Impala: A Modern, Open-Source SQL Engine for Hadoop

Marcel Kornacker | marcel@cloudera.com

CIDR 2015

Cloudera Impala — Agenda

- **Overview**
- Architecture and Implementation
- Evaluation

Impala: A Modern, Open-Source SQL Engine

- Implementation of an MPP SQL query engine for the Hadoop environment
- Designed for performance: brand-new engine, written in C++
- Maintains Hadoop flexibility by utilizing standard Hadoop components (HDFS, Hbase, Metastore, Yarn)
  - Reads widely used Hadoop file formats (e.g. Parquet, Avro, RC, …)
  - Runs on same nodes that run Hadoop processes
- Plays well with traditional BI tools: exposes/interacts with industry-standard interfaces (odbc/jdbc, Kerberos and LDAP, ANSI SQL)

Impala from The User's Perspective

- Create tables as virtual views over data stored in HDFS or Hbase
- Schema metadata stored in Metastore, basis of HCatalog
  - Shared and can be accessed by Hive, Pig, etc..
- Connect via ODBC/JDBC; authenticate via Kerberos or LDAP
- ANSI SQL-92 with SQL-2003 analytic window functions, UDFs/UDAs, correlated subqueries,..
- Data types:
  - Integer and floating point type, STRING, CHAR, VARCHAR, TIMESTAMP
  - DECIMAL(<precision>, <scale>) up to 38 digits of precision

Impala: History

- Developed by Cloudera and fully open-source (ASF license)
  - Hosted on github (https://github.com/cloudera/impala)
- Released as beta in 10/2012
- 1.0 version available in 05/2013
- current version: 2.1

Roadmap: Impala 2.1+

- Nested data structures: Structs, arrays, maps in Parquet, Avro, json, …
  - natural extension of SQL: expose nested structures as tables
  - no limitation on nesting levels or number of nested fields in single query
- Multithreaded execution past scan operator
- More resource management and admission control
- Support for S3-backed tables
- Additional data types: DATE, TIME, DATETIME
- More SQL: ROLLUP/GROUPING SETS, INTERSECT/MINUS, MERGE
- Improved query planning, more elaborate statistics
- Physical tuning

Cloudera Impala — Agenda

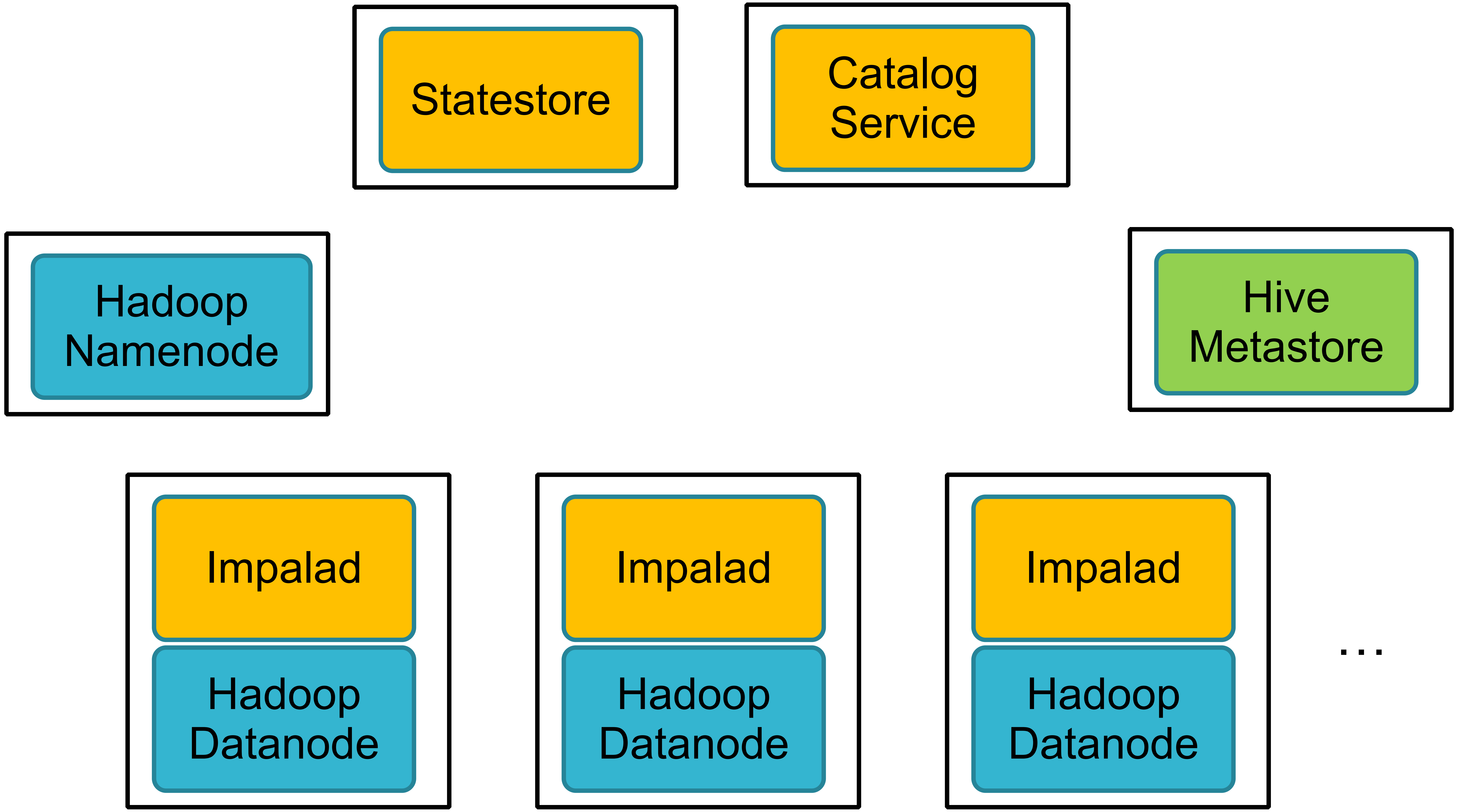
- Overview
- **Architecture and Implementation**
  - High-level design
  - Components
  - Query Planning
  - Query Execution
  - Run-time Code Generation
  - Parquet File Format
- Evaluation

Impala Architecture: Distributed System

- Daemon process (impalad) runs on every node with data
- Each node can handle user requests
  - Load balancer configuration for multi-user environments recommended
- Metadata management: catalog service (single node)
- System state repository and distribution: statestore (single node)
- Catalog service and Statestore are stateless

cloudera
Ask Bigger Questions

Impala Architecture



Impala Statestore

- Central system state repository
  - name service (membership)
  - metadata
- Soft-state
  - all data can be reconstructed from the rest of the system
  - cluster continues to function when statestore fails, but per-node state becomes increasingly stale
- Sends periodic heartbeats
  - pushes new data
  - checks for liveness

Impala Catalog Service

- Metadata:
  - databases, tables, views, columns, …
  - but also: files, block replica locations, block device ids
- Catalog service:
  - metadata distribution hub: sends all metadata to all impalad’s via statestore
  - interface to persistent metadata storage, mediator between Hive’s MetaStore and impala’s

Impala Execution Daemon (impalad)

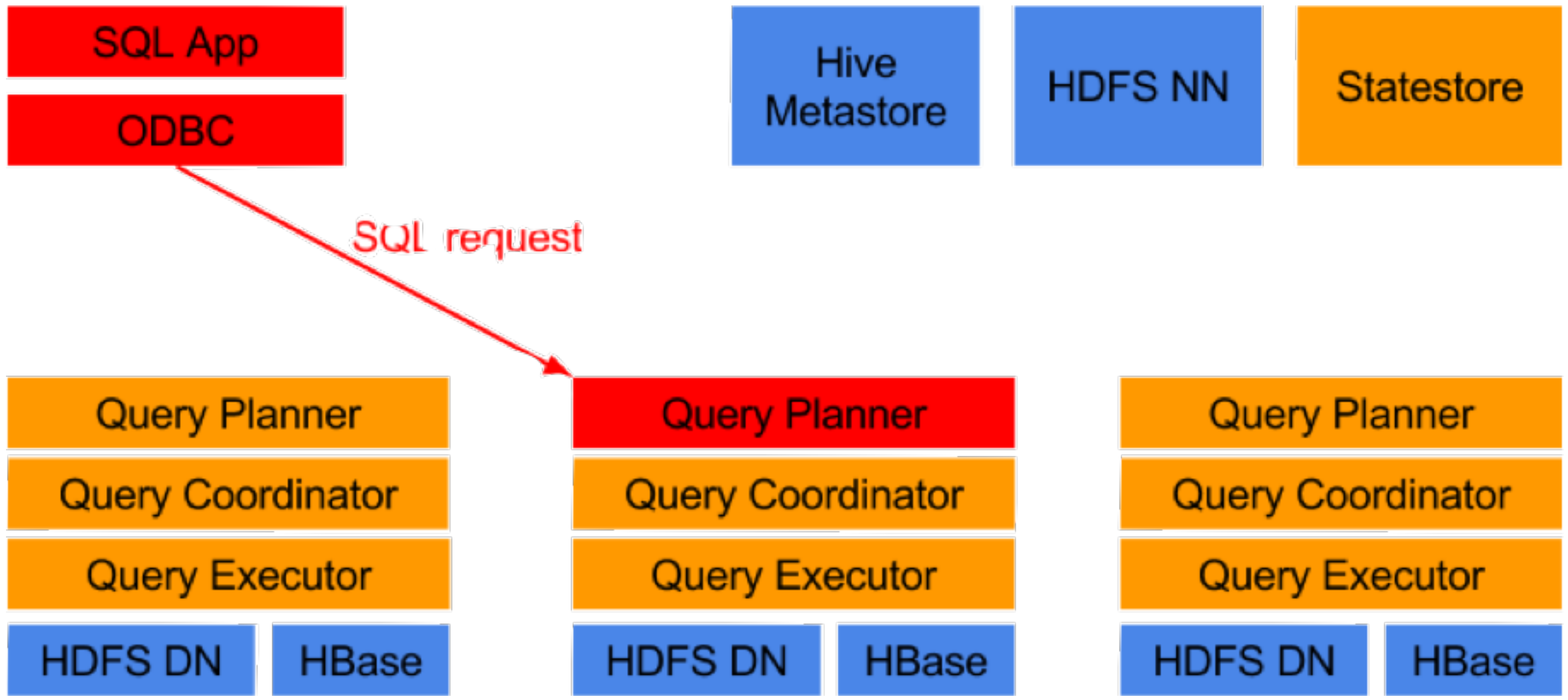
- Frontend in Java: parse, analyze and plan SQL queries
- Backend in C++: coordinate and/or execute plan fragments
- Local cache of metadata
- Web UI with machine info, logs, metrics
- RPC/communication: Thrift

Impala Query Execution at the high-level

- Query execution phases:
  - Client request arrives via odbc/jdbc
  - Query planner turns request into collection of plan fragments
  - Coordinator initiates execution on remote impalad's
- During execution
  - Intermediate results are streamed between executors
  - Query results are streamed back to client
  - Subject to limitations imposed by blocking operators
    - top-n, aggregation, sorting

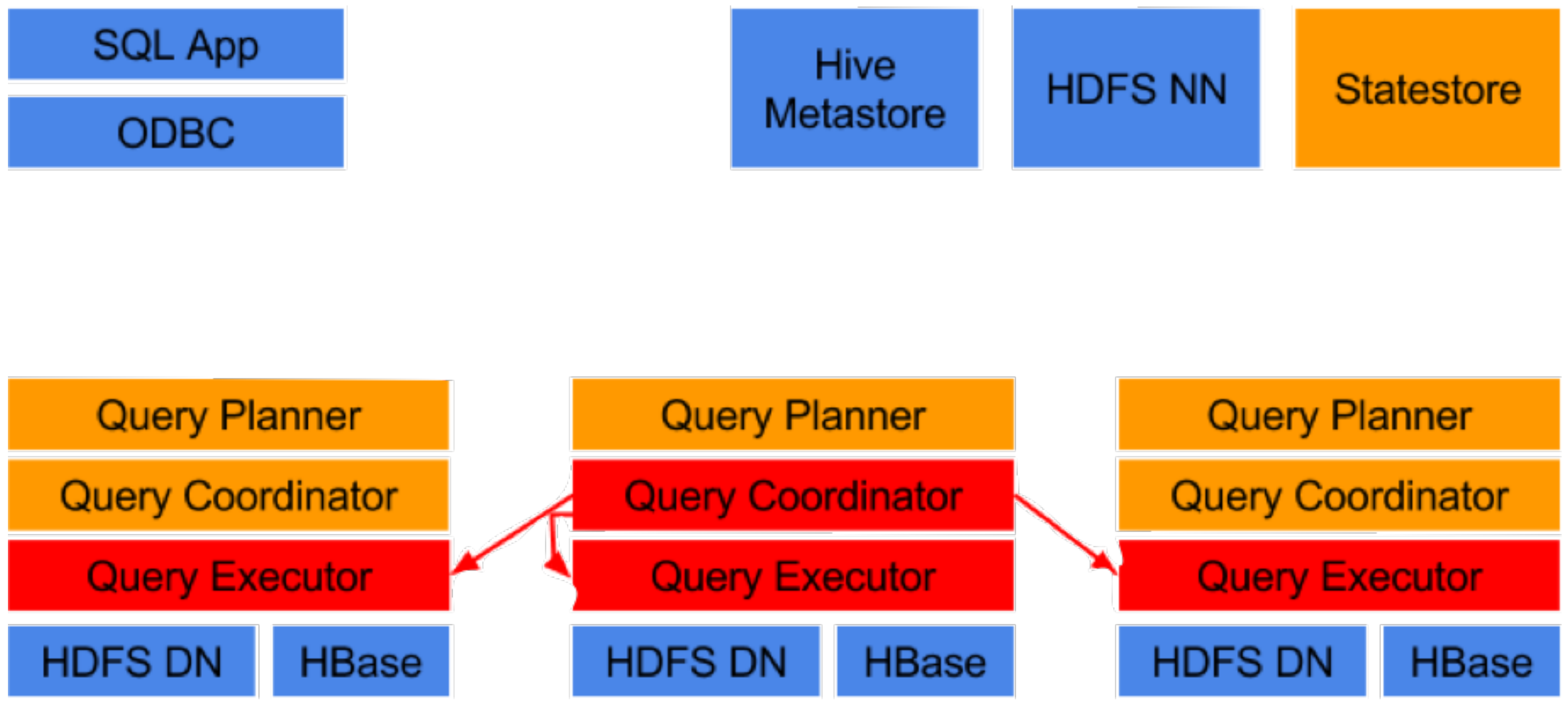
Impala Query Execution

- Request arrives via odbc/jdbc



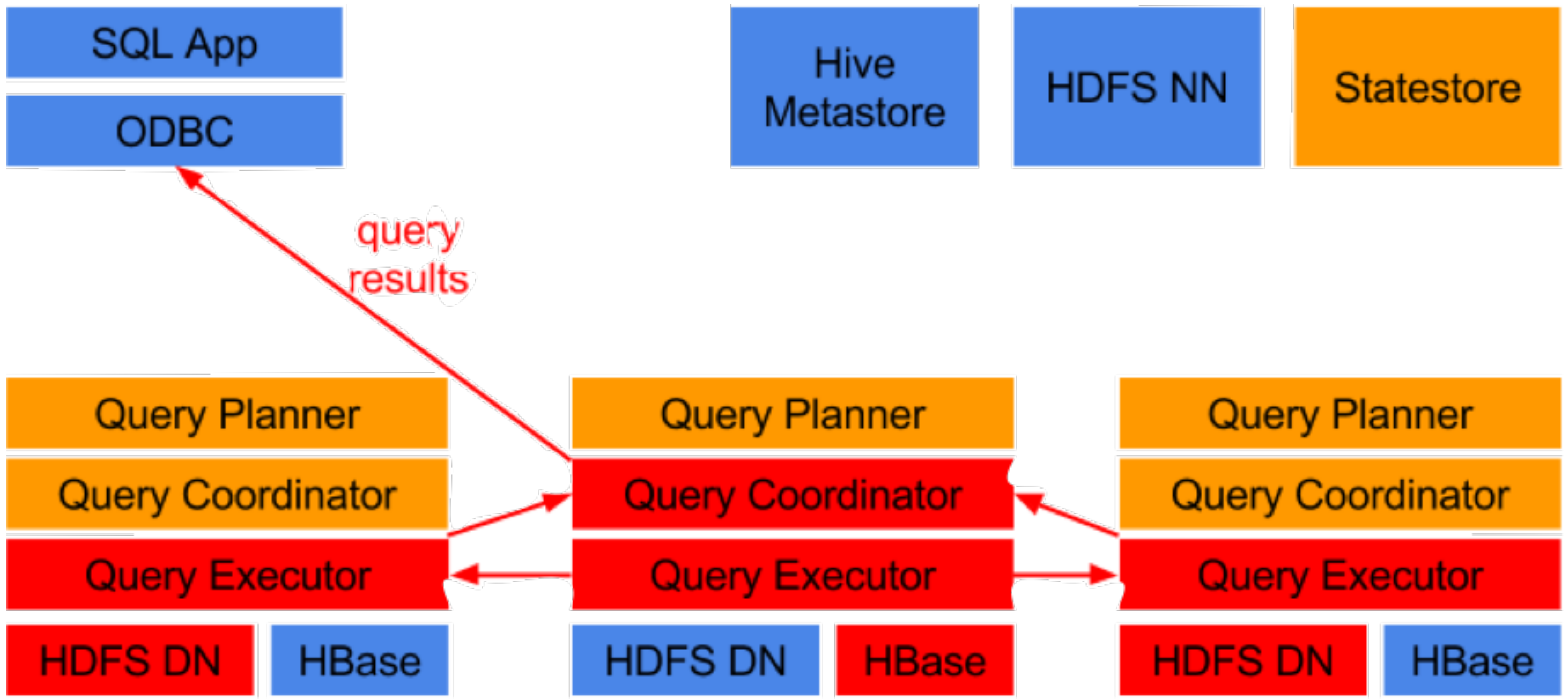
Impala Query Execution

- Planner turns request into collection of plan fragments
- Coordinator initiates execution on remote impalad nodes



Impala Query Execution

- Intermediate results are streamed between impalad's
- Query results are streamed back to client

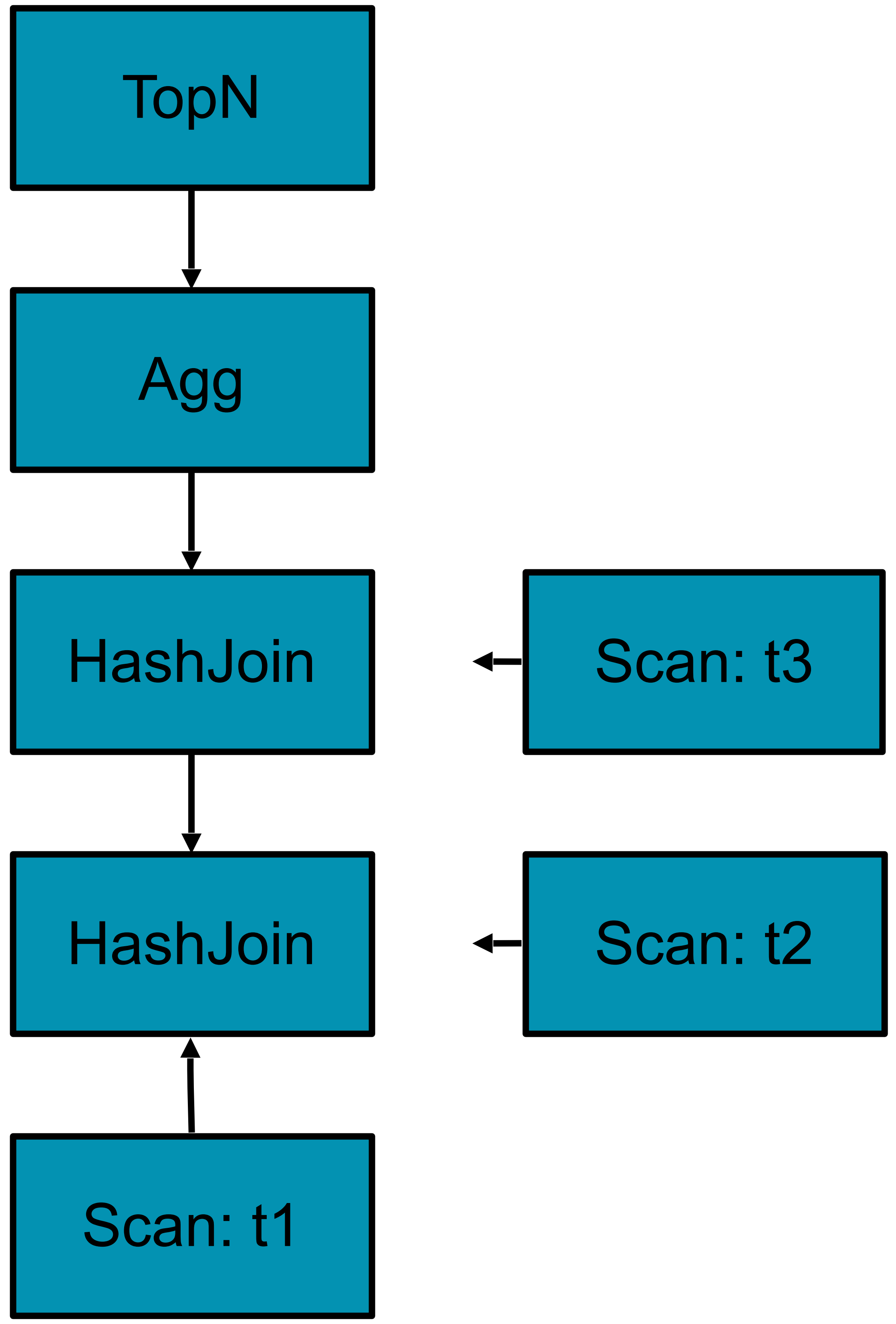


Query Planning: Overview

- 2-phase planning process:
  - single-node plan: left-deep tree of plan operators
  - partitioning of operator tree into plan fragments for parallel execution
- Parallelization of operators across nodes:
  - all query operators are fully distributed
- Cost-based join order optimization
- Cost-based join distribution optimization

Query Planning: Single-Node Plan

```
SELECT t1.custid,
    SUM(t2.revenue) AS revenue
FROM LargeHdfsTable t1
JOIN LargeHdfsTable t2 ON (t1.id1 = t2.id)
JOIN SmallHbaseTable t3 ON (t1.id2 = t3.id)
WHERE t3.category = 'Online'
GROUP BY t1.custid
ORDER BY revenue DESC LIMIT 10
```



Query Planning: Distributed Plans

- Goals:
  - maximize scan locality, minimize data movement
  - full distribution of all query operators (where semantically correct)
- Parallel joins:
  - broadcast join: join is collocated with left input; right-hand side table is broadcast to each node executing join
    -> preferred for small right-hand side input
  - partitioned join: both tables are hash-partitioned on join columns
    -> preferred for large joins
  - cost-based decision based on column stats/estimated cost of data transfers

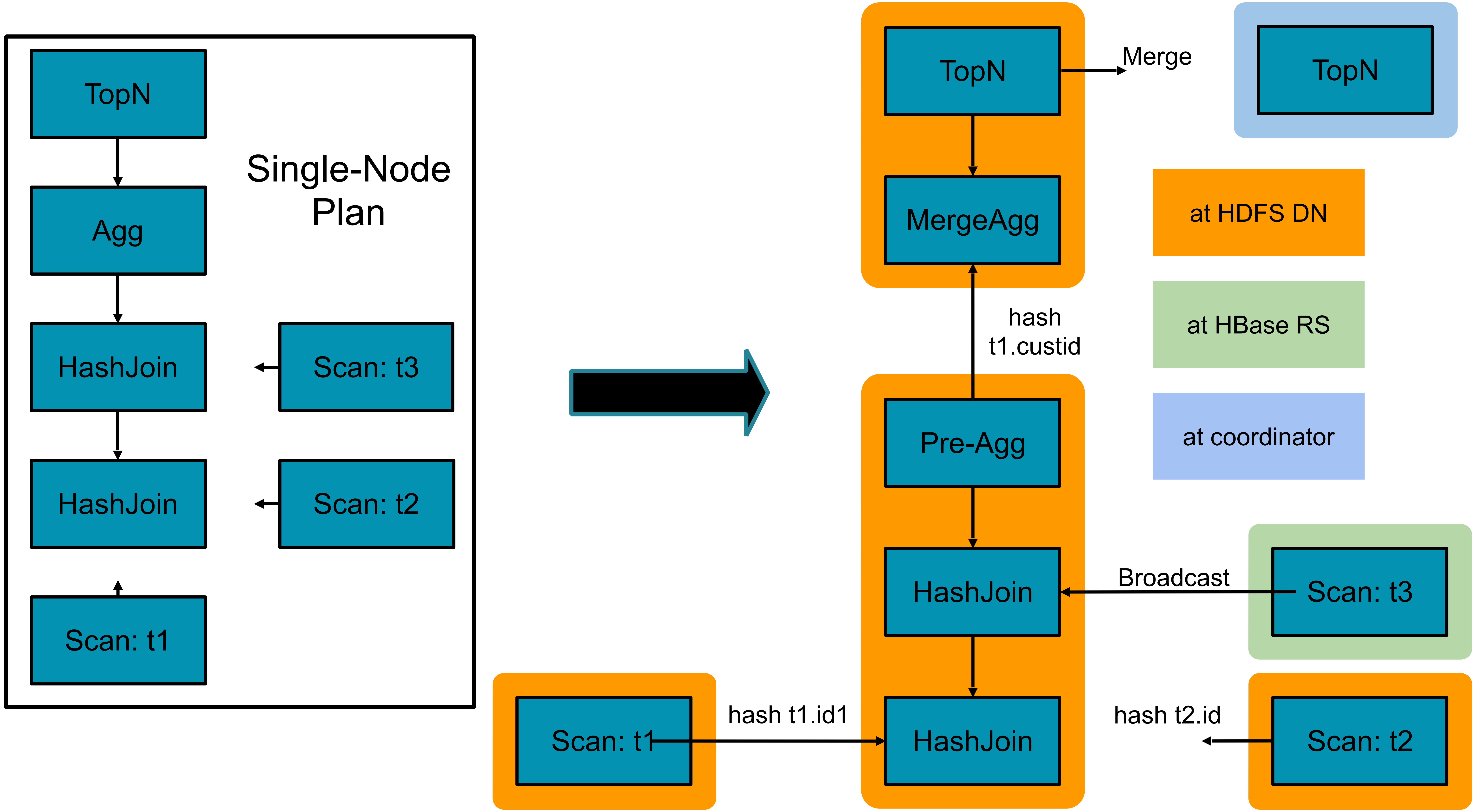
Query Planning: Distributed Plans

- Parallel aggregation:
  - pre-aggregation where data is first materialized
  - merge aggregation partitioned by grouping columns
- Parallel top-N:
  - initial top-N operation where data is first materialized
  - final top-N in single-node plan fragment

Query Planning: Distributed Plans — Example

- Scans are local: each scan receives its own fragment
- 1st join: large x large -> partitioned join
- 2nd scan: large x small -> broadcast join
- Pre-aggregation in fragment that materializes join result
- Merge aggregation after repartitioning on grouping column
- Initial top-N in fragment that does merge aggregation
- Final top-N in coordinator fragment

Query Planning: Distributed Plans



Impala Execution Engine

- 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

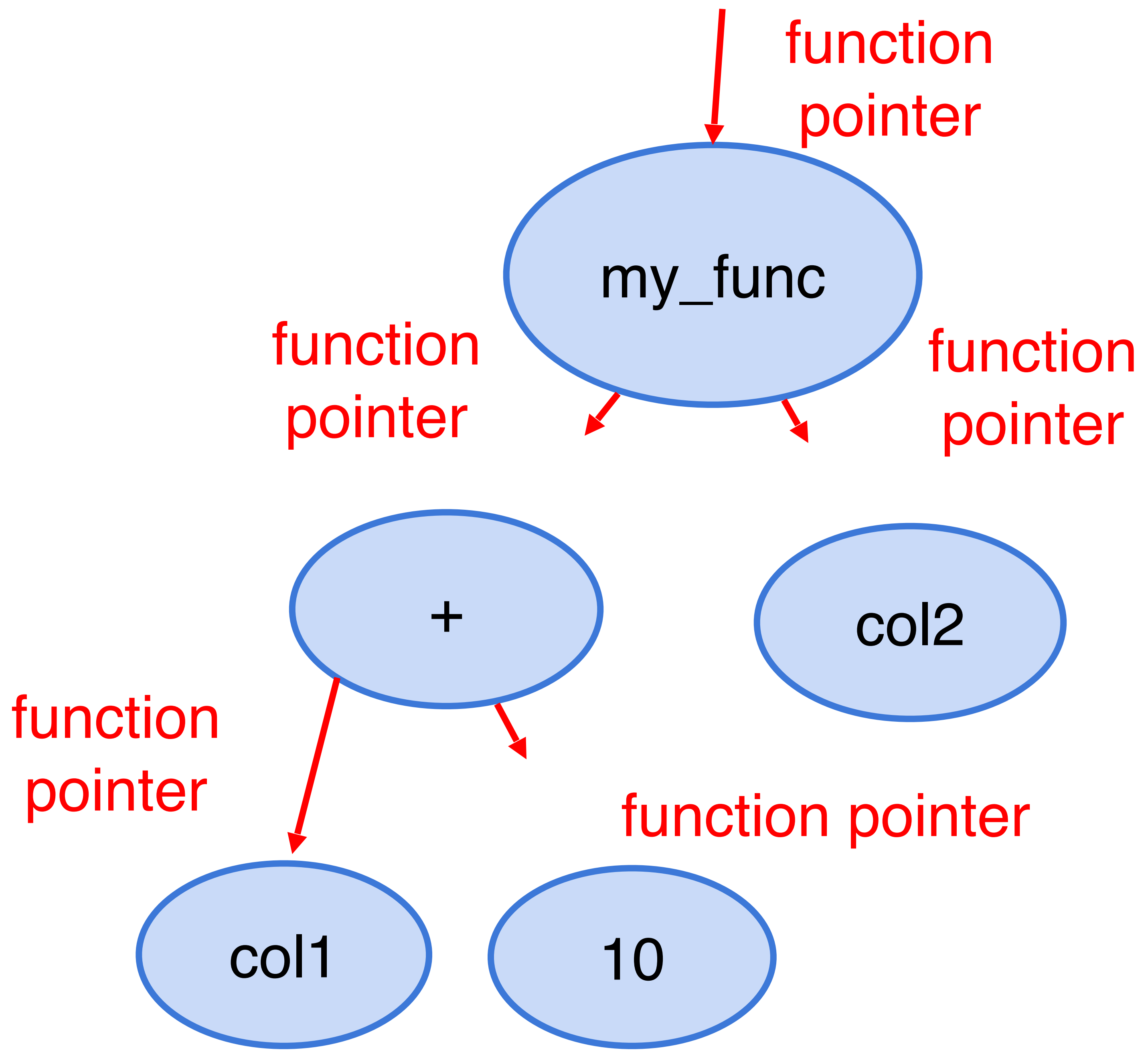
cloudera
Ask Bigger Questions

Impala 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)

Impala Runtime Code Generation – Example

```
IntVal my_func(const IntVal& v1, const IntVal& v2) {
  return IntVal(v1.val * 7 / v2.val);
}
SELECT my_func(col1 + 10, col2) FROM ...
```

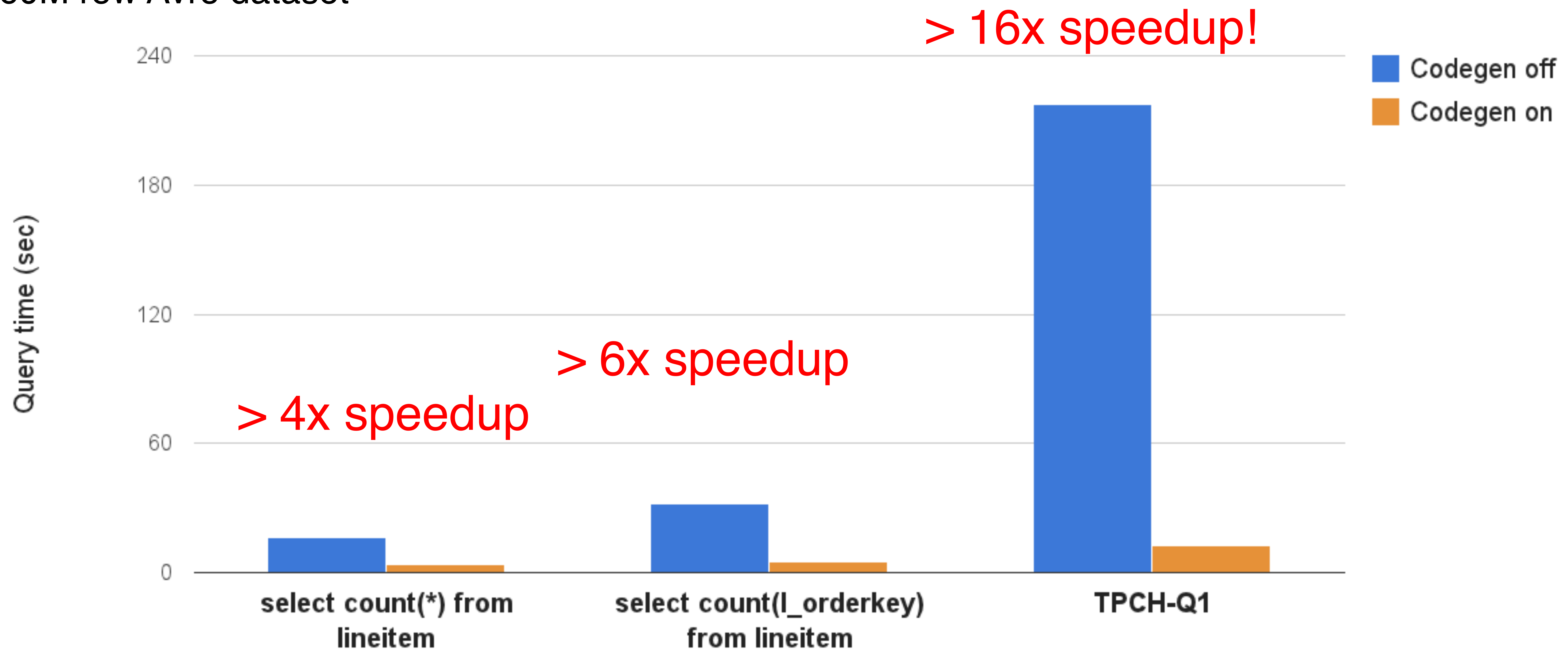


(col1 + 10) * 7 / col2

interpreted **codegen'd**

Impala Runtime Code Generation — Performance

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



cloudera Ask Bigger Questions

Resource Management: Admission Control

- Workload management in a distributed environment
- Enforce global limits on # of concurrently executing queries and/or memory consumption
- Admin configures pools with limits and assigns users to pools
- Decentralized: avoids single-node bottlenecks for low-latency, high-throughput scheduling
- Does not require Yarn/Llama
- Works in CDH4/CDH5

Resource Management: Admission Control

- Configure one or more resource pools
  - max # of concurrent queries, max memory, max queue size
  - same configuration as Yarn resource queues
  - easily configured via Cloudera Manager
- Each Impala node capable of making admission decisions: no single point of failure, no scaling bottleneck
- Incoming queries are executed, queued, or rejected
  - queue if too many queries running concurrently or not enough memory
  - reject if queue is full

Resource Management: YARN

- YARN is a centralized, cluster-wide resource management system that allows frameworks to share resources without resource partitioning between frameworks
- Impala can do resource reservation via YARN for individual queries
- However, YARN is targeted at batch environments: results in extra cost for both latency and throughput

Resource Management in Impala

- Admission control and YARN-based resource management cater to different workloads
- Use admission control for:
  - low-latency/high-throughput workloads
  - mostly 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
- Future roadmap: low-latency/high-throughput mixed workloads without resource partitioning

HDFS: A Storage System for Analytic Workloads

- High-efficiency data scans at or near hardware speed, both from disk and memory
- Short-circuit reads: bypass DataNode protocol when reading from local disk
  -> read at 100+MB/s per disk
- HDFS caching: access explicitly cached data w/o copy or checksumming
  -> access memory-resident data at memory bus speed
  -> enable in-memory processing

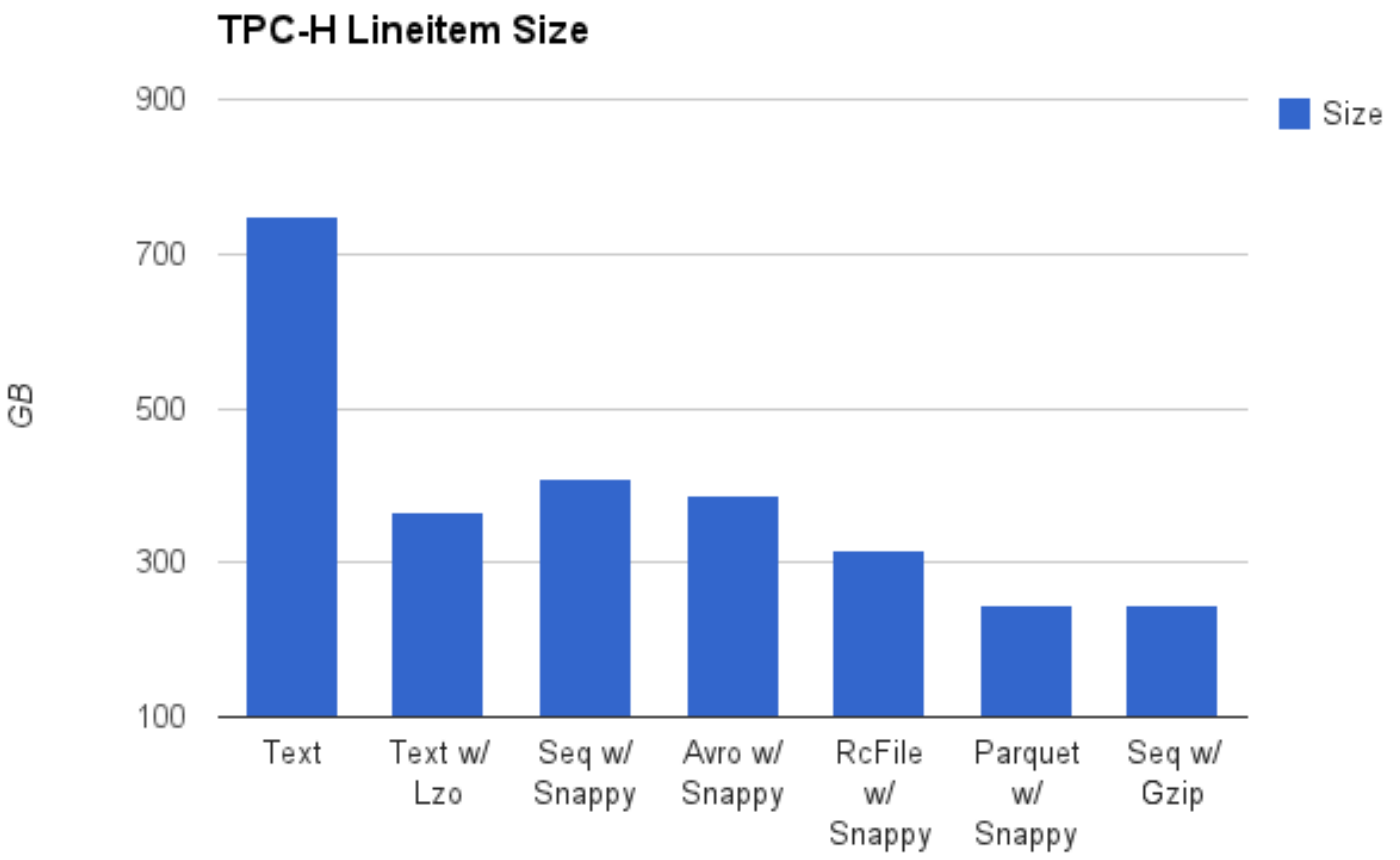
Parquet: Columnar Storage for Hadoop

- State-of-the-art, open-source columnar file format
- Available for (most) Hadoop processing frameworks:
  Impala, Hive, Pig, MapReduce, Cascading, …
- Offers both high compression and high scan efficiency
- Co-developed by Twitter and Cloudera
  - with contributors from Criteo, Stripe, Berkeley AMPlab, LinkedIn
  - Now an Apache incubator project
- Used in production at Twitter and Criteo
- The recommended format for Impala

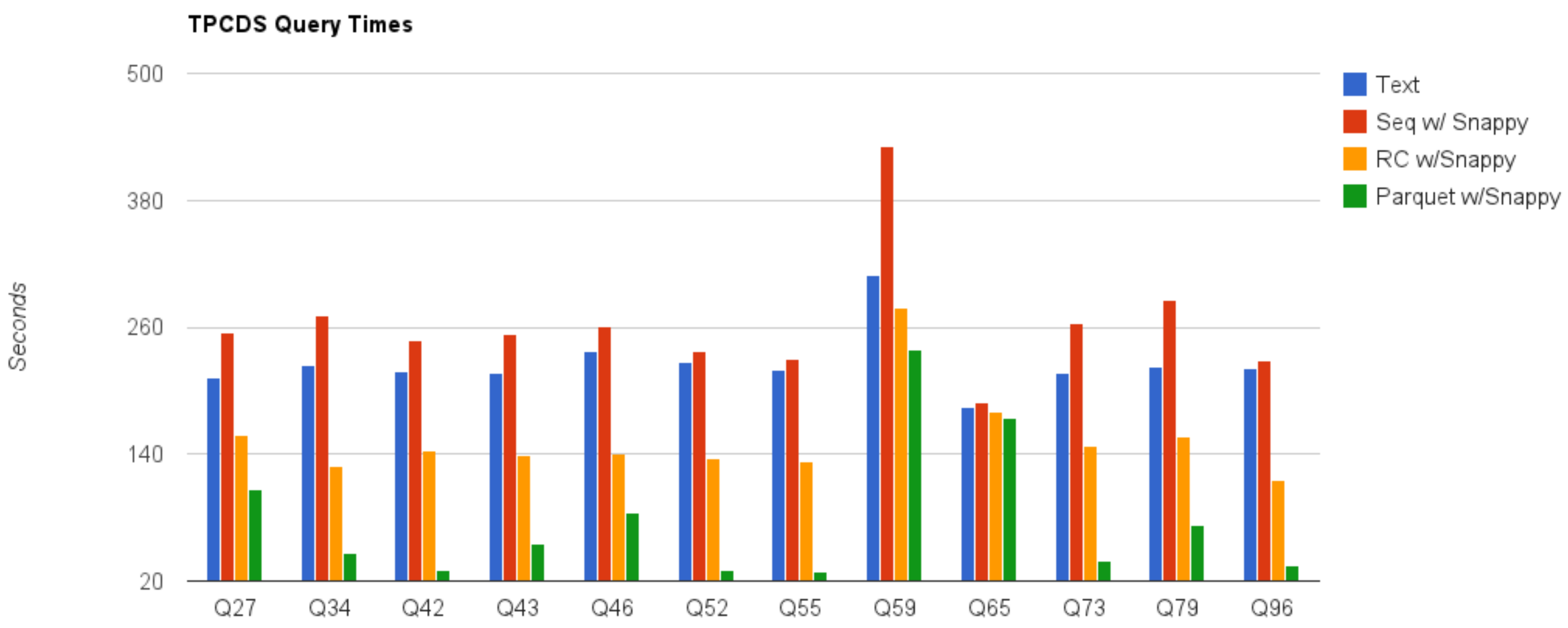
Parquet: The Details

- Columnar storage: column-major instead of the traditional row-major layout; used by all high-end analytic DBMSs
- Optimized storage of nested data structures: patterned after Dremel's ColumnIO format
- Extensible set of column encodings:
  - run-length and dictionary encodings in 1.2
  - delta and optimized string encodings in current version 2.0
- Embedded statistics: version 2.0 stores inlined column statistics for further optimization of scan efficiency
  - e.g. min/max indexes

Parquet: Storage Efficiency



Parquet: Scan Efficiency



cloudera
Ask Bigger Questions

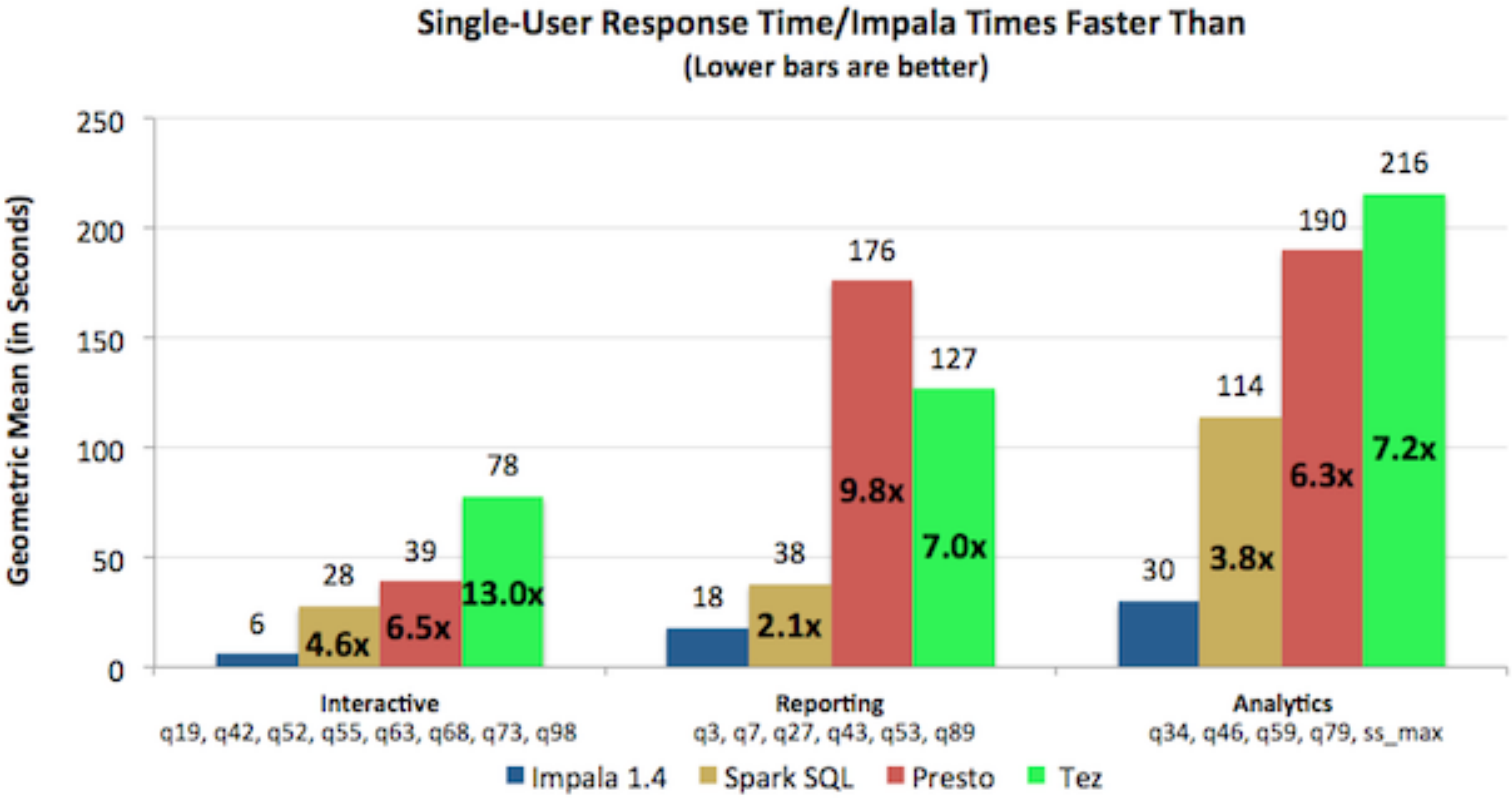
Cloudera Impala — Agenda

- Overview
- Architecture and Implementation
- **Evaluation**

Impala Performance

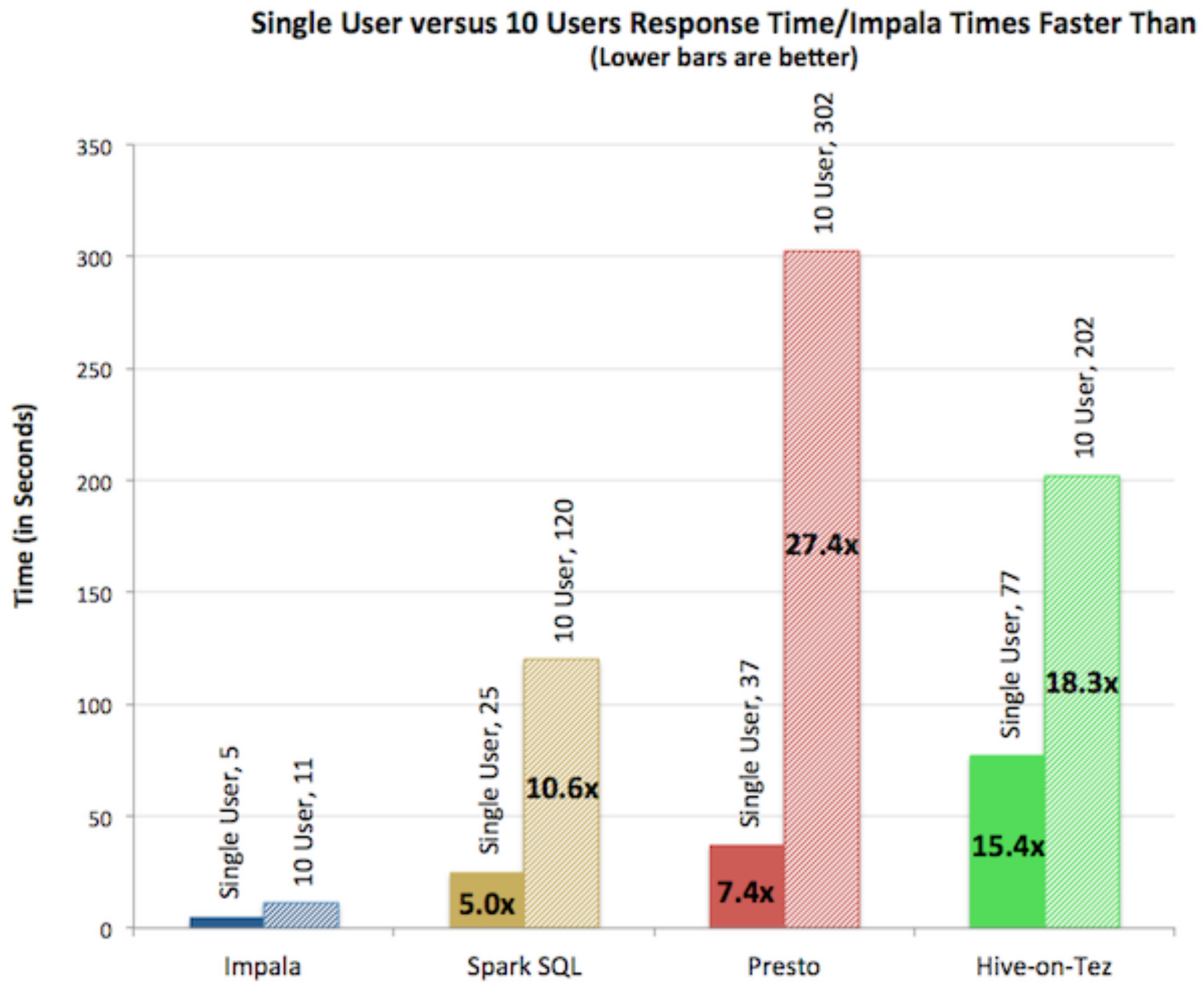
- Benchmark: TPC-DS
  - Subset of queries (21 queries)
  - 15TB scale factor data set
  - On 21-node cluster
    - 2 processors, 12 cores, Intel Xeon CPU E5-2630L 0 at 2.00GHz
    - 12 disk drives at 932GB each (one for the OS, the rest for HDFS)
    - 64GB memory
  - Comparison of: Impala 1.4, SparkSql 1.1, Presto 0.74, Hive 0.13 (with Tez)

Impala Performance: Single-User



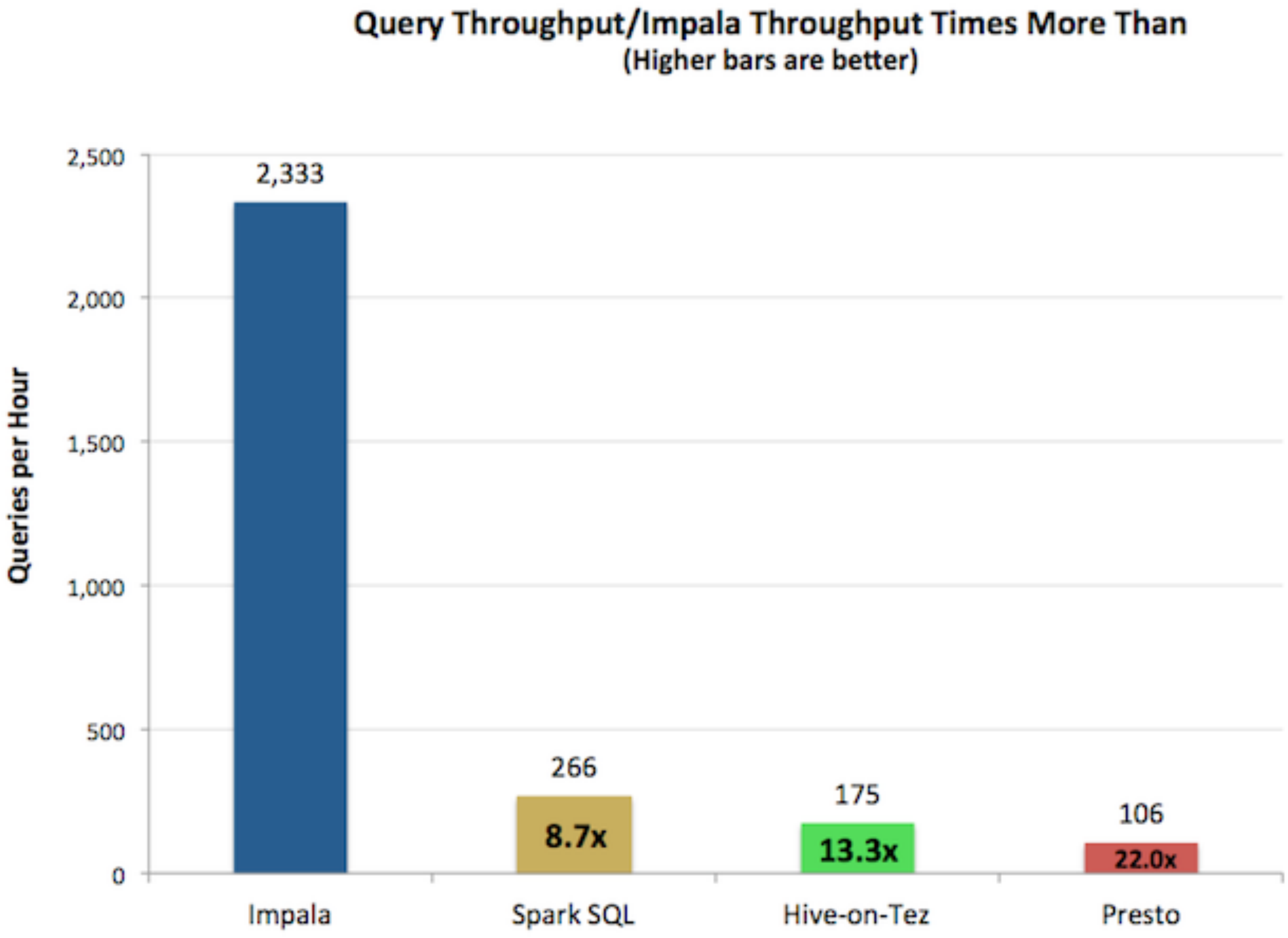
- single-user execution
- group queries by how much data they access:
  - interactive
  - reporting
  - deep analytic

Impala Performance: Multi-User



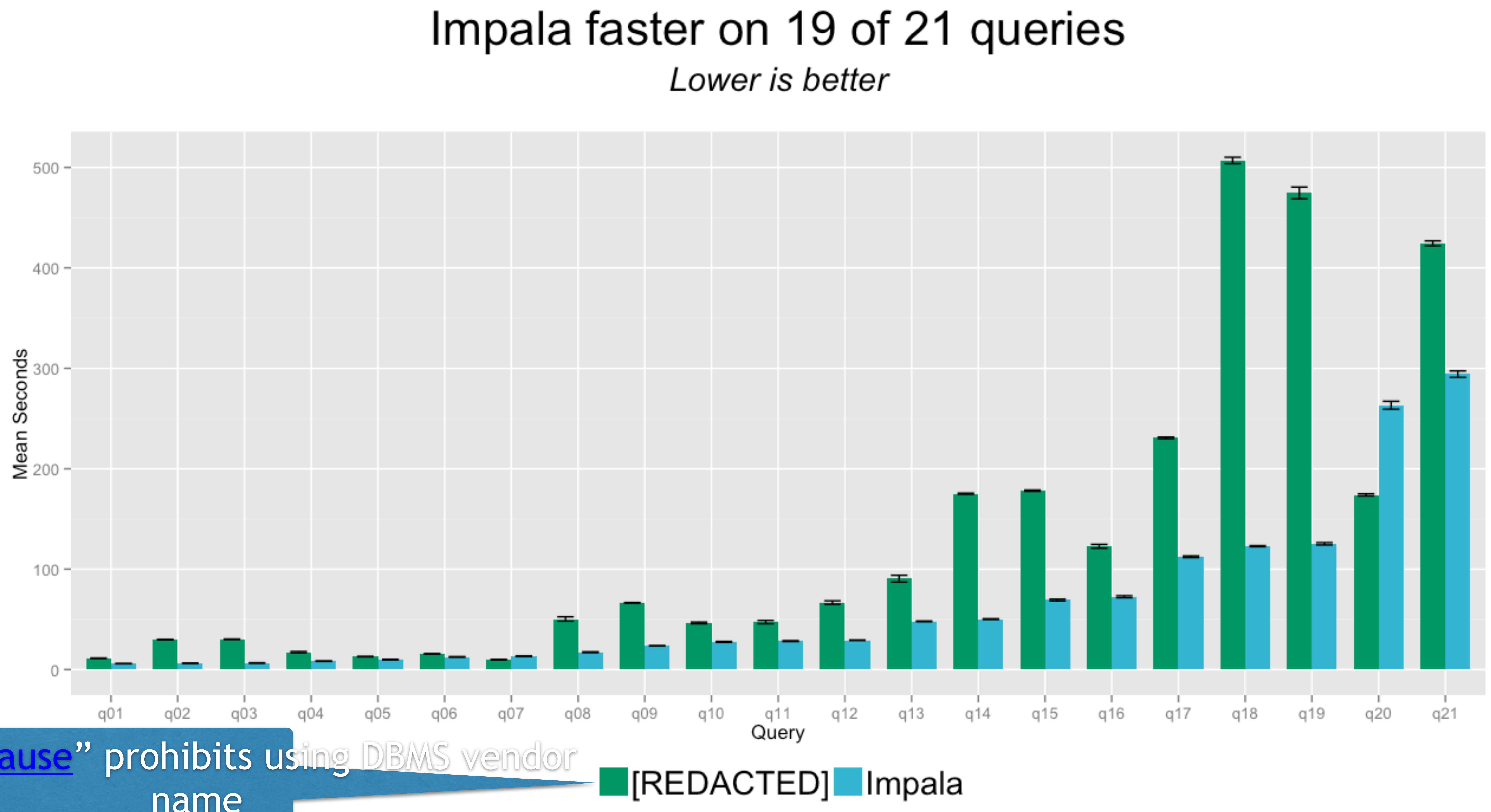
- 10 concurrent queries
- from the interactive bucket

Impala Performance: Multi-User



cloudera
Ask Bigger Questions

Impala vs. Commercial Competitor



"DeWitt Clause" prohibits using DBMS vendor name

cloudera
Ask Bigger Questions

Thank You