as absent columns in HBase
  - Secondary indexes stored as additional HBase tables
- Identification and redirection of relevant internal interfaces
  - Scan operators
  - Transactional interfaces

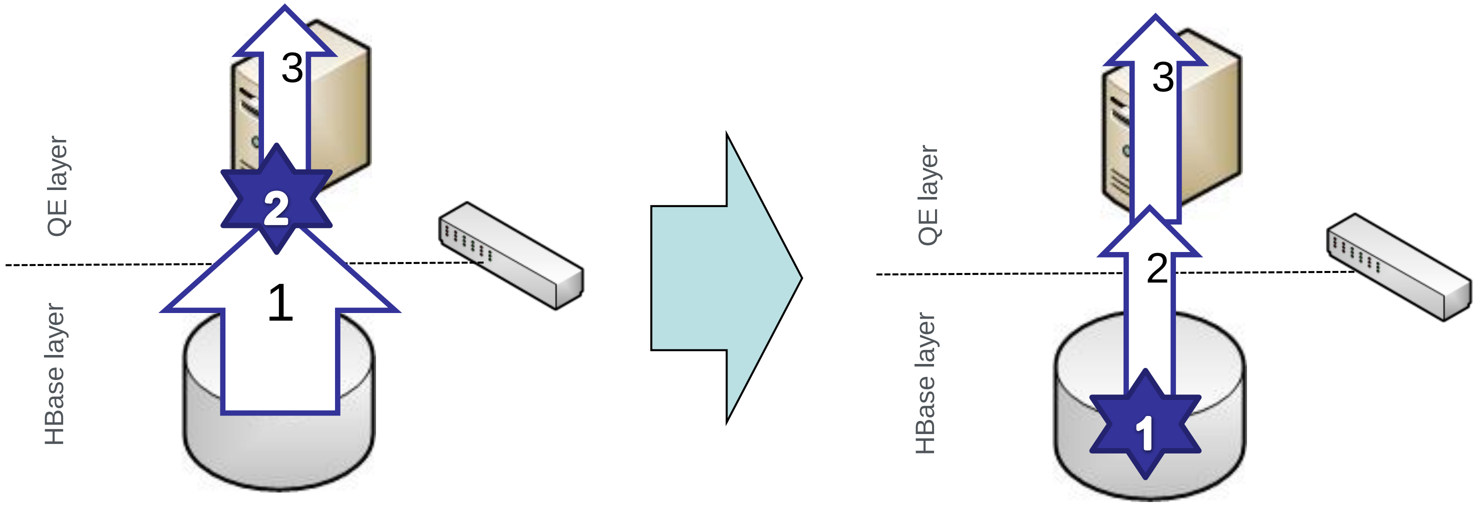
Performance improvements: reducing network traffic (indexed)

- Fetching data through secondary indexes:
  - Challenge: Due to encapsulation, when doing row fetches, the information about the scan was no longer available
  - Direct implementation would use 1 scan on the index + N individual fetches on the table
  - Solution: to propagate additional information in Derby. It now uses 1 scan on the index + 1 bulk fetch on the table



Performance improvements: reducing network traffic (not indexed)

- HBase selection pushed down into Hbase:
  - Challenge: Filtering at the scan operator level does not avoid network traffic
  - Solution: Individual conditions in Derby are translated using the HBase's SingleColumnValueFilter
  - Combine tests on single columns using the HBase's FilterList

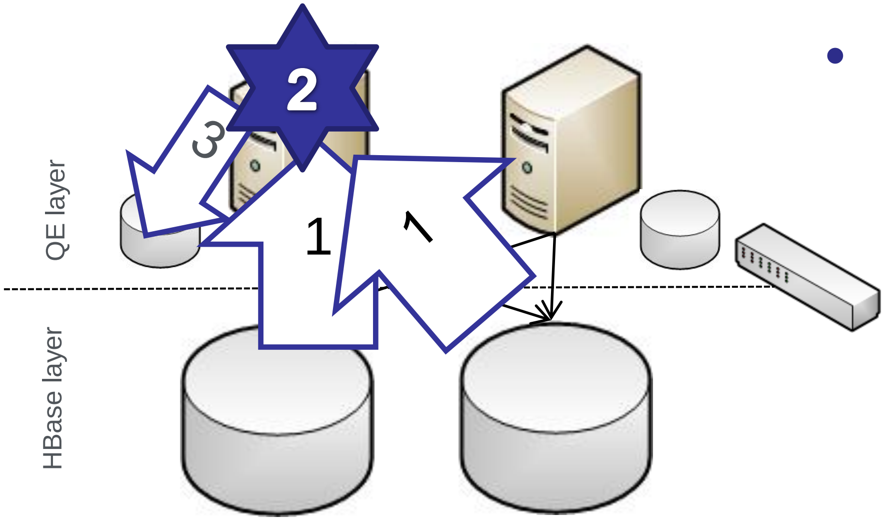


Performance improvements: optimizer

- Optimizing to the integrated stack:
  - Takes advantage of encapsulation in the computation of costs for each operator
  - Scan cost computation considering HBase operations used for implementing them
  - Batch sizes and weights selected using a calibration database

Performance improvements: statistics computation

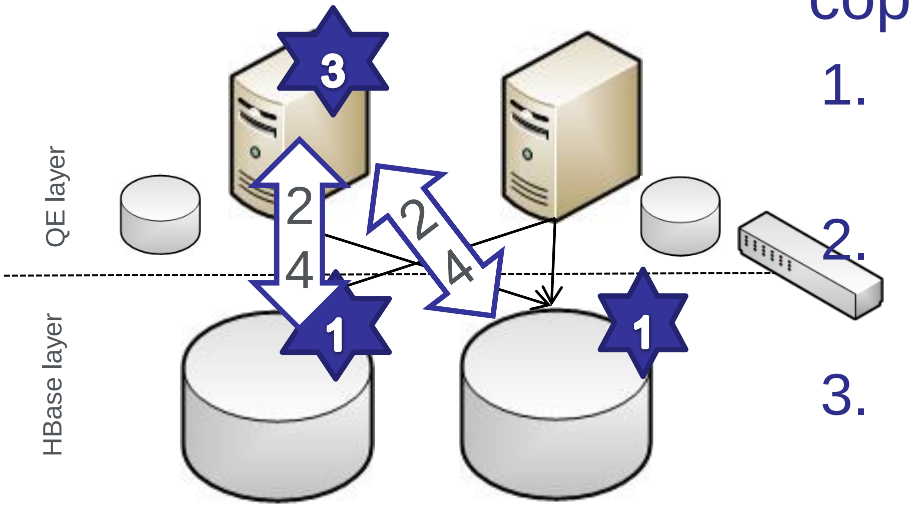
- Derby’s optimize uses cardinality statistics: The number of unique values in the index keys (primary and secondary)
- Computed using a table scan on the index
- Results stored in a system table (SYSSTATISTICS).



- What happens after simple adaptation of QE:
  1. Read table into each instance
  2. Compute statistics
  3. Write back to local storage

FP7-257993

Performance improvements: statistics computation



- To limit the network traffic and exploit parallelism, we use HBase coprocessors:
  1. The coprocessor is loaded into all tables
  2. Each coprocessor returns partial (region) results
  3. A QE instance then merges all partial results
  4. Stores them back in a shared HBase table

Dynamic DDL

- Schema and Data Definition Language (DDL) in Derby:
  - Application schemas stored in relational system tables
  - System tables residing in the SYS namespace
  - Accessed using the DataDictionary interface, that caches the schema as native objects
- Contents:
  - Row counts
  - Columns and types

Dynamic DDL

- Schema storage in HBase
  - Schema is stored in regular tables but cached (slightly different access paths)
  - A single copy is shared by all QE instances
  - Local caches have to be kept consistent
- Lazy replication of row counts, changed by DML statements:
  - Batched using atomic operations
  - Re-read periodically

Dynamic DDL

- Provides support for:
  - CREATE/DROP TABLE
  - ALTER/ADD and ALTER/DROP COLUMN
- Assumption:
  - All transactions containing DDL statements must be declared as DDL transactions

Dynamic DDL

- Multi-versioned database schemas:
  - Made possible by flexible mapping between relational and HBase schemas
  - Required multiple concurrent active DataDictionaries in a QE
- Conflicting DDL statements prevented by transaction manager as a write-write conflict on SYS tables
- Notification through Zookeeper to update schema

Integration support

- HBase table and column names matching relational table, index, and column names
  - Needed for interoperability with NoSQL applications
  - Useful for debugging
- Simpler configuration with properties for debugging and logging
- Unit and integration tests:
  - Include subset of Apache Derby tests
  - Added tests for new functionality

Performance monitoring

- Statistics required for elastic management:
  - Average operation latency
  - Average operation size
- Operations recorded:
  - HBase and transactional primitives
- Results saved in Zookeeper:
  - /monitoring/queryengine/*instance_id*
  - Summary statistics and an histogram
  - Total and for the last measurement period

Results and contributions

- A logical architecture leveraging a sub-set of components that are commonly found within a traditional RDBMS
- Decision criteria to select an existing implementation as the base for CumuloNimbo's Query Engine, with an analysis of several candidates
- A prototype that validates the approach based on Derby and supporting stateless handling of DML statements
- Multiple performance optimizations, reducing network traffic and taking advantage of distributed computation in HBase
- Relational optimizer tuned to the proposed architecture
- Dynamic handling of DDL statements with multiversion schema
- Performance monitoring hooks
- Integration in the CumuloNimbo stack