Impala: A Modern SQL Engine for Hadoop

James Kinley, Solutions Architect
Cloudera, Inc.

Agenda

- Goals
- User view of Impala
- Impala internals
- Comparing Impala to other systems

Impala Overview: Goals

- **General-purpose SQL query engine:**
  - works for both analytical and transactional workloads
  - will support 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 (LLVM)
  - 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 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:**
  - uncompressed / lzo-compressed text files
  - sequence files and RCFile with snappy/gzip compression
  - Avro data files
  - Parquet columnar format (more on that later)

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 (CREATE TABLE)
- **Functional limitations:**
  - no custom UDFs, file formats, or SerDes
  - no beyond SQL (buckets, samples, transforms, arrays, structs, maps, xpath, json)
  - only hash joins; joined table has to fit in memory:
    - beta: of single node (broadcast)
    - GA: aggregate memory of all executing nodes (partitioned)

User View of Impala: Apache HBase

- **HBase functionality:**
  - uses Hive's mapping of HBase table into metastore table
  - predicates on rowkey columns are mapped into start/stop row
  - predicates on other columns are mapped into SingleColumnValueFilters
- **HBase functional limitations:**
  - no nested-loop joins
  - all data stored as text

Impala Architecture

- **Two binaries: impalad and statestored**
- **Impala daemon (impalad)**
  - handles client requests and all internal requests related to query execution
  - exports Thrift services for these two roles
- **State store daemon (statestored)**
  - provides name service and metadata distribution
  - also exports a Thrift service

Impala Architecture

- **Query execution phases**
  - request arrives via ODBC/JDBC
  - planner turns request into collections of plan fragments (tree of query operators)
  - 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 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)

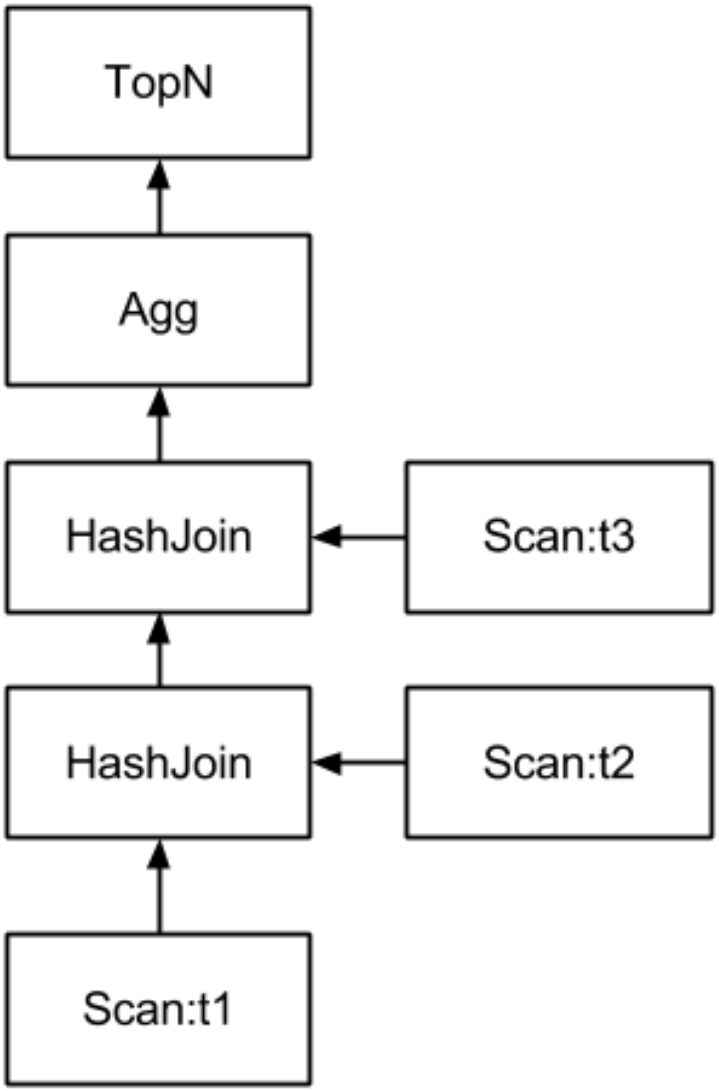
Query Planning: Example

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

```
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: Example

- Single-node plan:



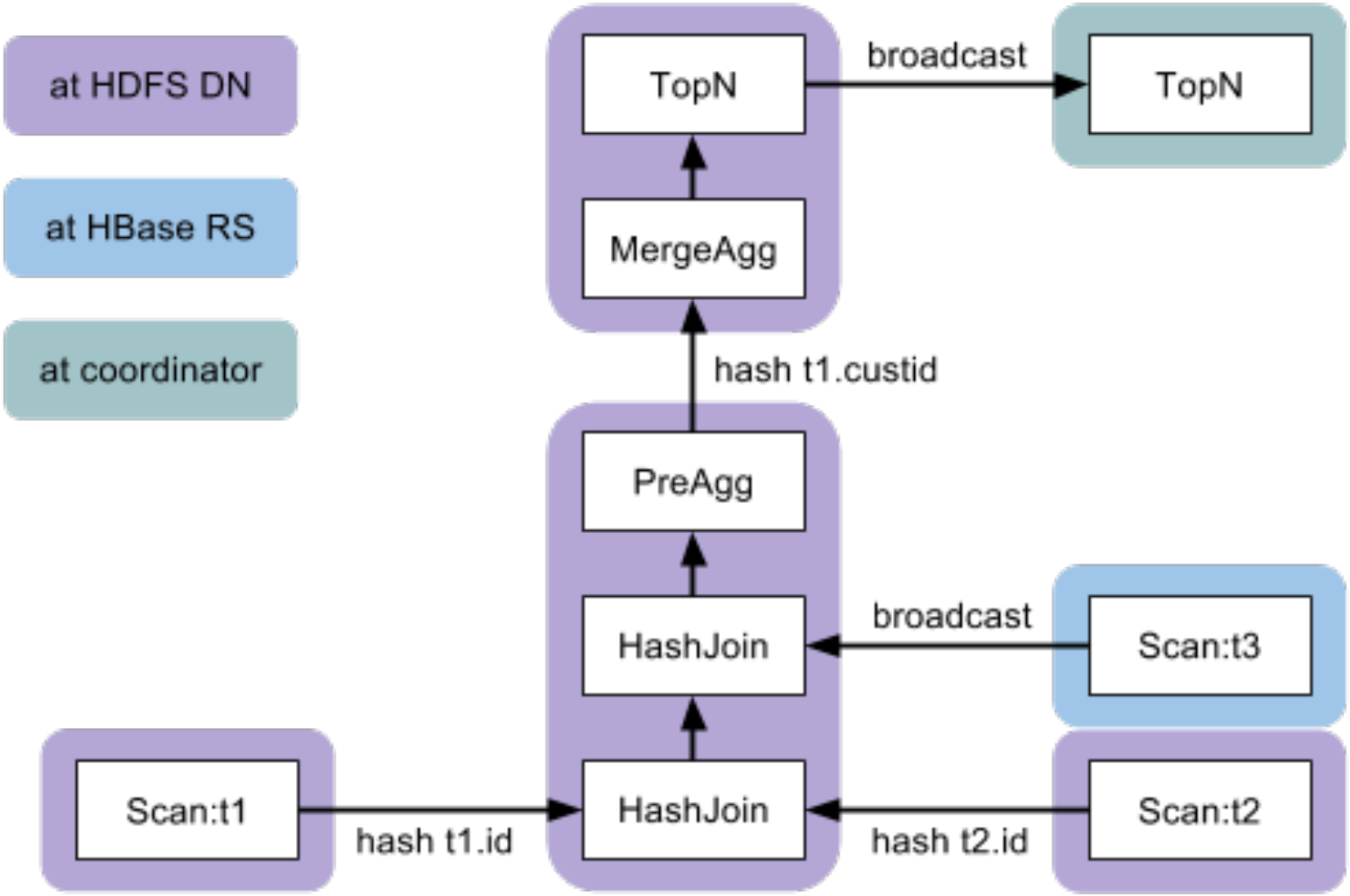
Query Planning: Distributed Plans

- **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

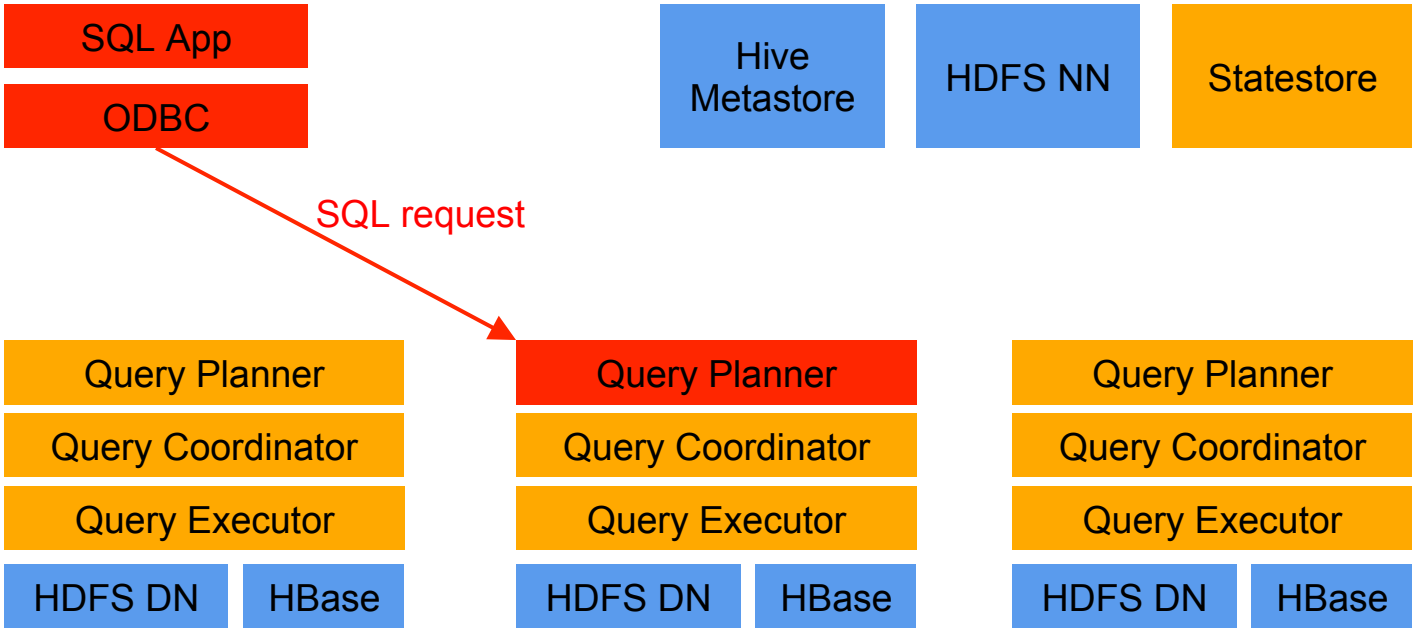
- **Parallel aggregation:**
  - pre-aggregation where data is first materialized
  - merge aggregation partitioned by grouping columns
- **Parallel top-N:**
  - initial top-N where data is first materialized
  - final top-N in single-node plan fragment

Query Planning: Distributed Plans



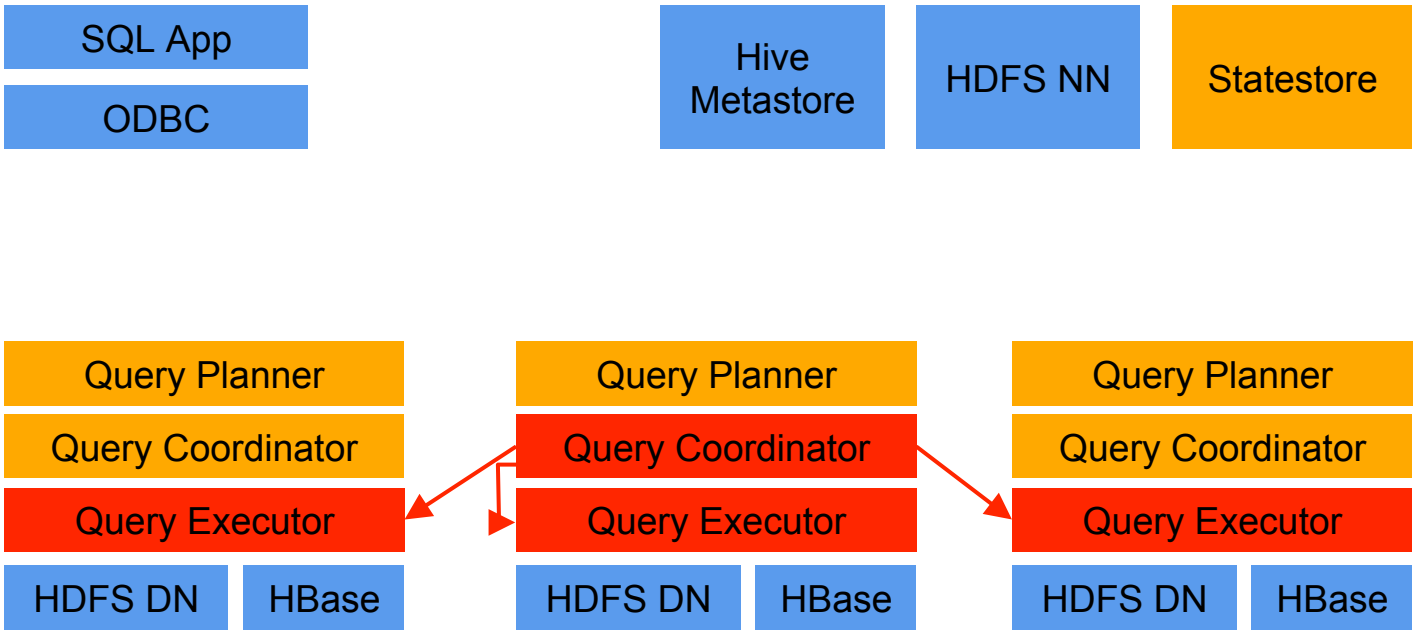
Impala Architecture: Query Execution

- Request arrives via ODBC/JDBC



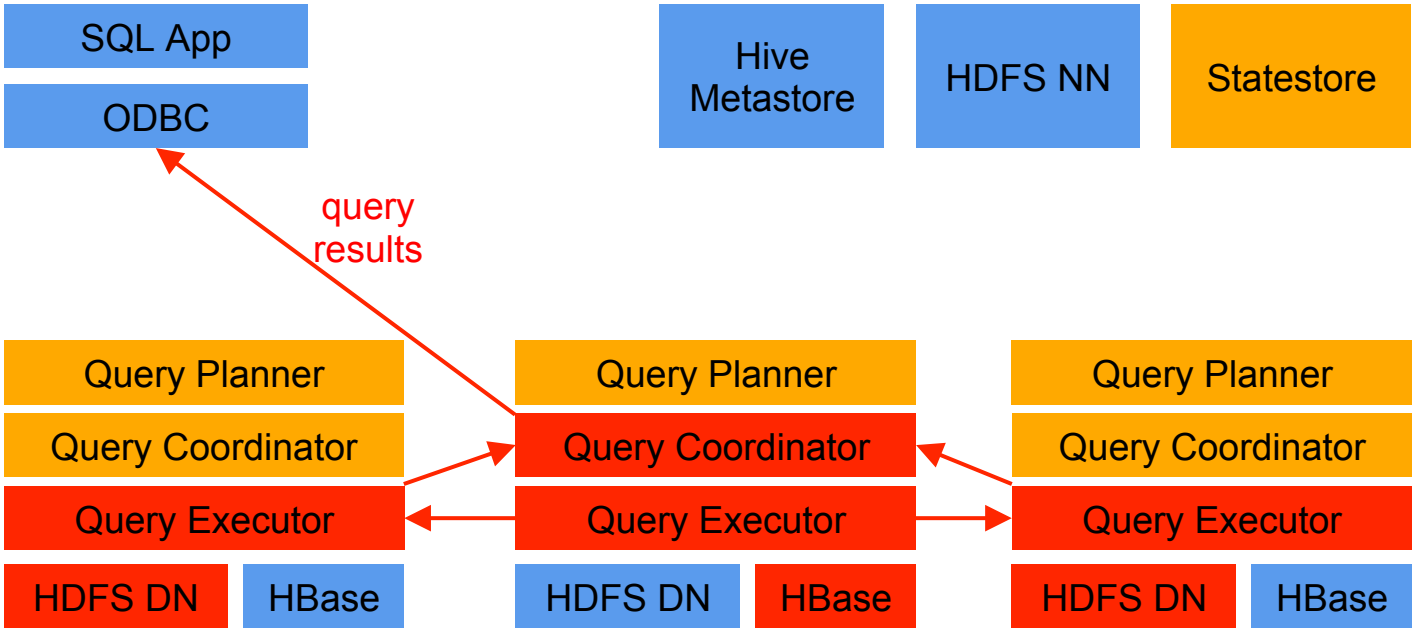
Impala Architecture: Query Execution

- Planner turns request into collections of plan fragments
- Coordinator initiates execution on remote impalad's



Impala Architecture: Query Execution

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



Metadata Handling

- Utilizes Hive's metastore
- Caches metadata: no synchronous metastore API calls during query execution
- Beta: impalad's read metadata from metastore at startup
- Post-GA: metadata distribution through statestore

Impala Execution Engine

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

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**
    - Abstract Syntax Tree to Intermediate Representation (IR)
    - IR optimization (compiler optimizations)
    - IR to machine code generation

http://blog.cloudera.com/blog/2013/02/inside-cloudera-impala-runtime-code-generation/

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

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: joint project between Cloudera and Twitter:**
  - new columnar format: Parquet;
    derived from Doug Cutting's Trevni
  - stores data in appropriate native/binary types
  - can also store nested structures similar to Dremel's ColumnIO
- **Distributed aggregation:** Impala
- **Impala plus Parquet:** a superset of the published version of Dremel (which didn't support joins)

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

Comparing Impala to Hive

- Impala's performance advantage over Hive: no hard numbers, but:
  - Impala can get full disk throughput (~100MB/sec/disk): I/O-bound workloads often faster by 3-4x
  - queries that require multiple map-reduce phases in Hive see a higher speedup
  - queries that run against in-memory data see a higher speedup (observed up to 100x)

Impala Roadmap: GA (April 2013)

- Improved query execution
  - partitioned joins
- Further performance improvements
- Guidelines for production deployment:
  - load balancing across impalad's
  - resource isolation within MR cluster

Impala Roadmap: 2013

- Additional SQL:
  - UDFs
  - SQL authorization and DDL
  - ORDER BY without LIMIT
  - window functions
  - support for structured data types
- Improved HBase support:
  - composite keys, complex types in columns, index nested-loop joins, INSERT/UPDATE/DELETE

Impala Roadmap: 2013

- Runtime optimizations:
  - straggler handling
  - join order optimization
  - improved cache management
  - data collocation for improved join performance
- 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!

- Beta version available since 24/10/12
- Latest version is 0.6 (as of 26/02);
- 0.7 will contain Parquet support
- We have packages for:
  - RHEL6.2/5.7
  - Ubuntu 10.04 and 12.04
  - SLES11
  - Debian6
- We're targeting GA for April 2013
- Questions/comments? impala-user@cloudera.org
- Email: kinley@cloudera.com, Twitter: @jrkinley