How Impala Works

Yue Chen

http://linkedin.com/in/yuechen2

http://dataera.wordpress.com

What’s Impala?

This is Impala…

Goal of Impala

- A general SQL engine for distributed systems, supporting both OLTP and OLAP.
- Interactive (real-time) queries.
- Built on top of HDFS and HBase.
- Engine is written in C++, fast.
- The database execution engine is like that of massively parallel processing (MPP) databases, not using MapReduce.

What’s Impala?

- In-memory, distributed SQL query engine (no MapReduce)
- Native backend code (C++)
- Distributed on HDFS data nodes

What’s Impala?

- **Interactive SQL**
  - Typically 5-65x faster than Hive (observed up to 100x faster)
  - Responses in seconds instead of minutes (sometimes sub-second)
- **Approx. ANSI-92 standard SQL queries with HiveQL**
  - Compatible SQL interface for existing Hadoop/CDH applications
  - Based on industry standard SQL
- **Natively on Hadoop/HBase storage and metadata**
  - Flexibility, scale, and cost advantages of Hadoop
  - No duplication/synchronization of data and metadata
  - Local processing to avoid network bottlenecks
- **Separate runtime from MapReduce**
  - MapReduce is designed and great for batch
  - Impala is purpose-built for low-latency SQL queries on Hadoop

Why Impala?

FAST!

Why HDFS?

- Low cost
- Reliability
- Easy to scale out

Architecture

Architecture Overview

- impalad daemon runs on HDFS nodes
- statestored for cluster metadata
- (Hive) metastore for database metadata
- Queries run on relevant nodes
- Data streamed to clients

Architecture Overview

- Submit queries via Hue/Beeswax, Thrift API, CLI, ODBC, JDBC
- No fault tolerance (query fails if any query on any node fails)
- Intermediate data never hits disk

statestored

- Acts as a cluster monitor
- Not a single point of failure

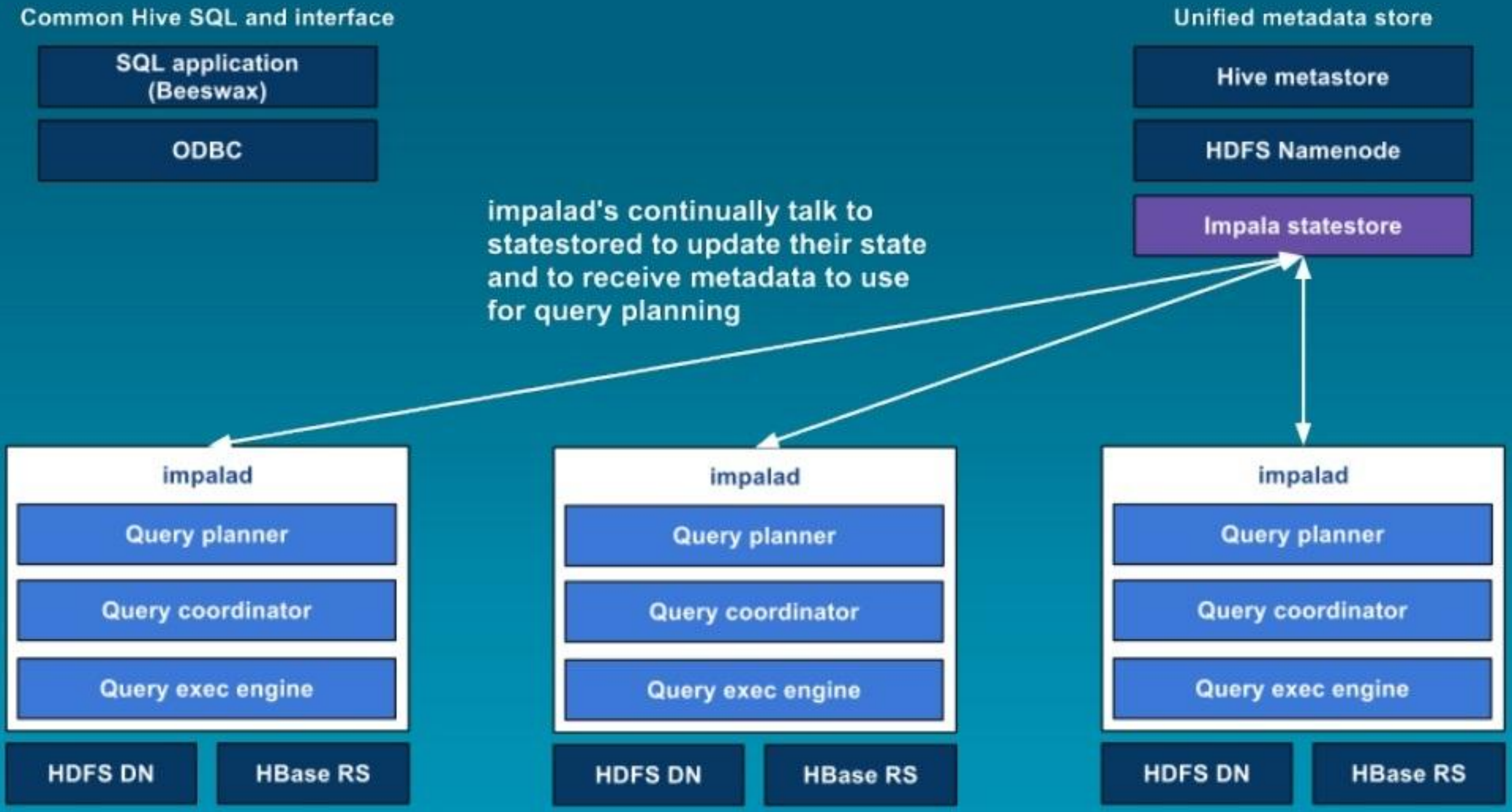
Metadata

- Uses Hive metastore
- Daemons cache metadata
- Can create tables in Hive or Impala

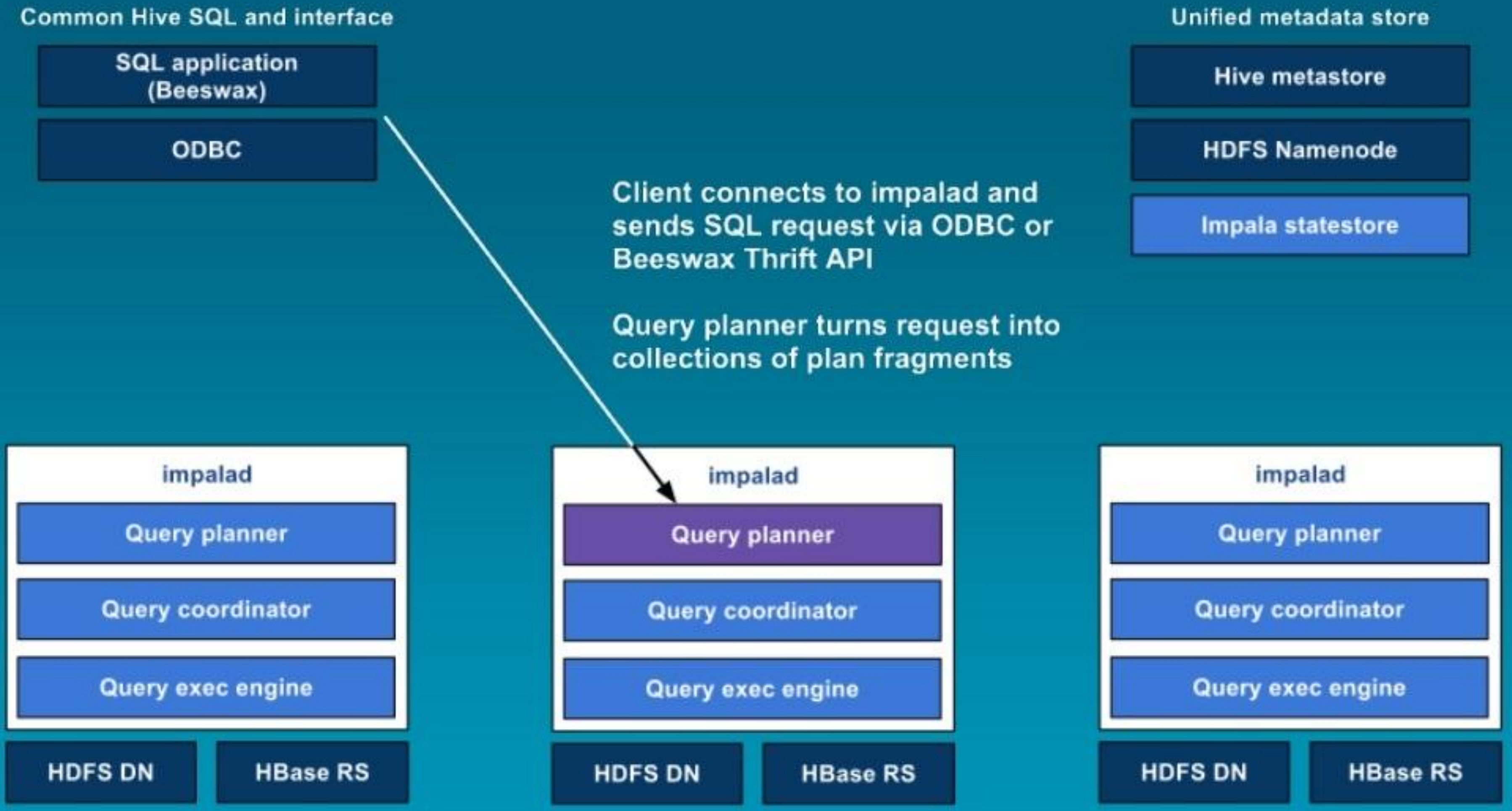
Impala Architecture Summary

- **impalad**
  - Runs on every node
  - Handles client requests
  - Handles query planning & execution
- **statestored**
  - Provides name service
  - Metadata distribution
  - Used for finding data
- **catalogd**
  - Relays metadata changes to all impalad’ s

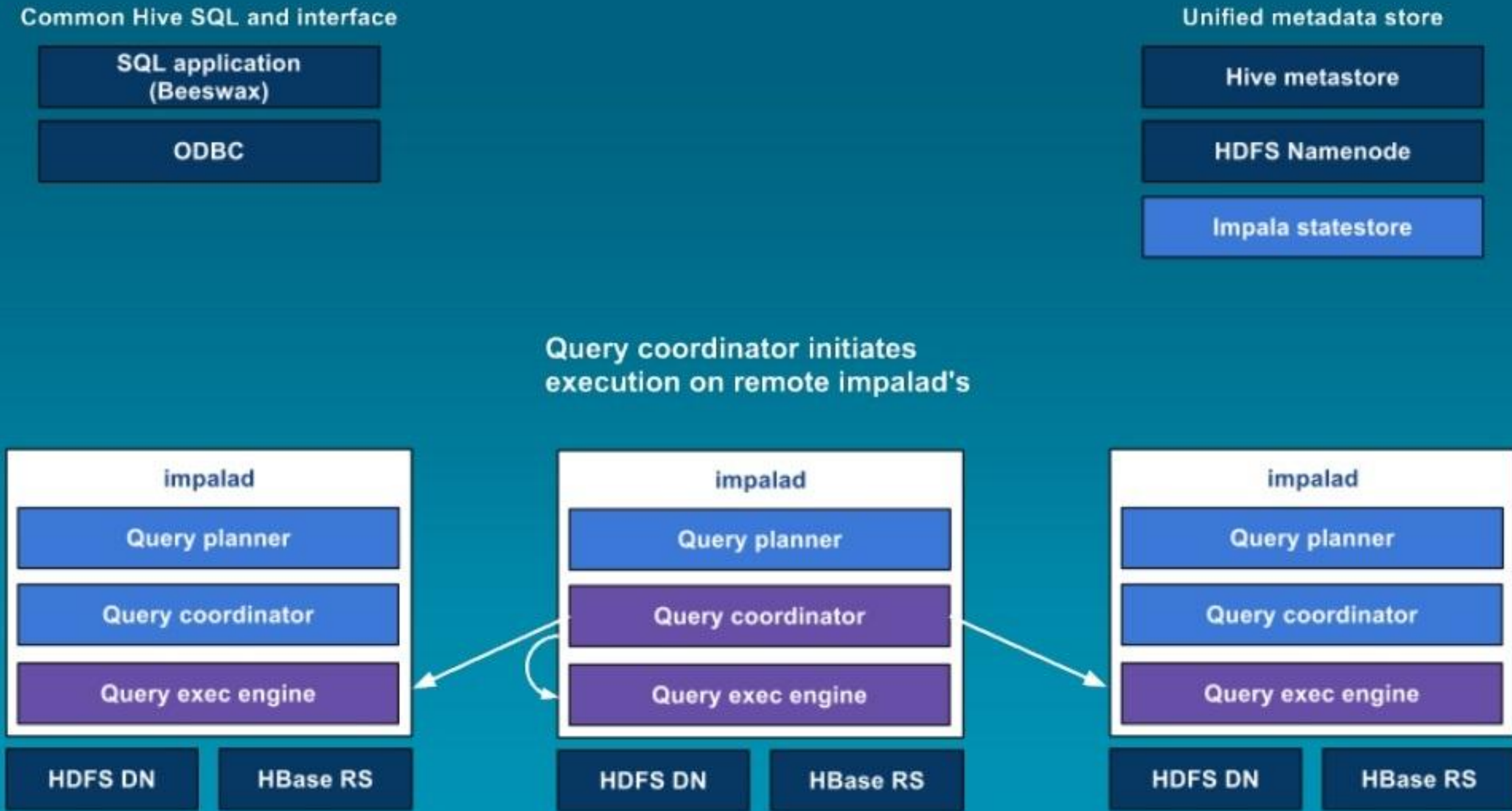
Impala: Architecture



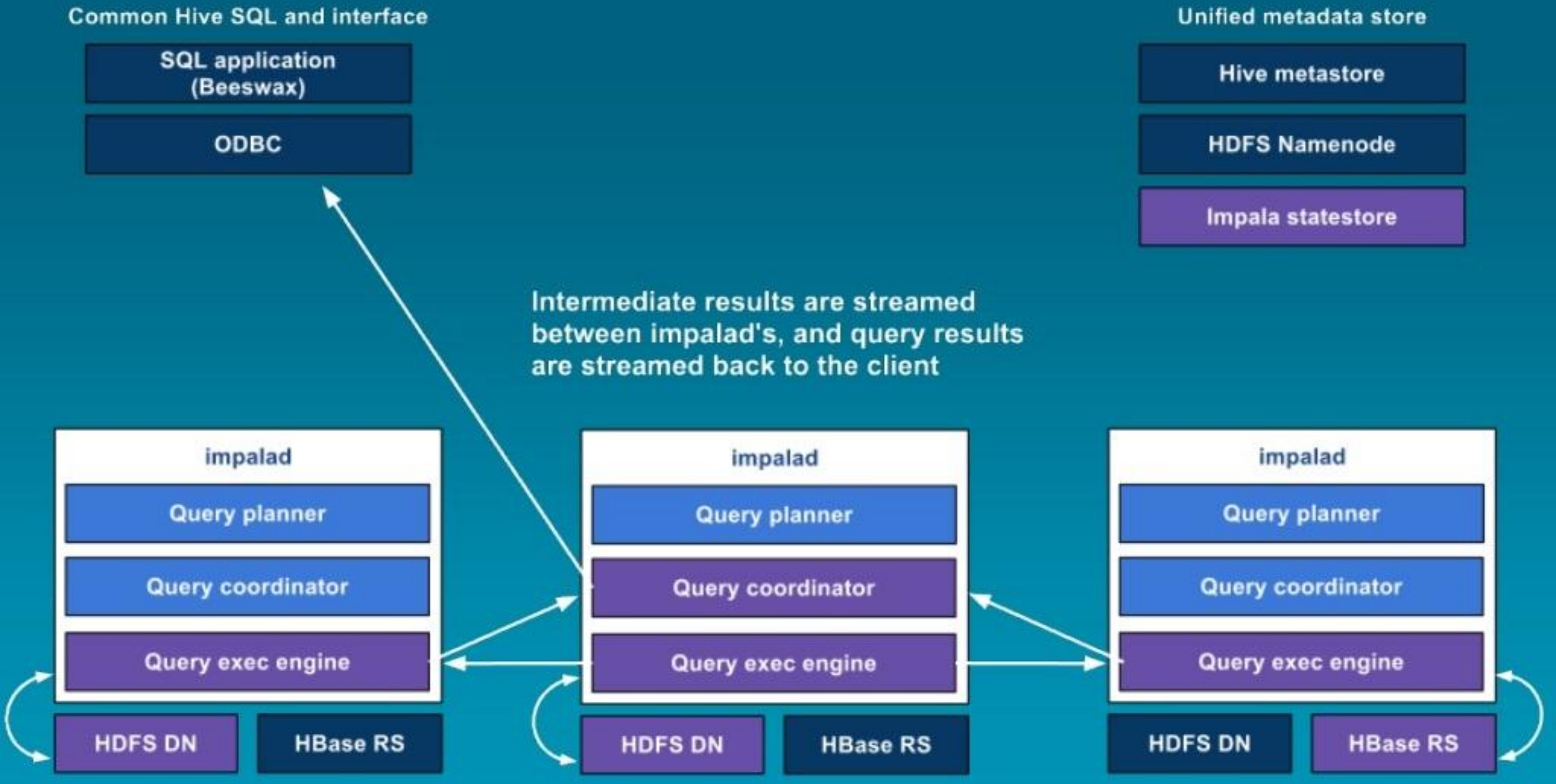
Impala: Architecture



Impala: Architecture



Impala: Architecture



Impala Architecture: Query Execution Phases

- **Client** SQL arrives via ODBC/JDBC/Hue GUI/Shell
- **Planner** turns request into collections of plan fragments
- **Coordinator** initiates execution on impalad's local to data
- During **execution**:
  - intermediate results are streamed between executors
  - query results are streamed back to client
  - subject to limitations imposed to blocking operators (top-n, aggregation)

Query Planning: Overview

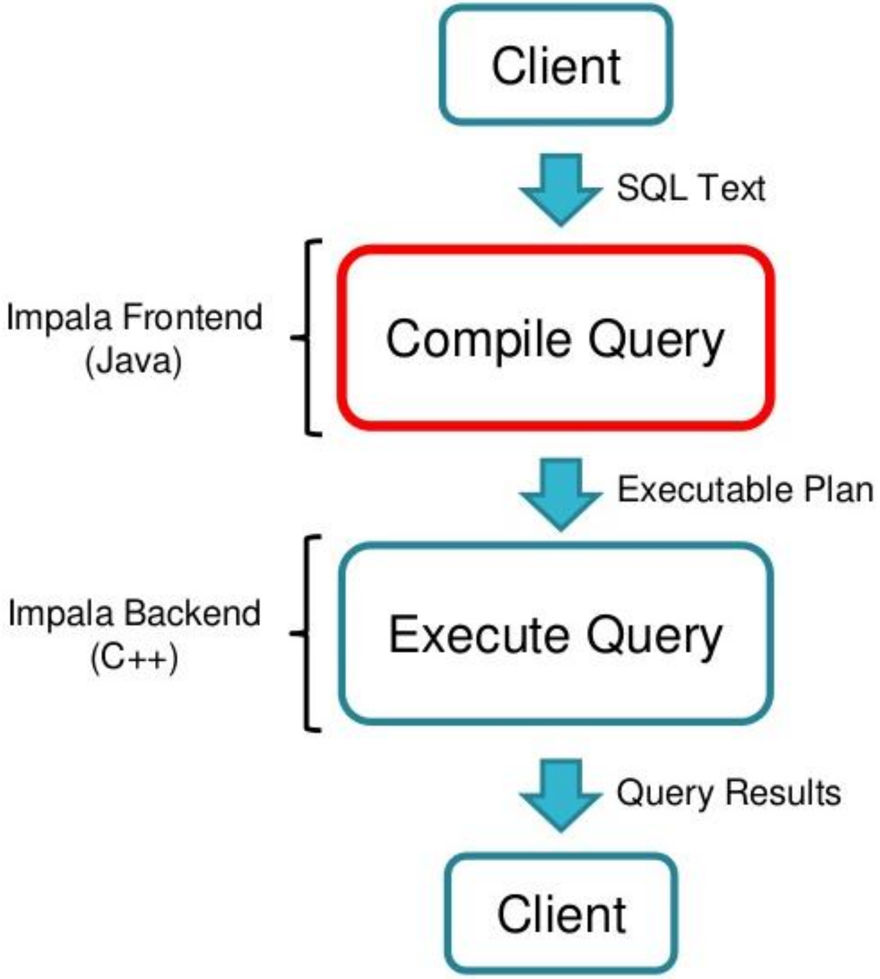
- **Java "front-end"**
- **2-phase planning process:**
  - single-node plan: left-deep tree of plan operators
  - partitioning of operator tree into plan fragments for parallel execution
- **Parallelization of plan operators:**
  - all query operators are fully distributed
  - joins: either broadcast or partitioned: decision is cost based
  - aggregation: fully parallel pre-aggregation and merge aggregation
  - top-n: initial stage done in parallel
- **Join order = FROM clause order** (soon to be cost-based)

Impala Partition

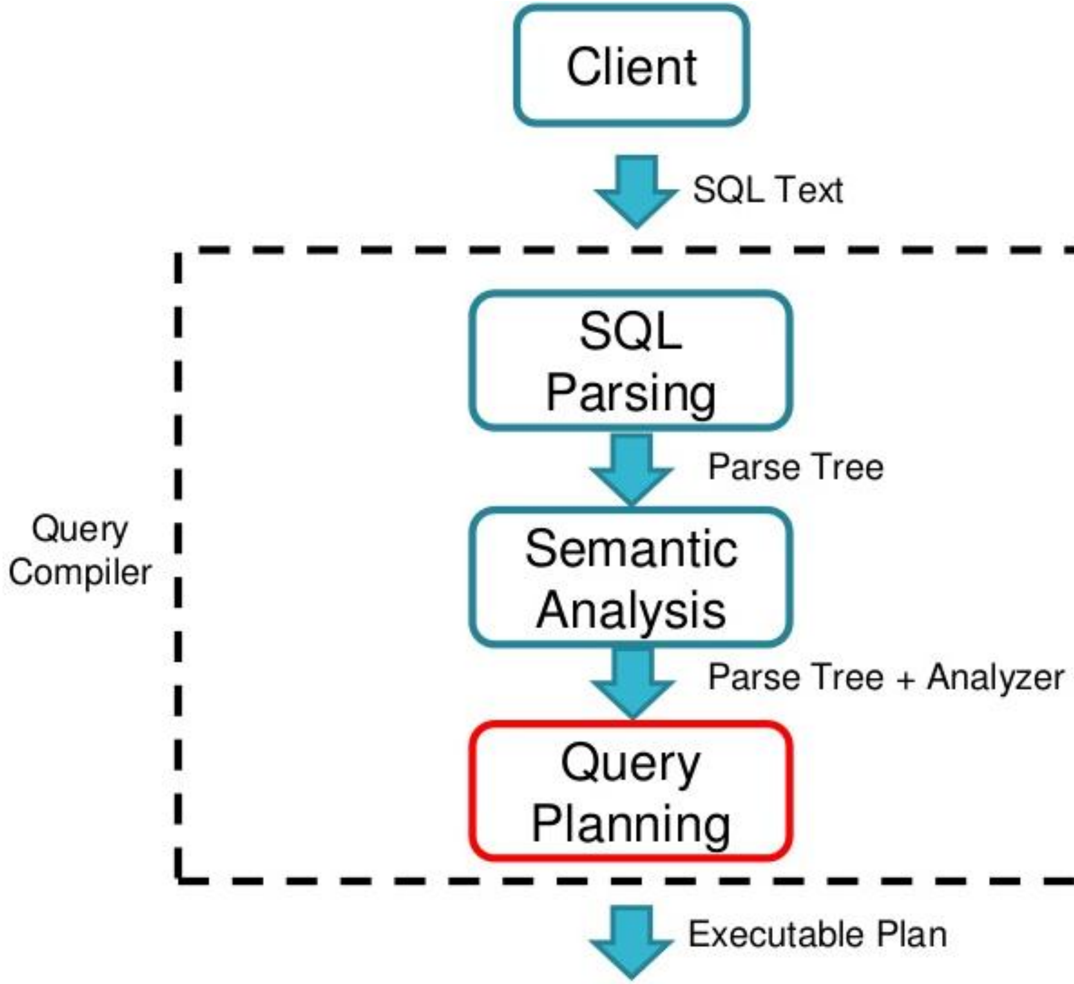
- Example:
  - create table census (name string, census_year int) partitioned by (year int);
  - insert into census partition (year=2010) values ('Smith',2010),('Jones',2010);
- Each partition has its own HDFS directory, and all the data for that partition is stored in a data file in that directory
- To manually define how to partition the table (e.g., year mod 5 == 0), we have to create a new column to store the calculation result and then do the partition

Frontend

Flow of a SQL query



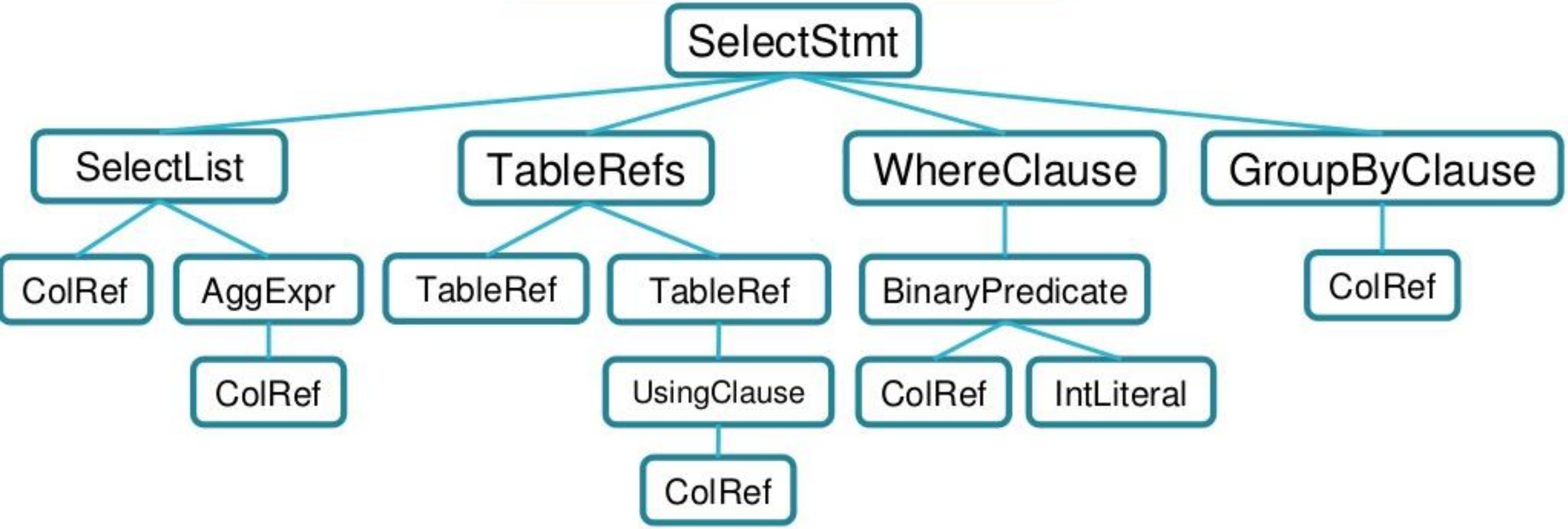
Query Compilation



Query Parsing

- Applies SQL grammar, reports syntax errors
- Produces parse tree capturing syntactic structure of query

```
SELECT c1, SUM(c2)
FROM t1 JOIN t2 USING(id)
WHERE c3 > 10 GROUP BY c1
```



Semantic Analysis

```
SELECT c1, SUM(c2)
FROM t1 JOIN t2 USING(id)
WHERE c3 > 10 GROUP BY c1
```

- **Precondition: Query is syntactically valid. Analysis operates on parse tree.**
- Consults table metadata
  - Do t1 and t2 exist? Does c1 exist in t1 or t2 (or both → error)? Does id exist in t1 and t2?
  - Does the user have privileges to SELECT from t1?
- Checks type compatibility of expressions, adds implicit casts
  - c3 > 10 → c3 > cast(10 as bigint)
- SQL rules (semantic, not syntactic)
  - Does c1 appear in the GROUP BY clause?

Semantic Analysis

- Expression substitution for views
  - Resolve column references against base tables

```
SELECT c1, SUM(c2)
FROM (SELECT dept AS c1, revenue AS c2,
        month AS c3 FROM t1) AS v
WHERE c3 > 10 GROUP BY c1
```

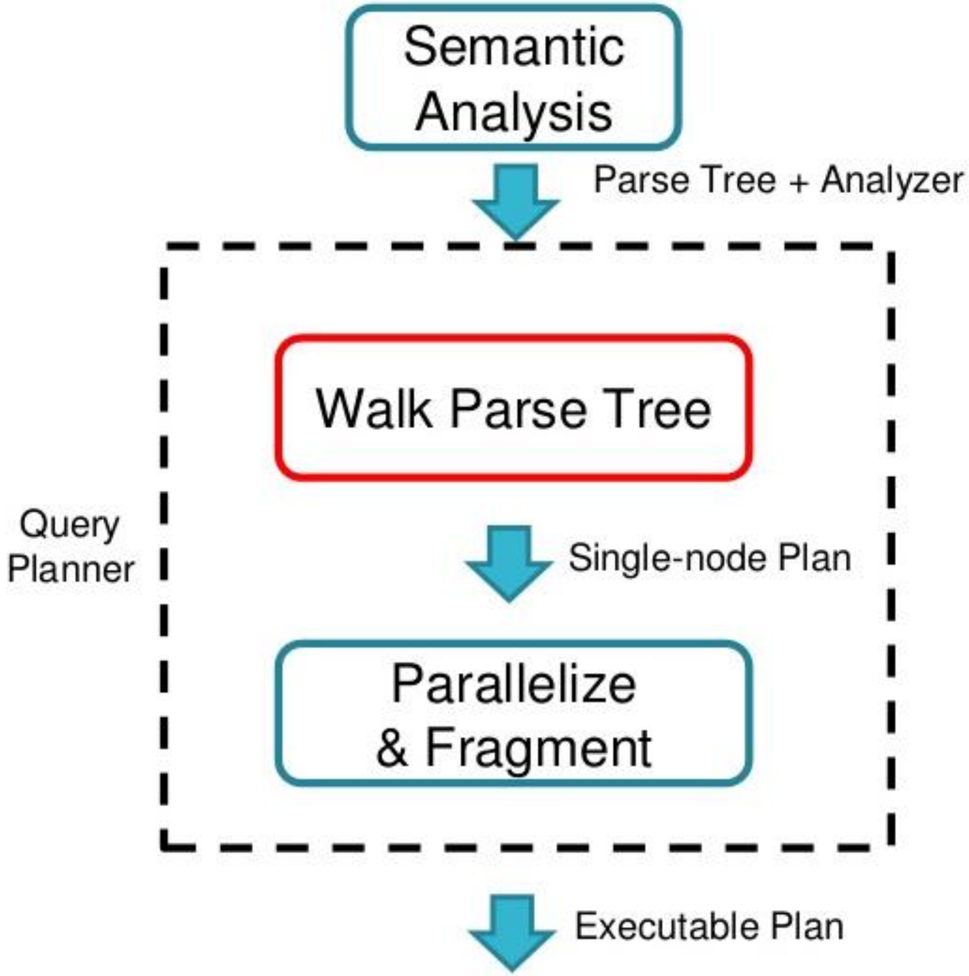
```
SELECT dept, SUM(revenue)
FROM t1
WHERE month > 10
GROUP BY dept
```

- Preparation for Planning
  - Register state in analyzer for correct predicate assignment during planning
  - Register predicates (WHERE, HAVING, ON, USING, etc.)
  - Register outer-joined tables
  - Compute value-transfer graph and equivalence classes for predicate inference
- (…)
- **Postcondition: Query is valid. An executable plan can be produced.**

Query Planning: Goals

- **Generates executable plan ("tree" of operators)**
  - Maximize scan locality using DataNode block metadata
  - Minimize data movement
- **Full distribution of operators**
- **Query operators**
  - Scan, HashJoin, HashAggregation, Union, TopN, Exchange

Query Planning: Overview

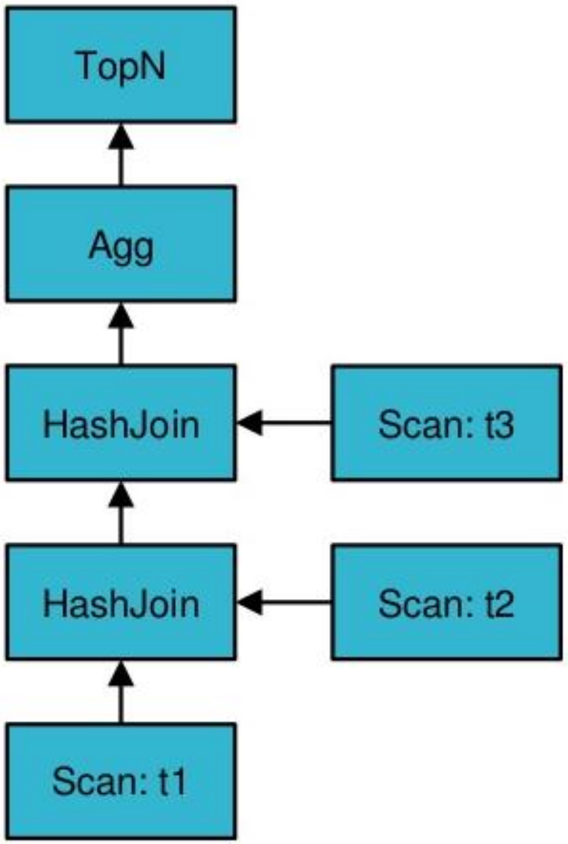


Query Planning: Single-Node Plan

- **Four major functions:**
  - 1. Parse Tree -> Plan Tree
  - 2. Assigns predicates to lowest plan node
  - Optimizes join order
  - Prunes irrelevant columns

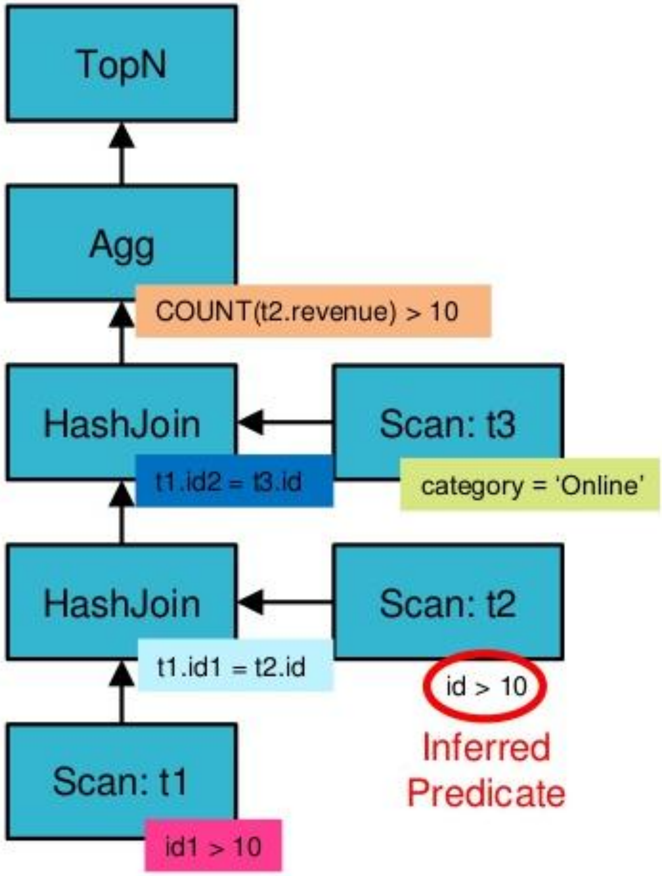
Parse Tree → Single-Node Plan Tree

SELECT t1.dept, SUM(t2.revenue)
FROM LargeHdfsTable t1
JOIN HugeHdfsTable t2 **ON** (t1.id1 = t2.id)
JOIN SmallHbaseTable t3 **ON** (t1.id2 = t3.id)
WHERE t3.category = 'Online‘ **AND** t1.id > 10
GROUP BY t1.dept
HAVING COUNT(t2.revenue) > 10
ORDER BY revenue **LIMIT** 10



Predicate Assignment & Inference

```
SELECT t1.dept, SUM(t2.revenue)
FROM LargeHdfsTable t1
JOIN HugeHdfsTable t2 ON (t1.id1 = t2.id)
JOIN SmallHbaseTable t3 ON (t1.id2 = t3.id)
WHERE t3.category = 'Online' AND t1.id > 10
GROUP BY t1.dept
HAVING COUNT(t2.revenue) > 10
ORDER BY revenue LIMIT 10
```



Join-Order Optimization

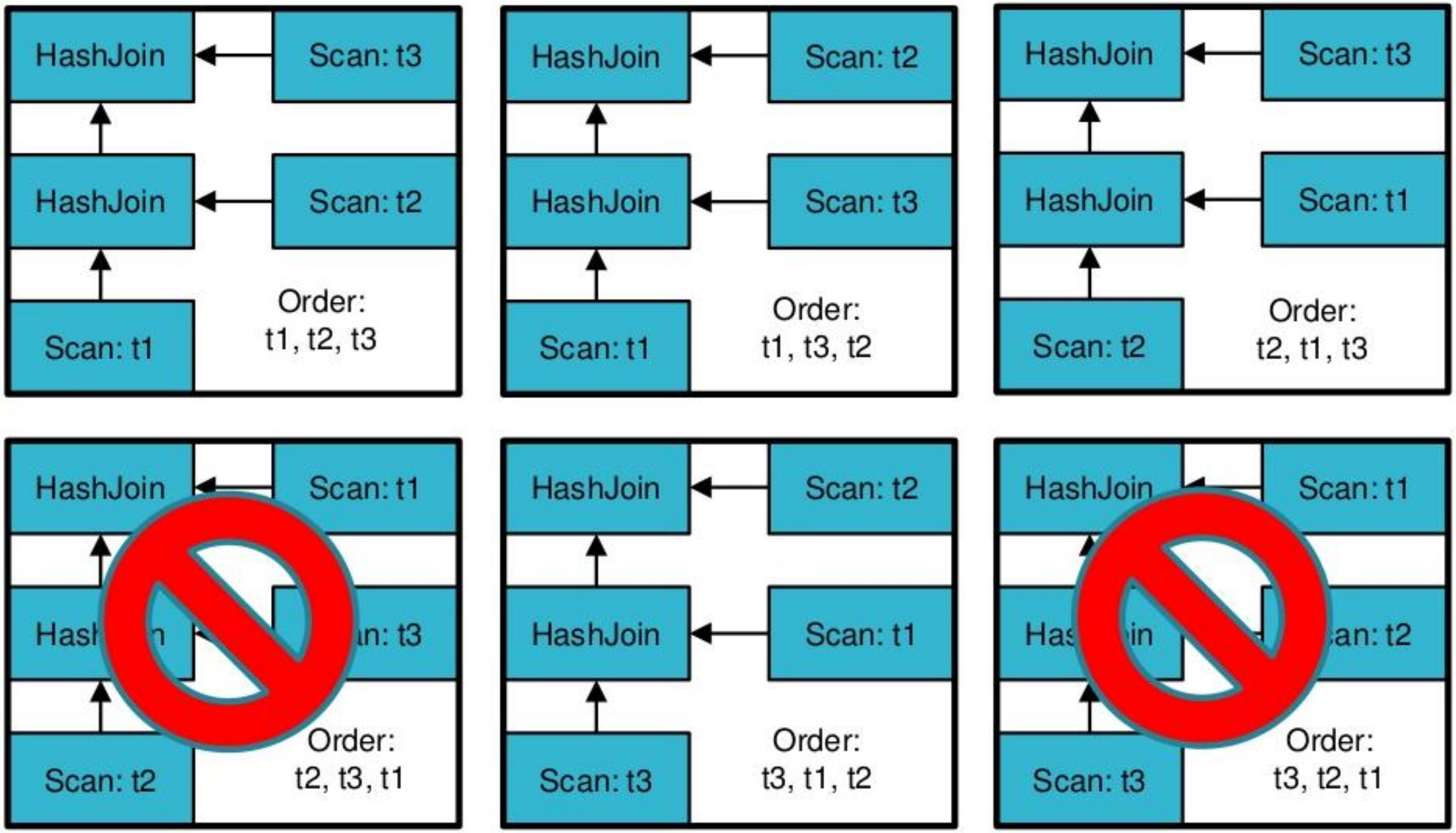
- Impala only considers left-deep join trees
- (Right join input is a table, not another join)
- Find cheapest valid join order
- Relies heavily on table and column statistics

Invalid Join Orders

```
SELECT t1.dept, SUM(t2.revenue)
FROM LargeHdfsTable t1
JOIN HugeHdfsTable t2 ON (t1.id1 = t2.id)
JOIN SmallHbaseTable t3 ON (t1.id2 = t3.id)
WHERE t3.category = 'Online' AND t1.id > 10
GROUP BY t1.dept
HAVING COUNT(t2.revenue) > 10
ORDER BY revenue LIMIT 10
```

No explicit or implicit predicate between t2 and t3

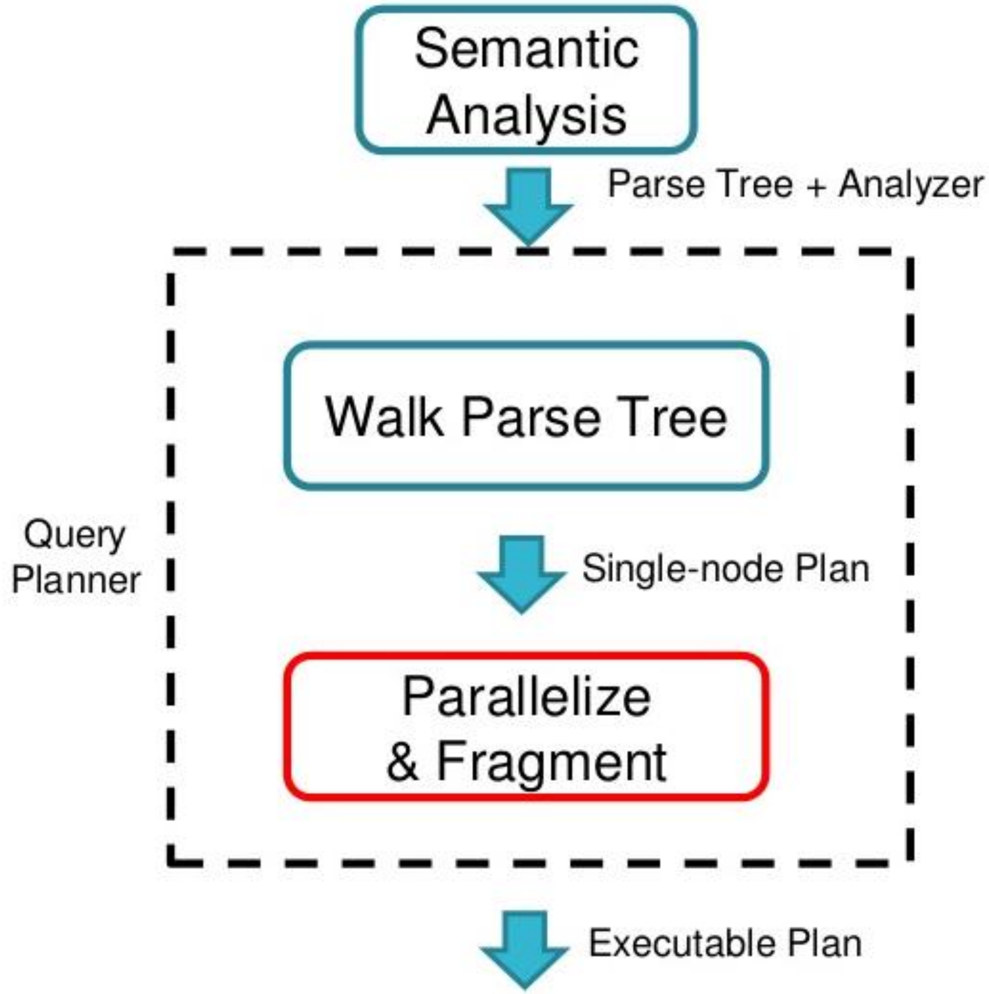
Join-Order Optimization



Join-Order Optimization

- **Impala’ s Implementation:**
- **1. Heuristic**
  - Order tables descending by size
  - Best plan typically has largest table on the left (if valid)
- **2. Plan enumeration & costing**
  - Generates all possible join orders starting from a given left-most table (starting with largest one)
  - Ignore invalid join orders
  - *Estimates intermediate result sizes (key!)*
  - Chooses plan that minimizes intermediate result sizes

Query Planning: Overview



Query Planning: Distributed Plans

- **Goals:**
  - maximize scan locality, minimize data movement
  - full distribution of all query operators
- **Parallel joins:**
  - broadcast join: join is collocated with left input, right-hand side is broadcast to each node executing the 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 and 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 where data is first materialized
  - final top-N in single-node plan fragment

Two Types of Hash Joins

- **Default is BROADCAST (aka Replicated)**
  - Each node ends up with a copy of the right table(s)
  - Left side, read locally and streamed through local hash join
  - Good for one large table and multiple small tables
- **Alternative hash join type is SHUFFLE (aka partitioned)**
  - Right side hashed and shuffled; each node gets 1/N of the data
  - Left side hashed and shuffled, then streamed through join
  - Best choice for large_table JOIN large_table

Join Hint

```
select …
from large_table
join [broadcast] small_table
```

```
select …
from large_table
join [shuffle] large_table
```

Determine Join Type from EXPLAIN

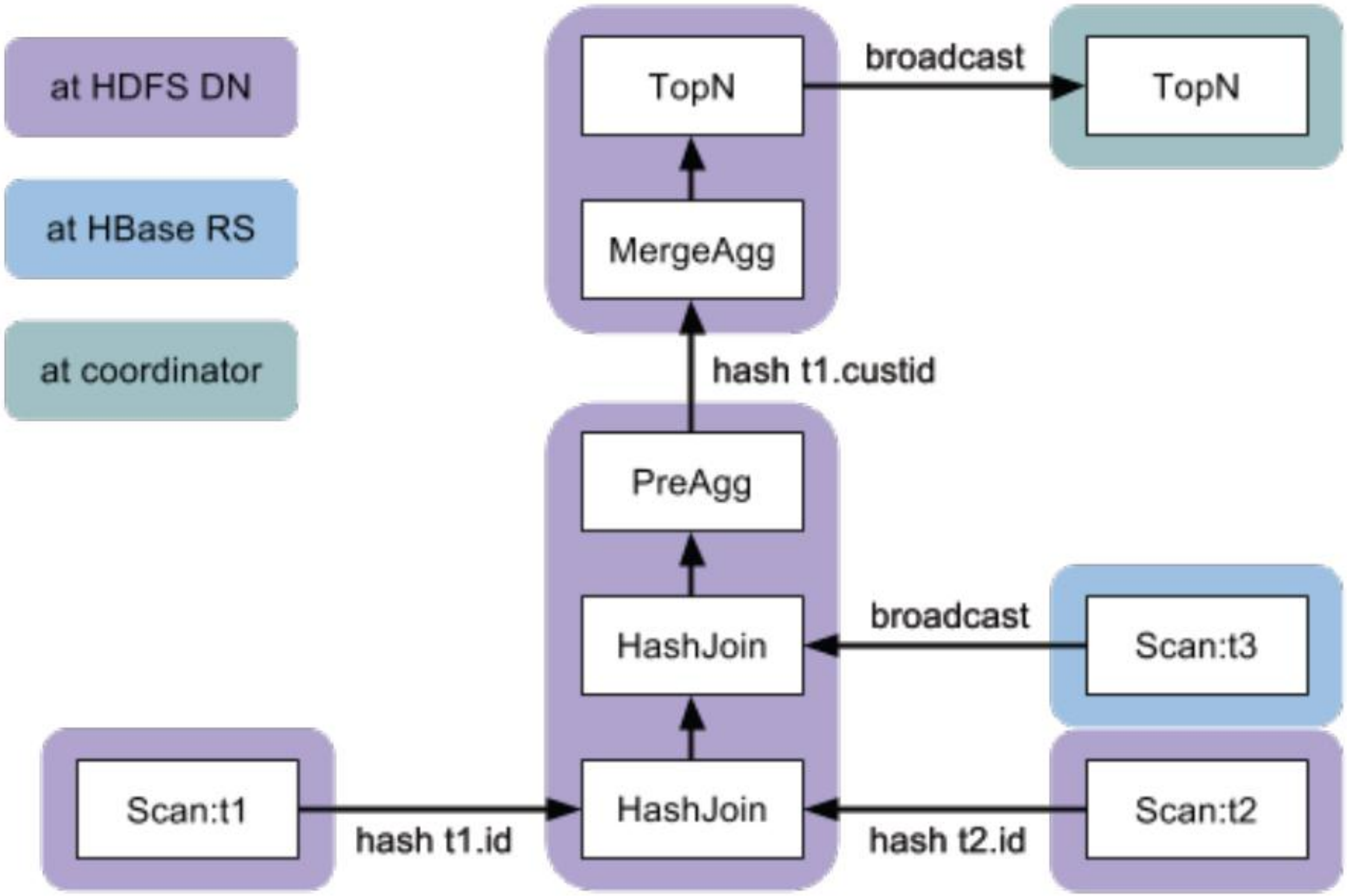
```
explain
select
  s_state,
  count(*)
from store_sales
join store on (ss_store_sk = s_store_sk)
group by
  s_state;

2:HASH JOIN
|  join op: INNER JOIN (BROADCAST)
|  hash predicates:
|    ss_store_sk = s_store_sk
|  tuple ids: 0 1
|
|----4:EXCHANGE
|       tuple ids: 1
|
0:SCAN HDFS
  table=tpcds.store_sales
  tuple ids: 0
```

```
explain
select
  c_preferred_cust_flag,
  count(*)
from store_sales
join customer on (ss_customer_sk = c_customer_sk)
group by
  c_preferred_cust_flag;

2:HASH JOIN
|  join op: INNER JOIN (PARTITIONED)
|  hash predicates:
|    ss_customer_sk = c_customer_sk
|  tuple ids: 0 1
|
|----5:EXCHANGE
|       tuple ids: 1
|
4:EXCHANGE
tuple ids: 0
```

Query Planning: Distributed Plans



HDFS Improvement Motivated by Impala

- Exposes HDFS block replica disk location information
- Allows for explicitly co-located block replicas across files
- In-memory caching of hot tables/files
- Reduces copies during reading, short-circuit reads

Disk Location of Block Replica

- **Problem:**
  - DataNode knows which DataNode blocks are on, not which disks
  - Only the DNs are aware of block replica->disk mapping
- **Impala wants to make sure that separate plan fragments operate on data on separate disks**
  - Maximizes aggregate available disk throughput

Disk Location of Block Replica

- **Solution:**
  - Adds new RPC call to DataNode to expose which volumes (disks) replicas are stored on
  - During query planning phase, impalad…
    - Determines all DNs data for query is stored on
    - Queries these DNs to get volume information
  - During query execution phase, impalad…
    - Queues disk reads so that only 1 or 2 reads ever happen to a given disk at a given time
  - With this additional information, Impala is able to ensure disk reads are large, minimizing seeks

Co-located Block Replicas

- **Problem:**
  - When performing a join, a single impalad may have to read from both a local file and a remote file on another DN
  - Ideally all reads should be done on local disks (assuming that local read is faster than remote read)

Co-located Block Replicas

- **Solution:**
  - Adds features to HDFS to specify that a set of files should have their replicas placed on the same set of nodes
  - Gives Impala more control of data
  - Can ensure that tables/files which are joined frequently have their data co-located
  - Additionally, more fine-grained block placement control allows for potential improvement in columnar storage format like Parquet

In-memory Caching

- **Problem:**
  - Impala queries are IO-bound
- **Memory is fast and getting cheaper**

In-memory Caching

- **Solution:**
  - Adds facility to HDFS to explicitly read specific HDFS files into memory
  - Allows Impala to read data at full memory bandwidth speed
  - Gives cluster operator control over which files/tables are queried frequently and thus could be kept in memory

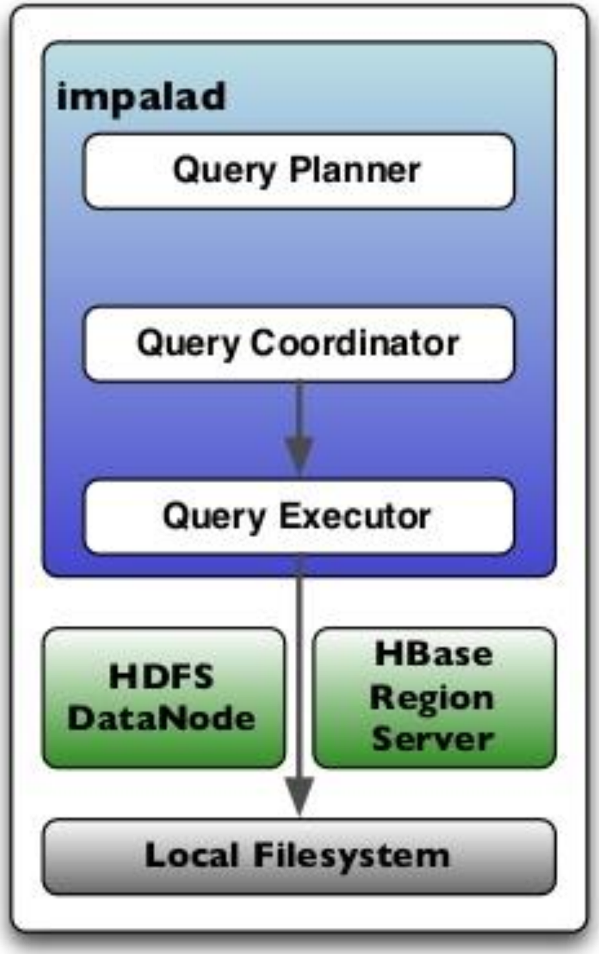
Short-circuit Reads

- **Problem:**
  - A typical read in HDFS must be read from disk by DataNode, copied into DN memory, sent over network, copied into client buffers.

Short-circuit Reads

- **Solution:**
  - Reads are performed directly on local files, using direct buffers
  - In HDFS, allow for reads to completely bypass DataNode when client is co-located with block replica files, avoiding overhead of HDFS API
  - Reads data directly from disk to client buffers

Read directly from disk



Code Generation

Why Code Generation?

Why Code Generation?

SPEED!

Why Code Generation?

- **Code generation (codegen) lets us use query-specific information to do less work**
  - Remove conditionals
  - Propagate constant offsets, pointers, etc.
  - Inline virtual function calls

```
void MaterializeTuple(char* tuple) {
  for (int i = 0; i < num_slots_; ++i) {
    char* slot = tuple + offsets_[i];
    switch(types_[i]) {
      case BOOLEAN:
        *slot = ParseBoolean();
        break;
      case INT:
        *slot = ParseInt();
        break;
      case FLOAT: …
      case STRING: …
      // etc.
    }
  }
}
```

interpreted

```
void MaterializeTuple(char* tuple) {
  *(tuple + 0) = ParseInt();     // i = 0
  *(tuple + 4) = ParseBoolean(); // i = 1
  *(tuple + 5) = ParseInt();     // i = 2
}
```

codegen'd

```
void MaterializeTuple(char* tuple) {
  for (int i = 0; i < num_slots_; ++i) {
    char* slot = tuple + offsets_[i];
    switch(types_[i]) {
      case BOOLEAN:
        *slot = ParseBoolean();
        break;
      case INT:
        *slot = ParseInt();
        break;
      case FLOAT: …
      case STRING: …
      // etc.
    }
  }
}
```

interpreted

```
void MaterializeTuple(char* tuple) {
  *(tuple + 0) = ParseInt();     // i = 0
  *(tuple + 4) = ParseBoolean(); // i = 1
  *(tuple + 5) = ParseInt();     // i = 2
}
```

codegen'd

```
void MaterializeTuple(char* tuple) {
  for (int i = 0; i < num_slots_; ++i) {
    char* slot = tuple + offsets_[i];
    switch(types_[i]) {
      case BOOLEAN:
        *slot = ParseBoolean();
        break;
      case INT:
        *slot = ParseInt();
        break;
      case FLOAT: …
      case STRING: …
      // etc.
    }
  }
}
```

interpreted

```
void MaterializeTuple(char* tuple) {
  *(tuple + 0) = ParseInt();     // i = 0
  *(tuple + 4) = ParseBoolean(); // i = 1
  *(tuple + 5) = ParseInt();     // i = 2
}
```

codegen'd

```
void MaterializeTuple(char* tuple) {
  for (int i = 0; i < num_slots_; ++i) {
    char* slot = tuple + offsets_[i];
    switch(types_[i]) {
      case BOOLEAN:
        *slot = ParseBoolean();
        break;
      case INT:
        *slot = ParseInt();
        break;
      case FLOAT: …
      case STRING: …
      // etc.
    }
  }
}
```

interpreted

```
void MaterializeTuple(char* tuple) {
  *(tuple + 0) = ParseInt();     // i = 0
  *(tuple + 4) = ParseBoolean(); // i = 1
  *(tuple + 5) = ParseInt();     // i = 2
}
```

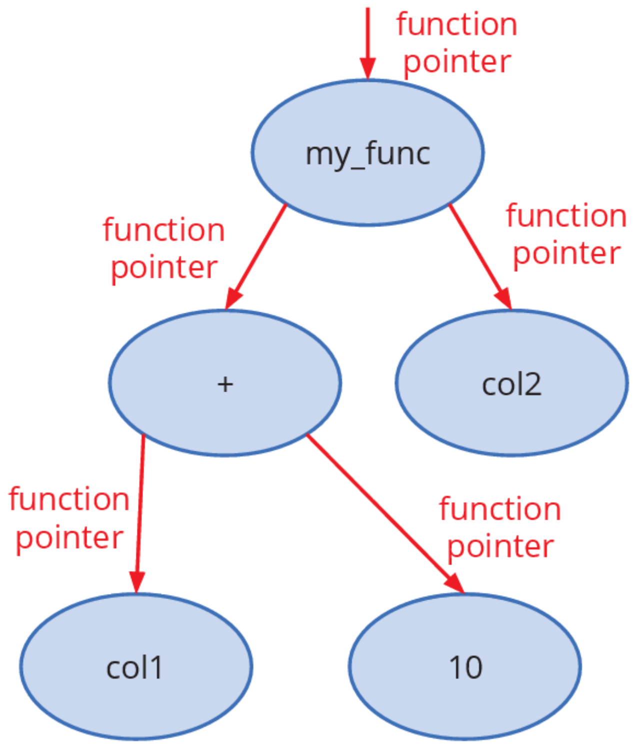
codegen'd

User-Defined Functions (UDFs)

- Allows users to extend Impala’s functionality by writing their own functions
- e.g. select my_func(c1) from table;
- Defined as C++ functions
- UDFs can be compiled to LLVM IR with Clang ⇒ inline UDFs
- IR can be just-in-time compiled (JIT’d) and replace the interpreted functions

```
IntVal my_func(const IntVal& v1, const IntVal& v2) {
  return IntVal(v1.val * 7 / v2.val);
}
```

```
SELECT my_func(col1 + 10, col2) FROM ...
```



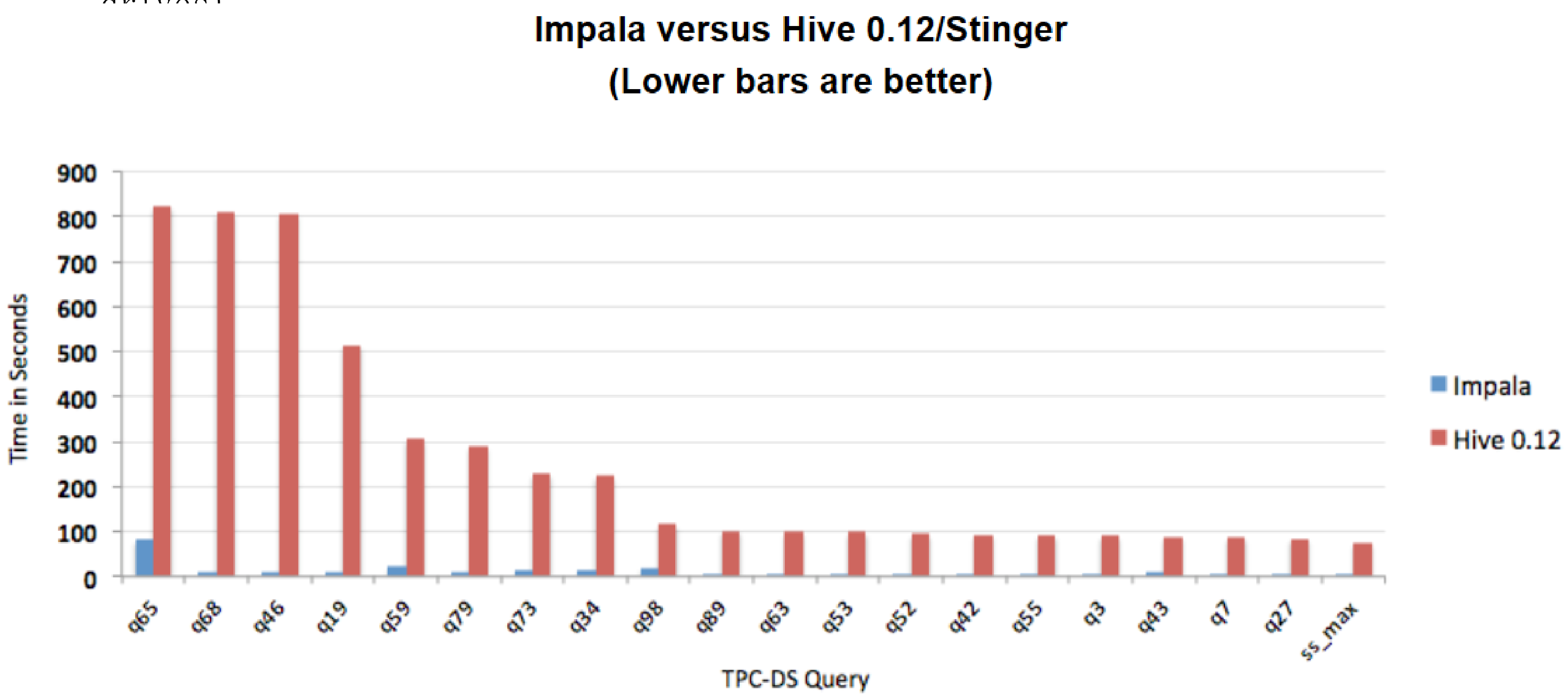
interpreted

(col1 + 10) * 7 / col2

codegen'd

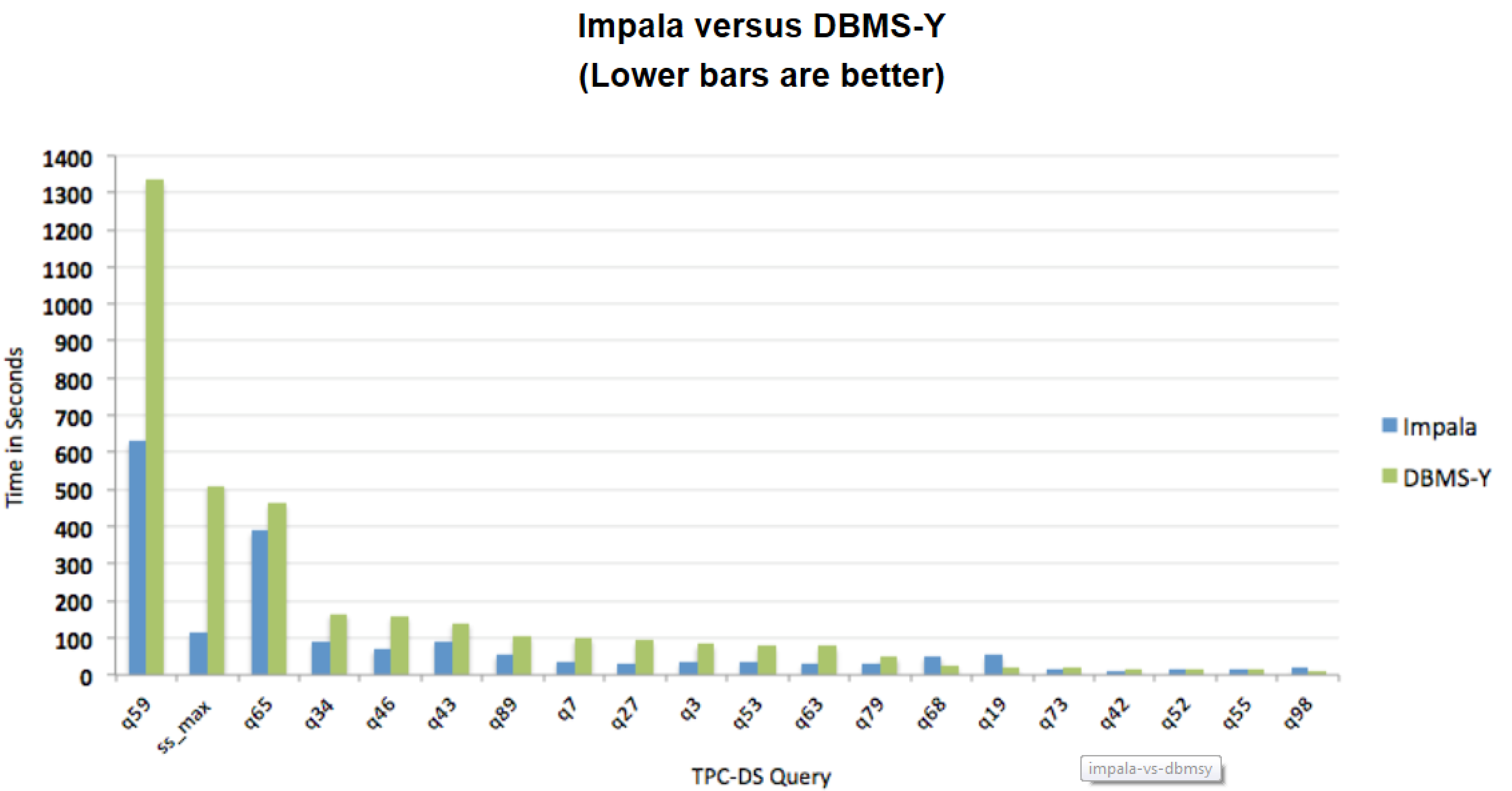
Performance (Jan 2014)

- 3TB (TPC-DS scale factor 3,000) across five typical Hadoop DataNodes (dual-socket, 8-core, 16-thread CPU; 96GB memory; 1Gbps Ethernet; 12 x 2TB disk drives)



Performance (Jan 2014)

- 30TB set of TPC-DS data (scale factor 30,000), 20 nodes with 96GB memory per node



Weaknesses and Limitations

- Data is immutable, no updating
- Response time is not microsecond
- Do not support some operations, like update and delete
- No beyond SQL and advanced data structures (buckets, samples, transforms, arrays, structs, maps, xpath, json)
- When broadcast join, smaller table has to fit in aggregate memory of all executing nodes
- No custom storage format
- LIMIT required when using ORDER BY
- High memory usage

References

- Cloudera Impala official documentation and slides