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

Marcel Kornacker
Cloudera, Inc.

Agenda

- Goals; user view of Impala
- Impala performance
- Impala internals
- Comparing Impala to other systems

Impala Overview: Goals

- General-purpose SQL query engine:
  - works both for analytical and transactional/single-row workloads
  - supports queries that take from milliseconds to hours
- Runs directly within Hadoop:
  - reads widely used Hadoop file formats
  - talks to widely used Hadoop storage managers
  - runs on same nodes that run Hadoop processes
- High performance:
  - C++ instead of Java
  - runtime code generation
  - completely new execution engine that doesn't build on MapReduce

User View of Impala: Overview

- Runs as a distributed service in cluster: one Impala daemon on each node with data
- User submits query via ODBC/JDBC, Impala CLI or Hue to any of the daemons
- Query is distributed to all nodes with relevant data
- If any node fails, the query fails
- Impala uses Hive's metadata interface, connects to Hive's metastore
- Supported file formats:
  - Parquet columnar format (more on that later)
  - sequence files and RCfile with snappy/gzip compression
  - Avro data files
  - uncompressed/lzo-compressed text files

User View of Impala: SQL

- SQL support:
  - patterned after Hive's version of SQL
  - essentially SQL-92, minus correlated subqueries
  - INSERT INTO ... SELECT ...
  - only equi-joins; no non-equi joins, no cross products
  - ORDER BY requires LIMIT
  - Limited DDL support
- Functional limitations:
  - no custom UDFs, file formats, SerDes
  - no beyond SQL (buckets, samples, transforms, arrays, structs, maps, xpath, json)
  - only hash joins; joined table has to fit in aggregate memory of all executing nodes

User View of Impala: HBase

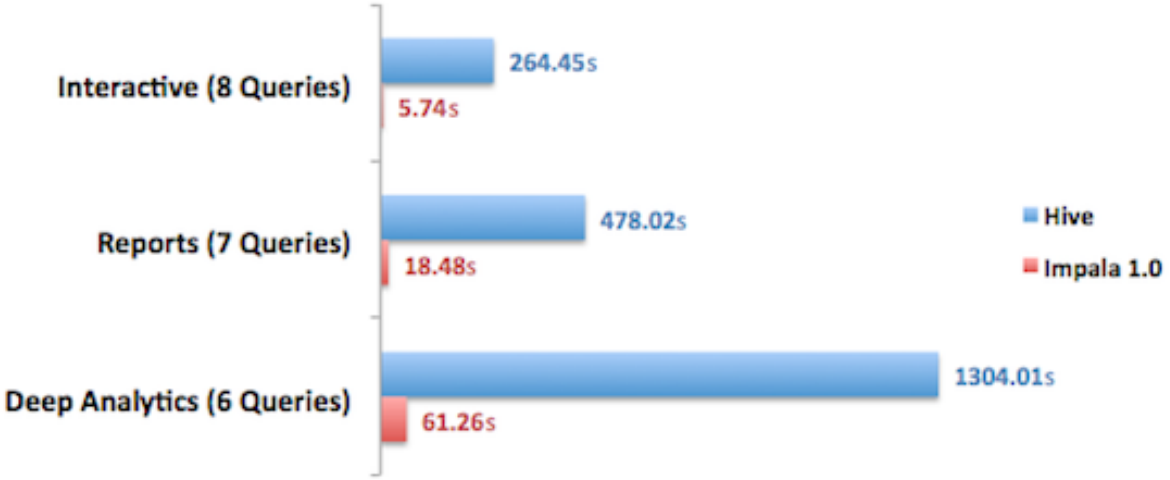
- Functionality highlights:
  - support for SELECT, INSERT INTO ... SELECT ..., and INSERT INTO ... VALUES(...)
  - predicates on rowkey columns are mapped into start/stop row
  - predicates on other columns are mapped into SingleColumnValueFilters
- But: mapping of HBase table into metastore table patterned after Hive
  - all data stored as scalars and in ascii
  - the rowkey needs to be mapped to a single string column

User View of Impala: HBase

- Roadmap:
  - full support for UPDATE and DELETE
  - storage of structured data to minimize storage and access overhead
  - composite row key encoding, mapped into an arbitrary number of table columns

Impala Single-User Performance

- Benchmark: 20 queries from TPC-DS, in 3 categories:
  - interactive: 1 month
  - Reports: several months
  - deep analytics: all data
- Main fact table: 5 years of data, 1TB, stored as snappy-compressed sequence files
- Cluster: 20 machines, 24 cores each

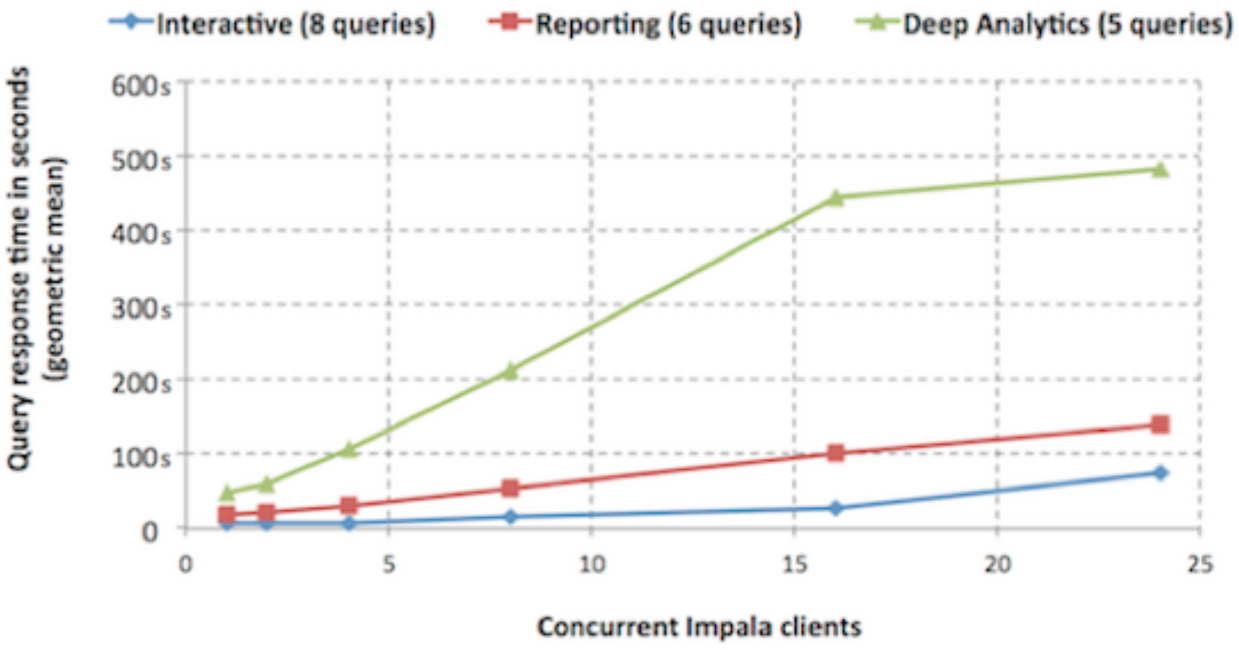


Impala Single-User Performance

- Speed-up over Hive:
  - interactive: 25x - 68x
  - Reports: 6x - 56x
  - deep analytics: 6x - 55x

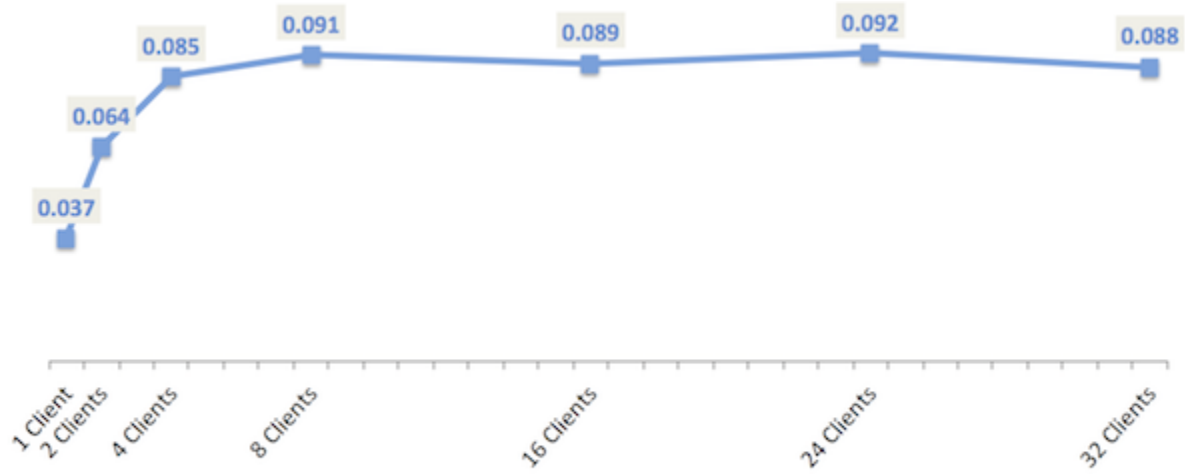
Impala Multi-User Performance

- Benchmark for query latency in multi-user env:
  - same dataset and workload as single-user benchm.
  - same hardware config
  - multiple clients issue queries in parallel



Impala Multi-User Performance

- Query throughput (in queries per second) in multi-user environment:
  - scaling up workload (not # of machines)
  - qps increases until cluster is saturated
  - qps stable at that point, system doesn't waste work



Impala Architecture

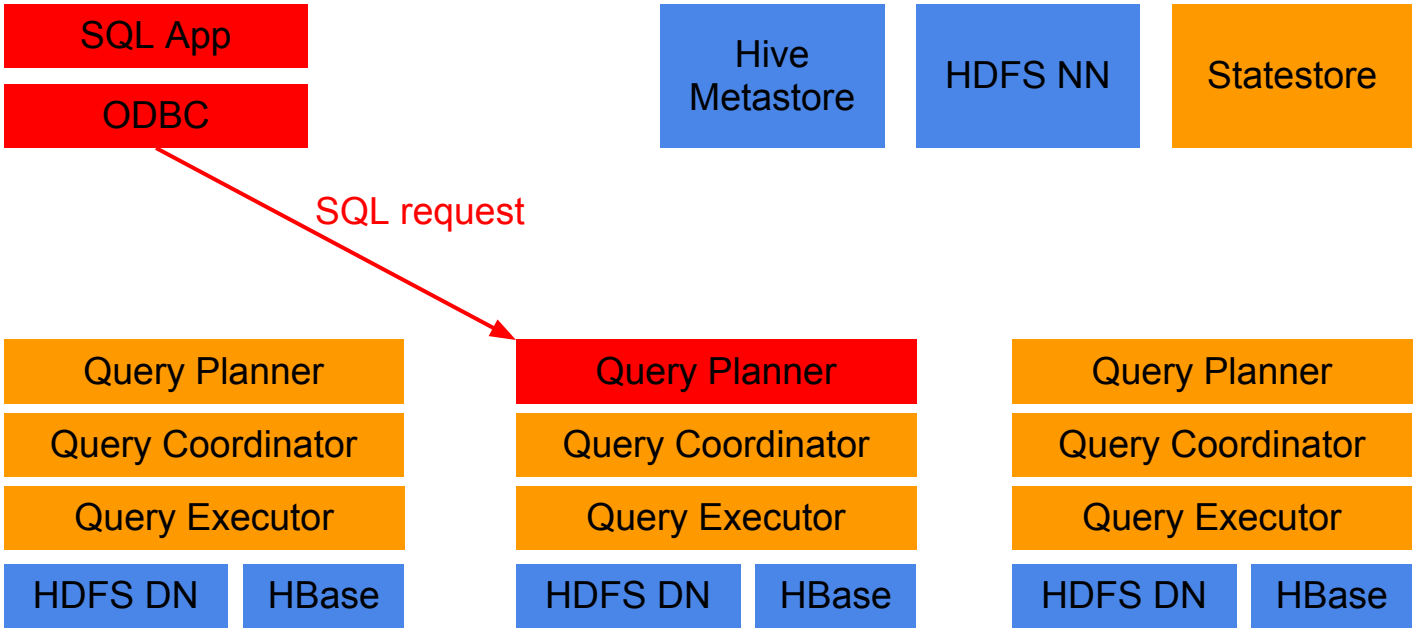
- Two binaries: impalad and statestored
- Impala daemon (impalad) - N instances
  - handles client requests and all internal requests related to query execution
- State store daemon (statestored) - 1 instance
  - provides name service and metadata distribution

Impala Architecture

- Query execution phases
  - request arrives via odbc/jdbc
  - planner turns request into collections of plan fragments
  - coordinator initiates execution on remote impalad nodes
- 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)

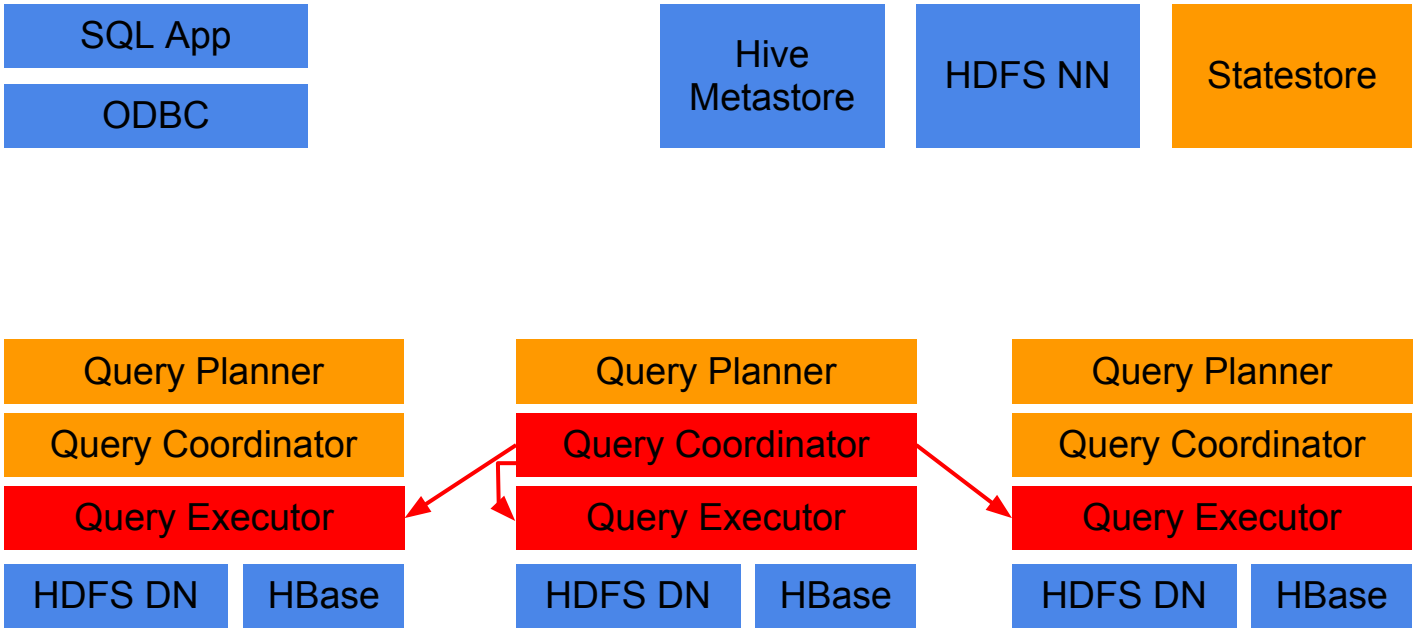
Impala Architecture: Query Execution

Request arrives via odbc/jdbc



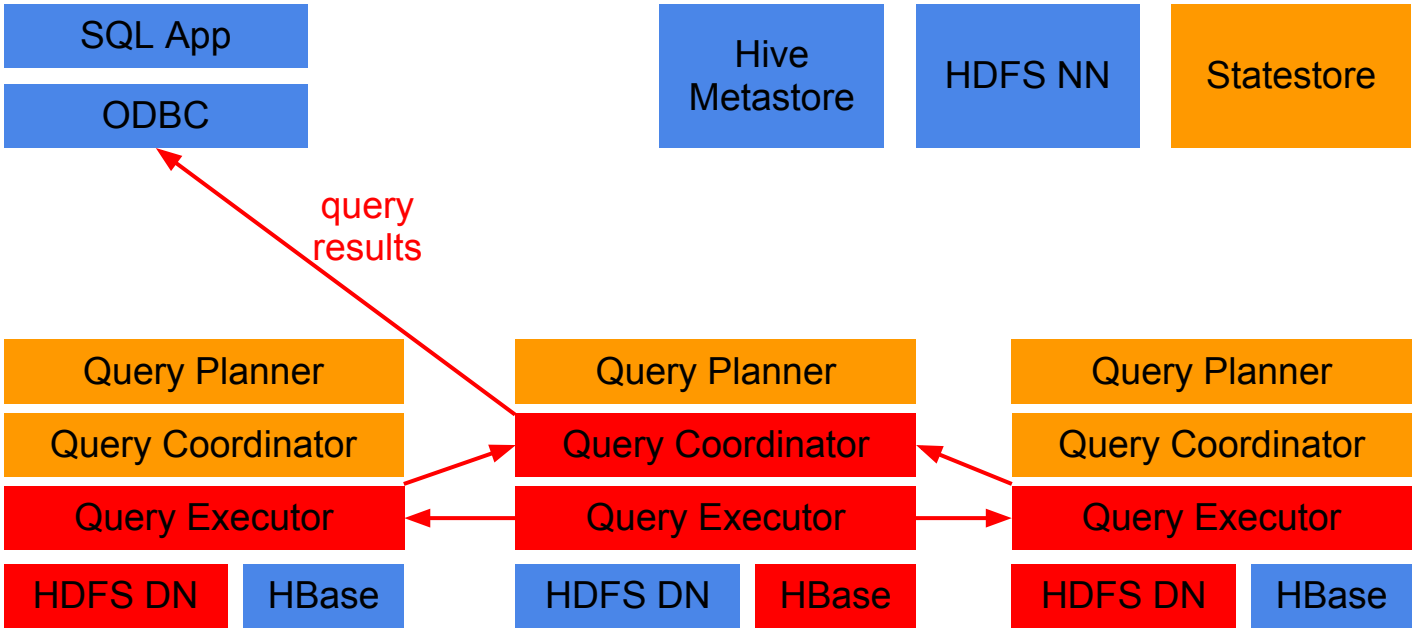
Impala Architecture: Query Execution

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



Impala Architecture: 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:
  - all query operators are fully distributed
- Join order = FROM clause order

  Post-GA: cost-based optimizer

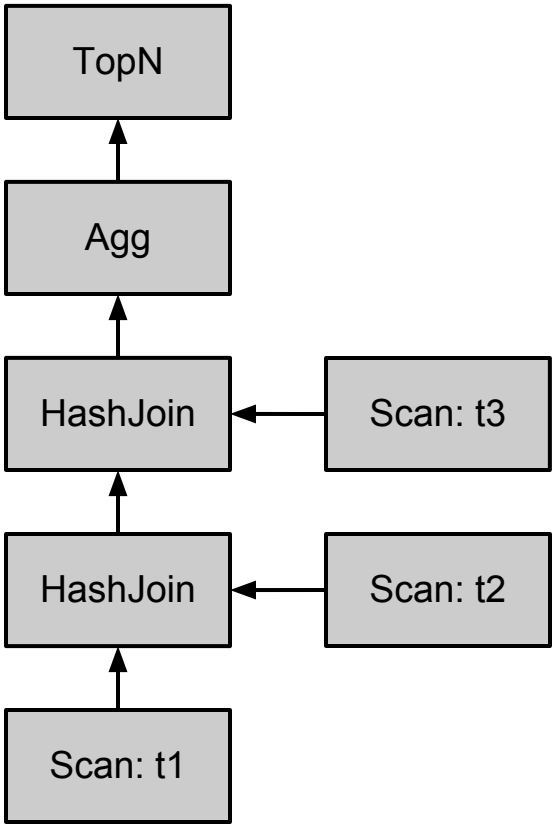
Query Planning: Single-Node Plan

- Plan operators: Scan, HashJoin, HashAggregation, Union, TopN, Exchange
- Example:

```
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: Single-Node Plan

- Single-node plan for example:



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

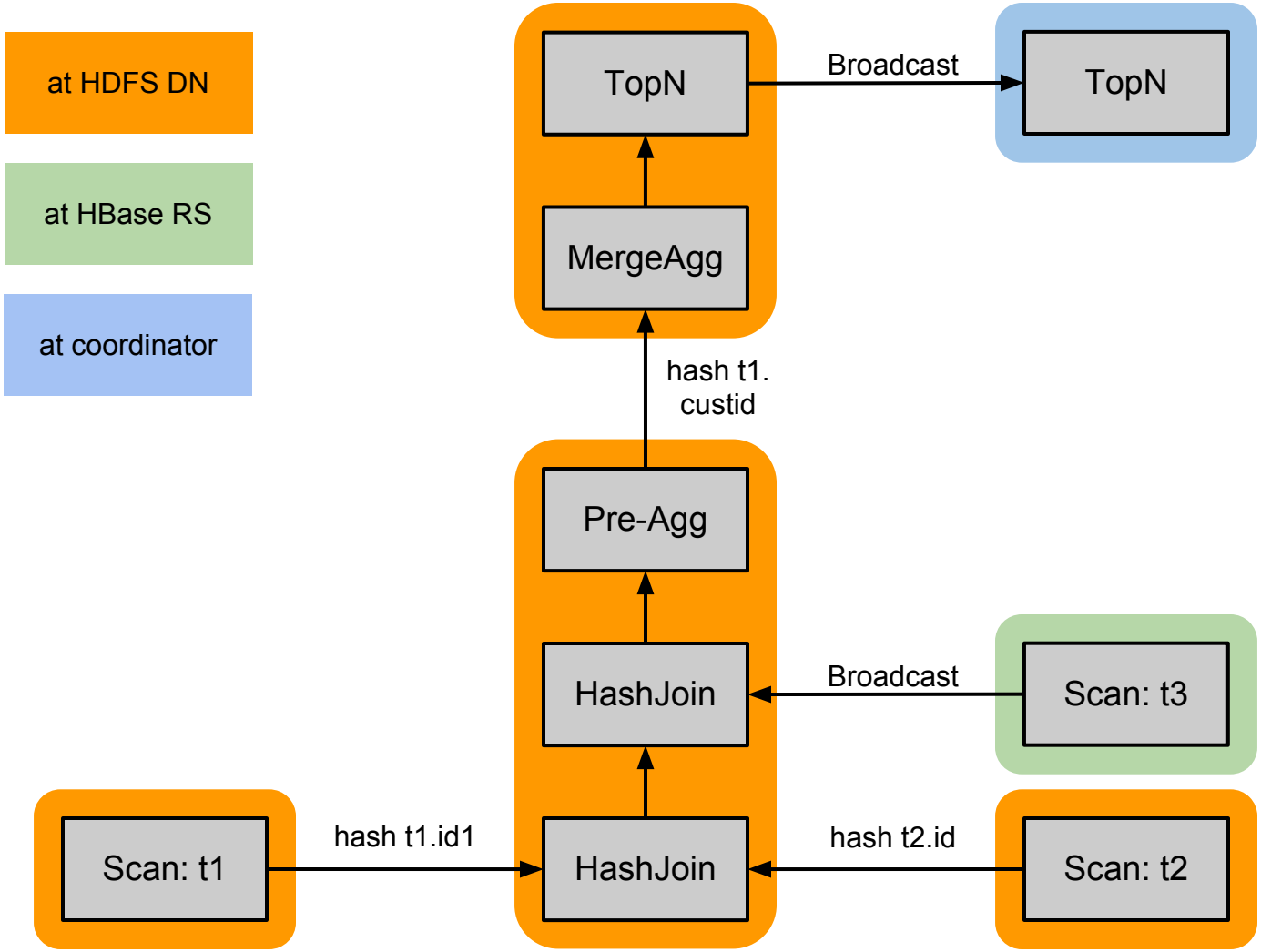
Query Planning: Distributed Plans

- 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

- In the 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



Metadata Handling

- Impala metadata:
  - Hive's metastore: logical metadata (table definitions, columns, CREATE TABLE parameters)
  - HDFS NameNode: directory contents and block replica locations
  - HDFS DataNode: block replicas' volume ids
- Caches metadata: no synchronous metastore API calls during query execution
- impalad instances read metadata from metastore at startup
- REFRESH [<tbl>]: selectively reloads metadata at single impalad instance
- Post-GA: metadata distribution through statestore
- Post-GA: HCatalog and AccessServer

Impala Execution Engine

- Written in C++ for minimal execution overhead
- Internal in-memory tuple format puts fixed-width data at fixed offsets
- Uses intrinsics/special cpu instructions for text parsing, crc32 computation, etc.
- Runtime code generation for "big loops"

Impala Execution Engine

- More on runtime code generation
  - example of "big loop": insert batch of rows into hash table
  - known at query compile time: # of tuples in batch, tuple layout, column types, etc.
  - generate at compile time: loop that inlines all function calls, contains no dead code, minimizes branches
  - code generated using llvm

Impala's Statestore

- Central system state repository
  - name service (membership)
  - Post-GA: metadata
  - Post-GA: other scheduling-relevant or diagnostic state
- 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

Statestore: Why not ZooKeeper

- ZK is not a good pub-sub system
  - Watch API is awkward and requires a lot of client logic
  - multiple round-trips required to get data for changes to node's children
  - push model is more natural for our use case
- Don't need all the guarantees ZK provides:
  - serializability
  - persistence
  - prefer to avoid complexity where possible
- ZK is bad at the things we care about and good at the things we don't

Comparing Impala to Dremel

- What is Dremel:
  - columnar storage for data with nested structures
  - distributed scalable aggregation on top of that
- Columnar storage in Hadoop: Parquet
- Distributed aggregation: Impala
- Impala plus Parquet: a superset of the published version of Dremel (which didn't support joins)

More about Parquet

- What it is:
  - columnar container format for all popular serialization formats: Avro, Thrift, Protocol Buffers
  - successor to Doug Cutting's Trevni
  - co-designed by Cloudera and Twitter
  - open source; hosted on github
- Features:
  - fully shredded nested data; repetition and definition levels similar to Dremel's ColumnIO
  - column values stored in native types (bool, int<x>, float, double, byte array)
  - support for index pages for fast lookup
  - extensible value encodings (run-length encoding, dictionary, ...)

Comparing Impala to Hive

- Hive: MapReduce as an execution engine
  - High latency, low throughput queries
  - Fault-tolerance model based on MapReduce's on-disk checkpointing; materializes all intermediate results
  - Java runtime allows for easy late-binding of functionality: file formats and UDFs.
  - Extensive layering imposes high runtime overhead
- Impala:
  - direct, process-to-process data exchange
  - no fault tolerance
  - an execution engine designed for low runtime overhead

Impala Roadmap: 2013

- Additional SQL:
  - UDFs
  - SQL authorization and DDL
  - ORDER BY without LIMIT
  - analytic/window functions
  - support for structured data types
- Improved HBase support:
  - composite keys, complex types in columns, index nested-loop joins, INSERT/UPDATE/DELETE
- Runtime optimizations:
  - straggler handling
  - join order optimization
  - improved cache management
  - data collocation for improved join performance

Impala Roadmap: 2013

- Better metadata handling:
  - automatic metadata distribution through statestore
- Resource management:
  - goal: run exploratory and production workloads in same cluster, against same data, without impacting production jobs

Try it out!

- Impala 1.0 was released on 04/30
- We have packages for:
  - RHEL6.2/5.7
  - Ubuntu 10.04 and 12.04
  - SLES11
  - Debian6
- Questions/comments? impala-user@cloudera.org
- My email address: marcel@cloudera.com