# Impala

## A Modern, Open Source SQL Engine for Hadoop

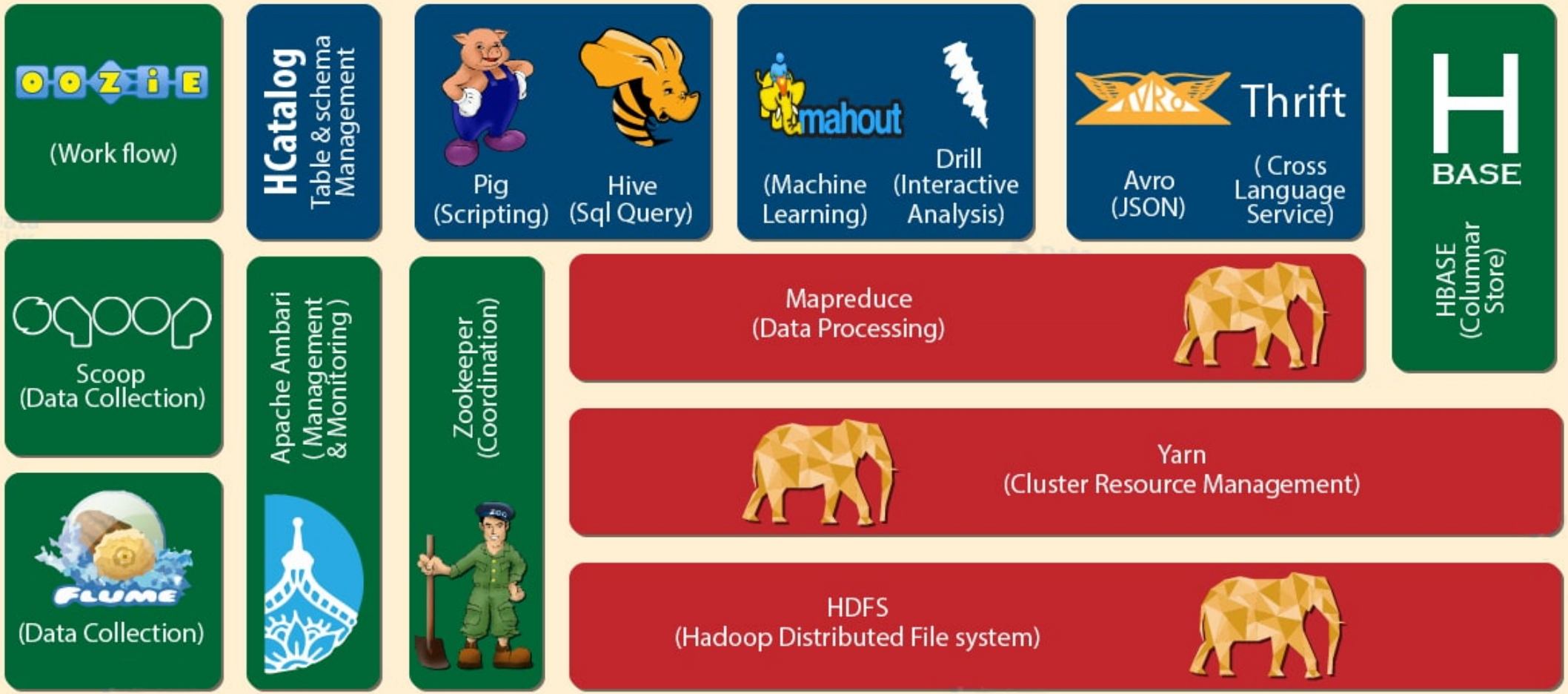
Yogesh
Chockalingam

# Agenda

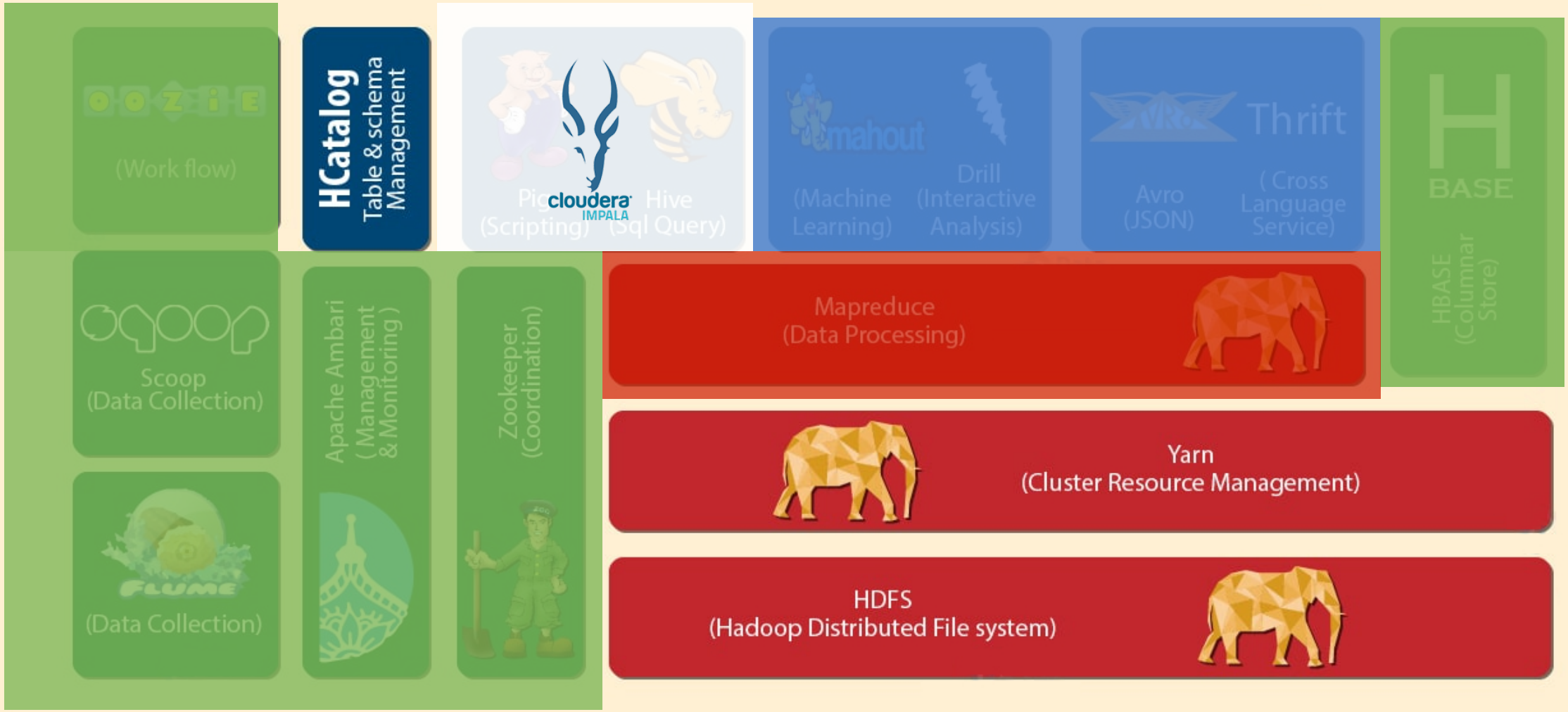
- Introduction
- Architecture
- Front End
- Back End
- Evaluation
- Comparison with Spark SQL

# Introduction

# Hadoop Ecosystem



# Hadoop Ecosystem



# Why not use Hive or HBase?

- Hive is a data warehousing tool built on top of Hadoop and uses Hive Query Language(HQL) for querying data stored in a Hadoop cluster.
- HQL automatically translates queries into MapReduce jobs.
- Hive doesn’t support transactions.

- HBase is a NoSQL database that runs on top of HDFS that provides real-time read/write access.

# Impala

- General purpose SQL query engine:
  - Works across analytical and transactional workloads
- High performance:
  - Execution engine written in C++
  - Runs directly within Hadoop
  - Does not use MapReduce
- MPP database support:
  - Multi-user workloads

# Creating tables

```
CREATE TABLE T (...) PARTITIONED BY (day int, month
  int) LOCATION '<hdfs-path>' STORED AS PARQUET;
```

For a partitioned table, data is placed in subdirectories whose paths reflect the partition columns' values.

For example, for day 17, month 2 of table T, all data files would be located in

```
<root>/day=17/month=2/
```

# Metadata

- Table metadata including the table definition, column names, data types, schema etc. are stored in HCatalog.

# INSERT / UPDATE / DELETE

- The user can add data to a table simply by copying/moving data files into the directory!
- Does NOT support UPDATE and DELETE.
  - Limitation of HDFS, as it does not support an in-place update.
  - Recompute the values and replace the data in the partitions.
- COMPUTE STATS <table> after inserts.
  - Those statistics will subsequently be used during query optimization.

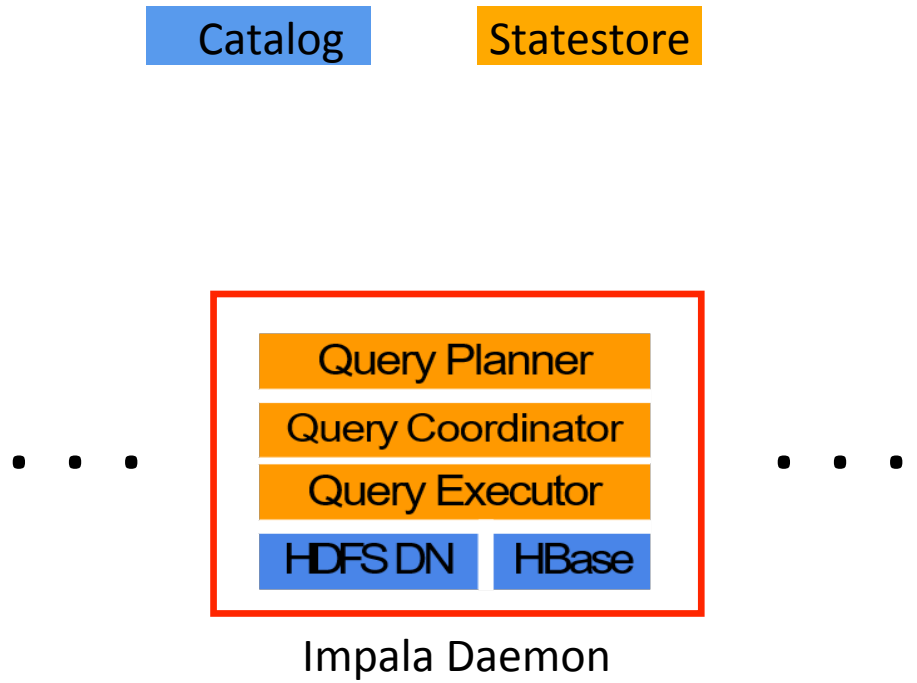
# Architecture

# I: Impala Daemon

Impala daemon service is dually responsible for:

1. Accepting queries from client processes and orchestrating their execution across the cluster. In this role it's called the query coordinator.
2. Executing individual query fragments on behalf of other Impala daemons.

- The Impala daemons are in constant communication with the *statestore*, to confirm which nodes are healthy and can accept new work.
- They also receive broadcast messages from the *catalog* daemon via the *statestore*, to keep track of metadata changes.



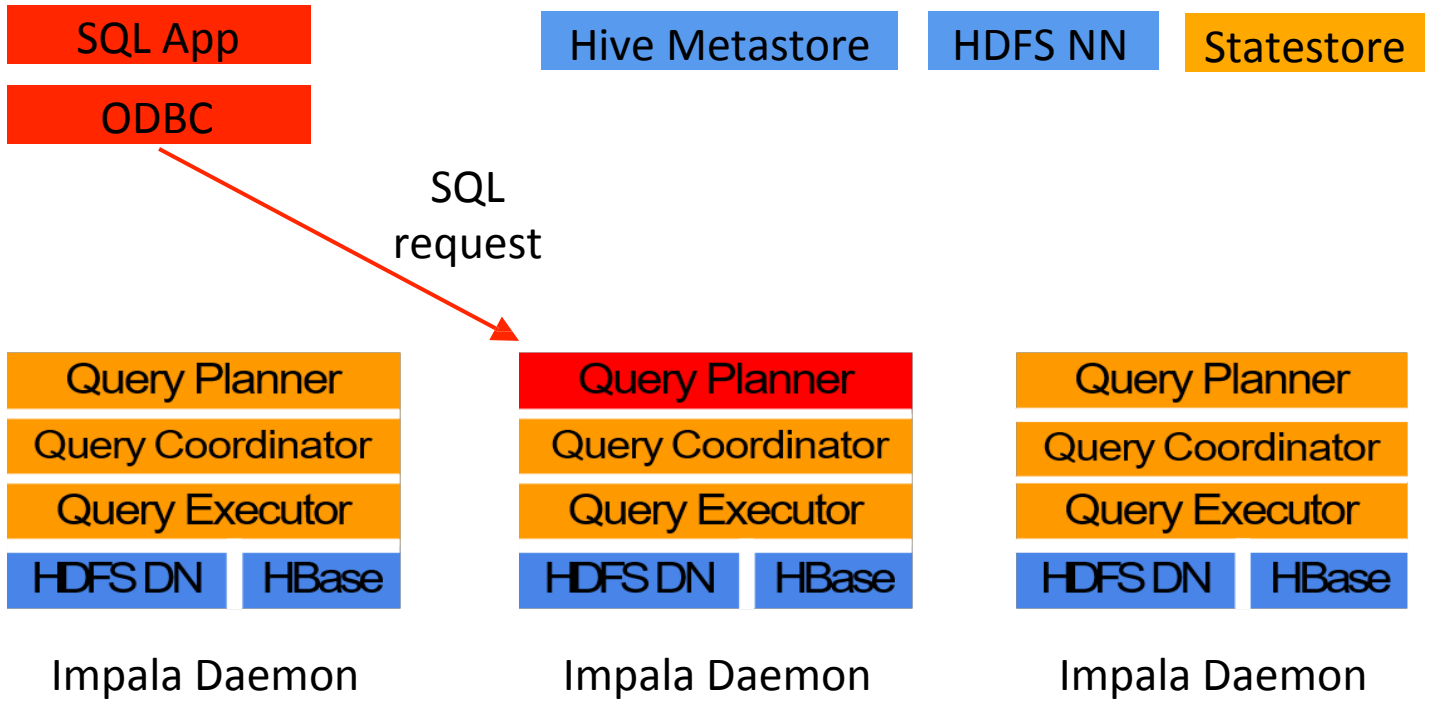
# II: Statestore Daemon

- Handles cluster membership information.
- Periodically sends two kinds of messages to Impala daemons:
  - Topic update: The new changes made since the last topic update message
  - Keepalive: A heartbeat mechanism
- If an Impala daemon goes offline, the statestore informs all the other Impala daemons so that future queries can avoid making requests to the unreachable node.

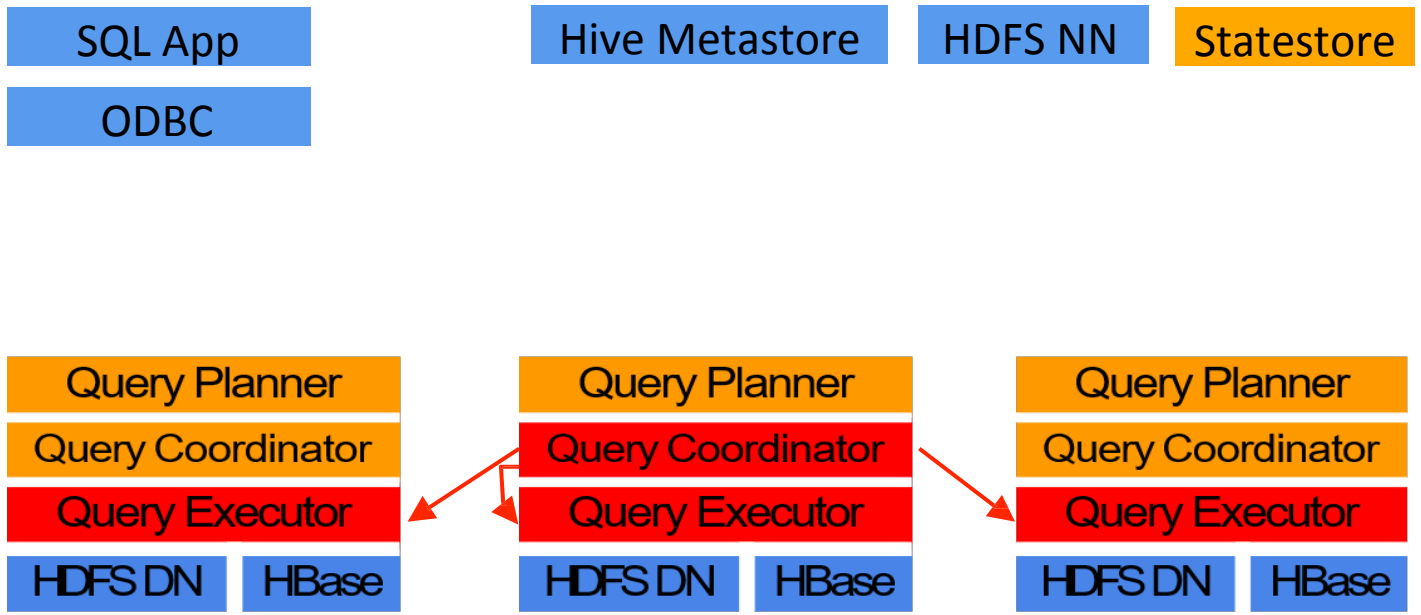
# III: Catalog Daemon

- Impala's catalog service serves catalog metadata to Impala daemons via the statestore broadcast mechanism, and executes DDL operations on behalf of Impala daemons.
- The catalog service pulls information from Hive Metastore and aggregates that information into an Impala-compatible catalog structure.
- This structure is then passed on to the statestore daemon which communicates with the Impala daemons.

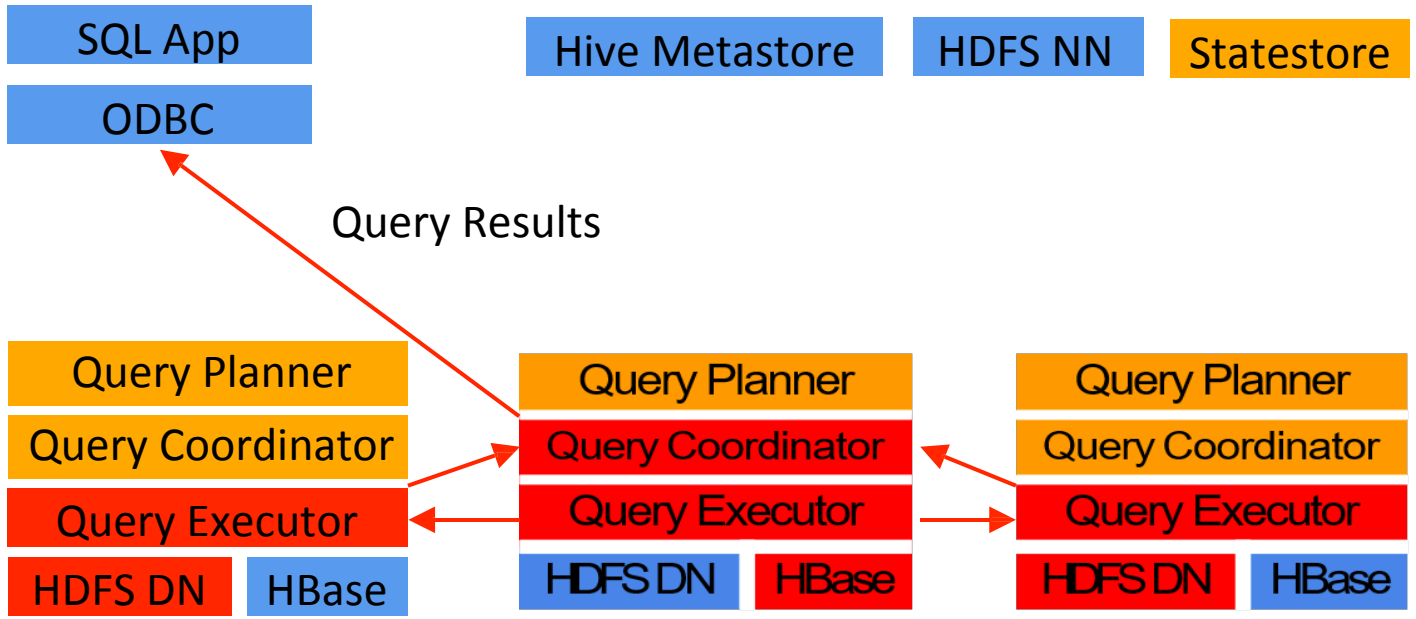
# 1. Request arrives from client via Thrift API



2. Planner turns request into collections of plan fragments. Coordinator initiates execution on remote Impala daemons.



3. Intermediate results are streamed between Impala daemons. Query results are streamed back to client.



# Front-End

# Query Plans

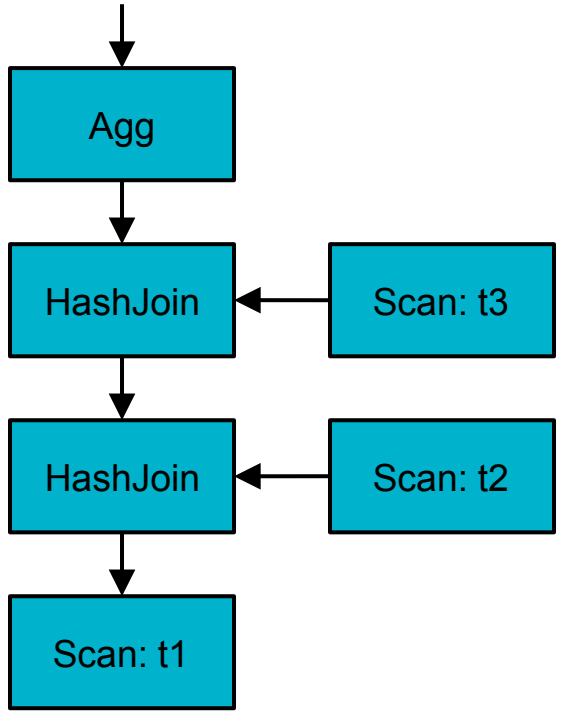
- The Impala frontend is responsible for compiling SQL text into query plans executable by the Impala backends.
- The query compilation process proceeds as follows:
  - Query parsing
  - Semantic analysis
  - Query planning/optimization
- Query planning
  1. Single node planning
  2. Plan parallelization and fragmentation

# Query Planning: Single Node

- In the first phase, the parse tree is translated into a non-executable single-node plan tree.

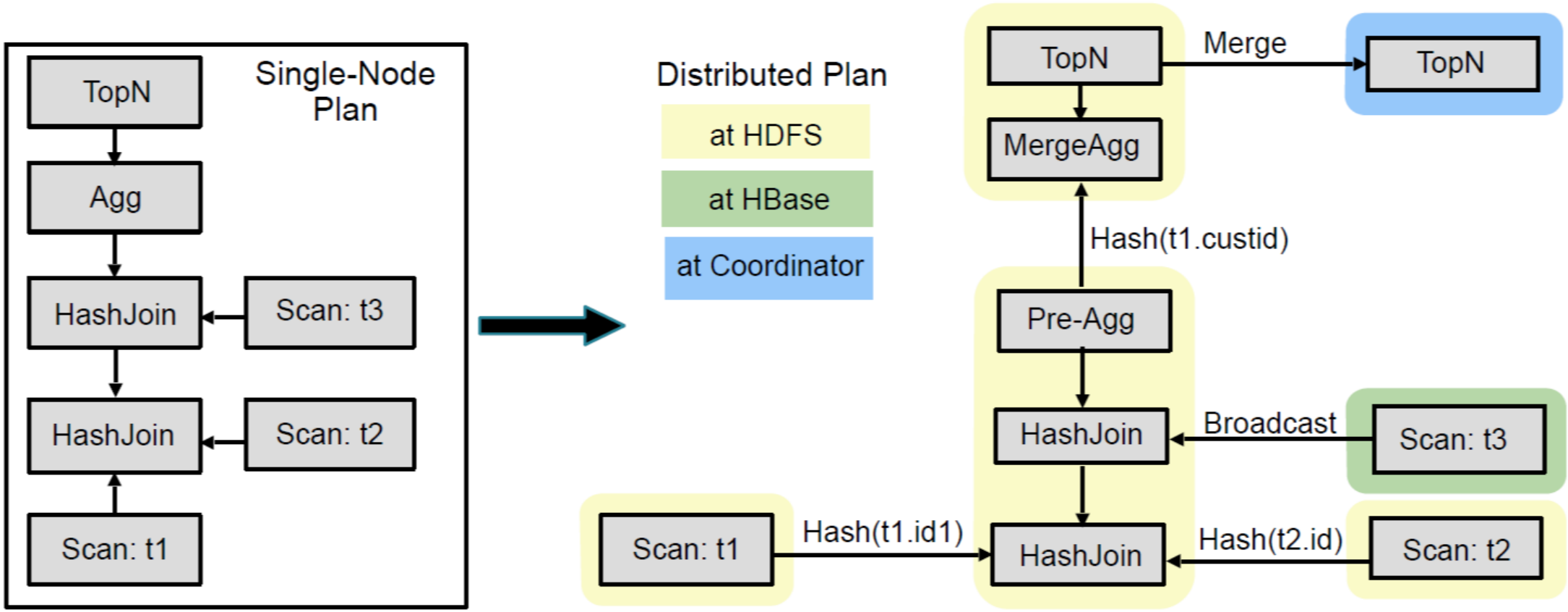
E.g. Query joining two HDFS tables (t1, t2) and one HBase table (t3) followed by an aggregation and order by with limit (top-n).

```
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 Nodes

- The second planning phase takes the single-node plan as input and produces a distributed execution plan. Goal:
  - To minimize data movement
  - Maximize scan locality as remote reads are considerably slower than local ones.
  - Cost--based decision based on column stats/estimated cost of data transfers
- Decide parallel join strategy:
  - **Broadcast Join**: Join is collocated with left-hand side 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.


Single-Node Plan
TopN
Agg
HashJoin
Scan: t3
HashJoin
Scan: t2
Scan: t1
Distributed Plan
at HDFS
at HBase
at Coordinator
TopN
Merge
TopN
MergeAgg
Hash(t1.custid)
Pre-Agg
HashJoin
Broadcast
Scan: t3
Scan: t1
Hash(t1.id1)
HashJoin
Hash(t2.id)
Scan: t2

# Back-End

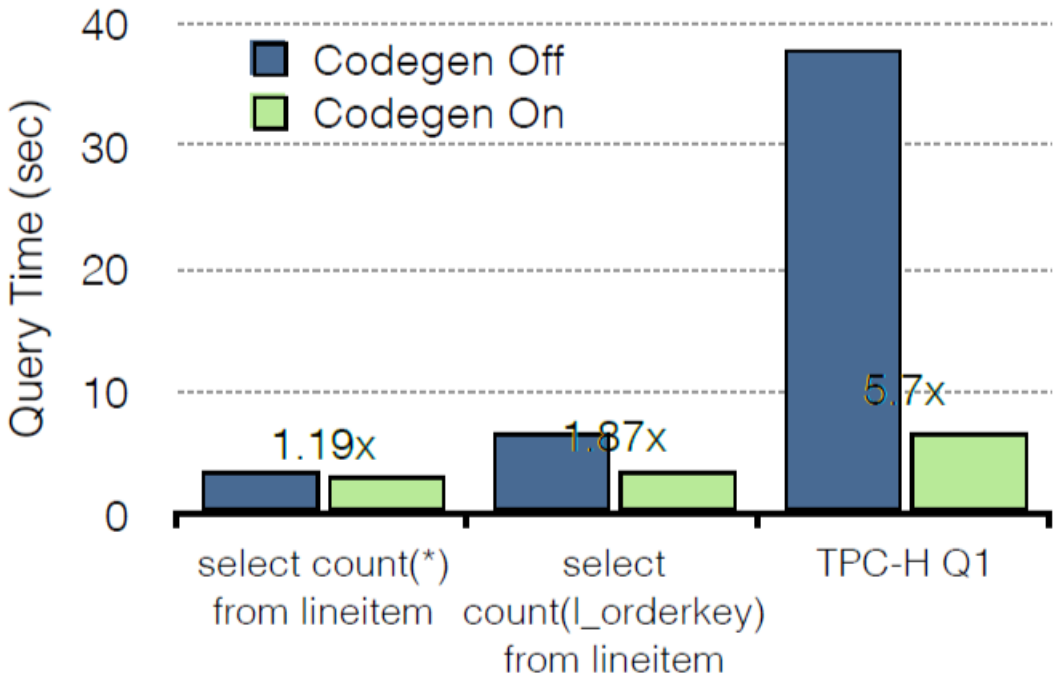
# Executing the Query

- Impala's backend receives query fragments from the front-end and is responsible for their execution.
- High performance:
  - Written in C++ for minimal execution overhead
  - Internal in-memory tuple format puts fixed-width data at fixed offsets
  - Uses intrinsic/special CPU instructions for tasks like text parsing and CRC computation.
  - **Runtime code generation for “big loops”**

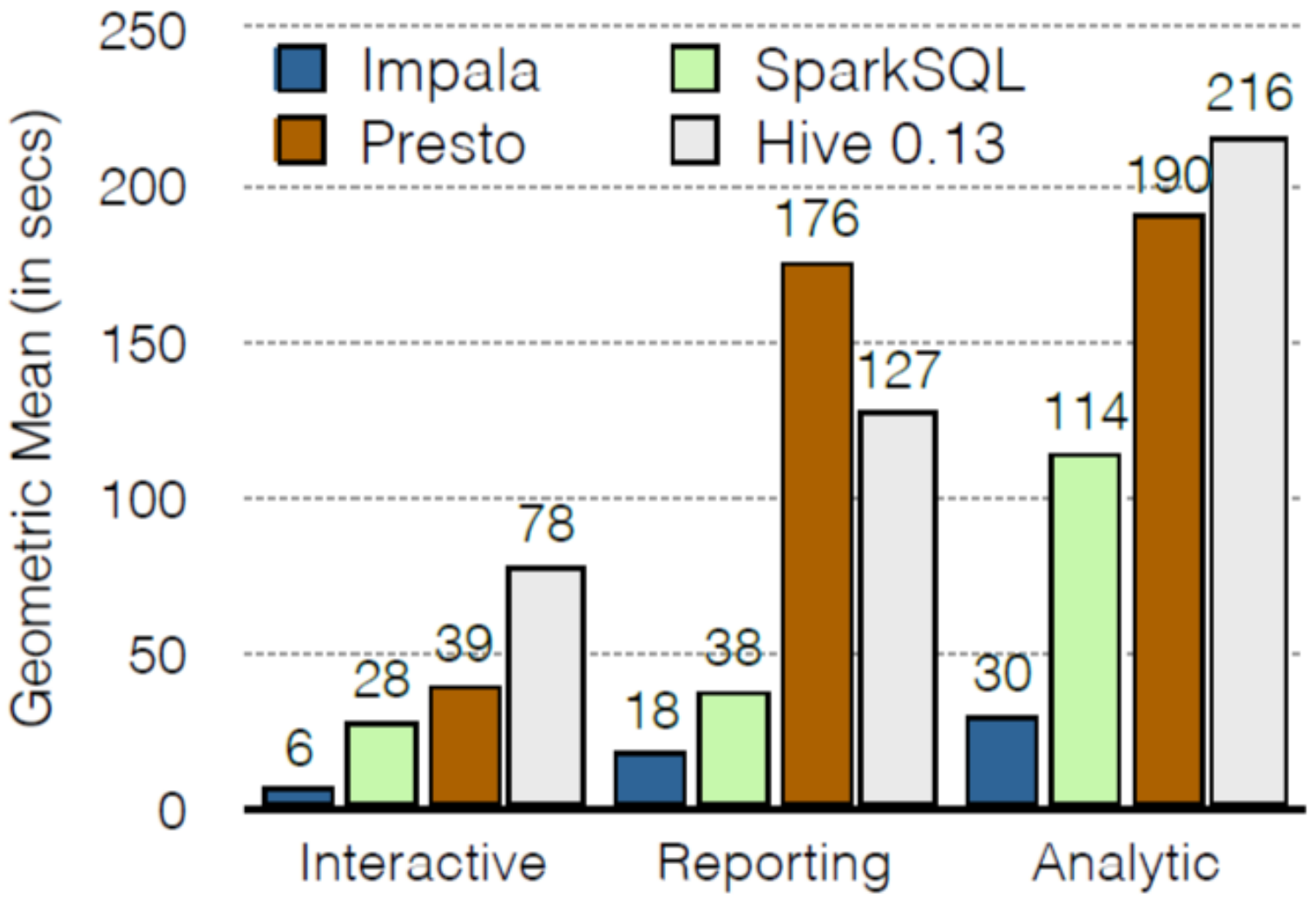
# Runtime Code Generation

Impala uses runtime code generation to produce query-specific versions of functions that are critical to performance.

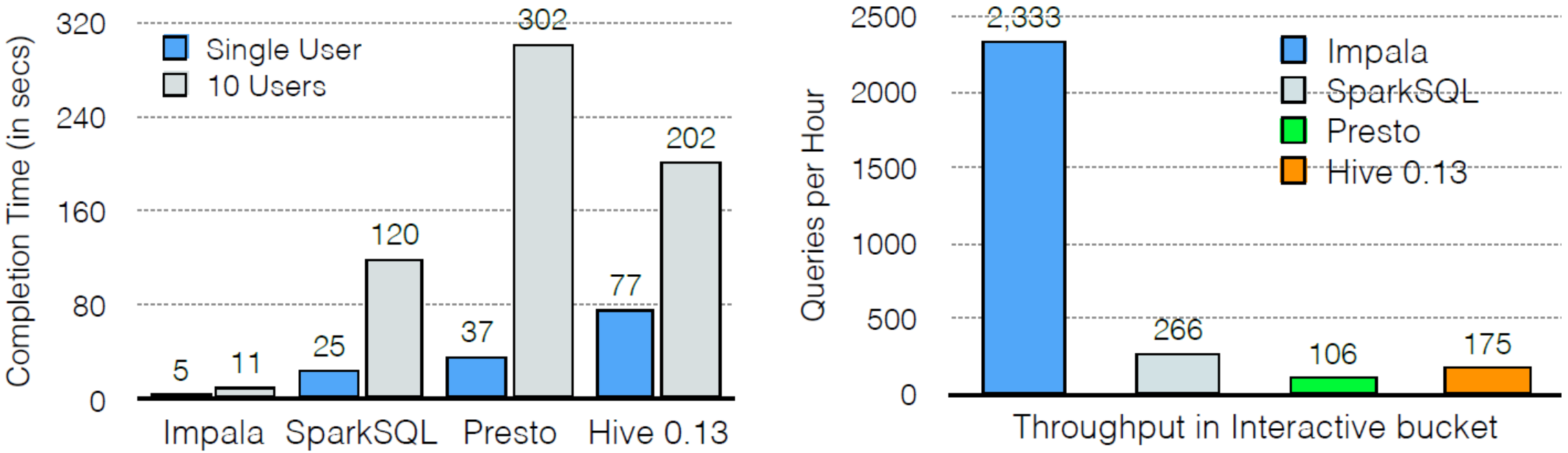
- For example, to convert every record to Impala's in-memory tuple format:
  - Known at query compile time: # of tuples in a batch, tuple layout, column types, etc.
  - Generate at compile time: unrolled loop that inlines all function calls, dead code elimination and minimizes branches.
  - Code generated using LLVM



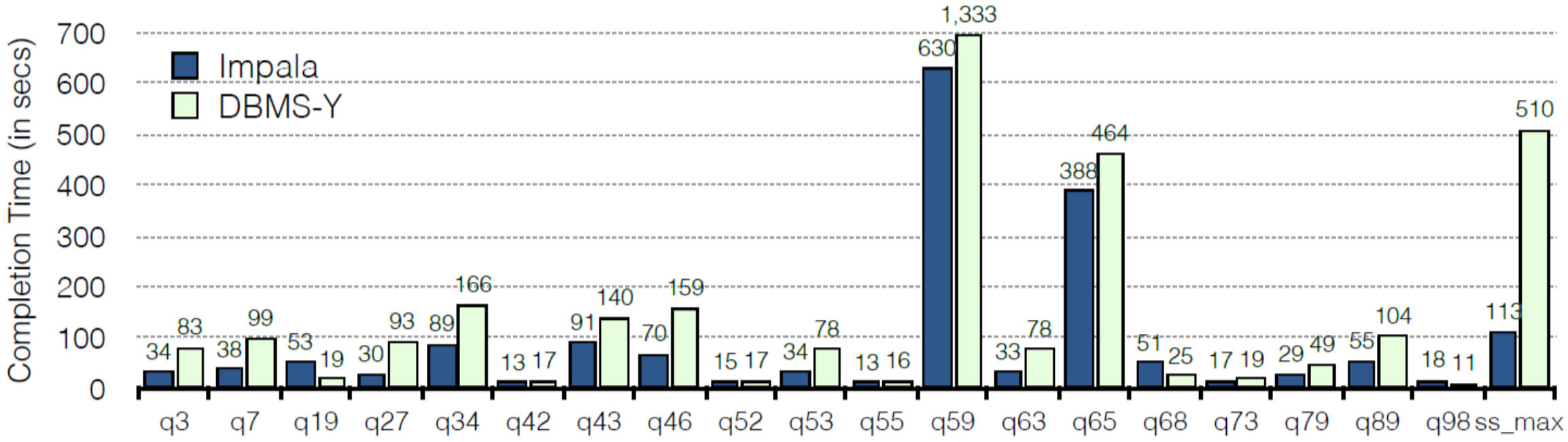
# Evaluation




Comparison of query response times on single-user runs.




Comparison of query response times and throughput on multi-user runs.




Comparison of the performance of Impala and a commercial analytic RDBMS.

https://github.com/cloudera/impala-tpcds-kit

# Comparison with Spark SQL

- Impala is faster than Spark SQL as it is an engine designed especially for the mission of interactive SQL over HDFS, and it has architecture concepts that helps it achieve that.
- For example the Impala ‘always-on’ daemons are up and waiting for queries 24/7 — something that is not part of Spark SQL.


Thank you!