Introduction to Apache Hive and Cloudera Impala

Mark Grover
Software Engineer, Cloudera
github.com/markgrover/hive-impala-bdtc

About Me

- Contributor to Apache Hive, Impala packaging
- Section Author of O'Reilly's Programming Hive book
- Software Developer at Cloudera
- @mark_grover
- github.com/markgrover/hive-impala-bdtc

Why not just MapReduce?

- Catered to developers
  - Lot of boilerplate to do simple group by/aggregations
  - Cumbersome for analysts to use

So why not just MapReduce?

- Catered to developers
- Lot of boilerplate to do simple aggregations
- Tons of analysts out there who understand SQL

Hive

- Data warehouse system for Hadoop
- Enables Extract/Transform/Load (ETL)
- Associate structure with a variety of data formats
  - Logical Table -> Physical Location
  - Logical Table -> Physical Data Format Handler (SerDe)
- Integrates with HDFS, HBase, MongoDB, etc.
- Query execution in MapReduce

Why use Hive?

- MapReduce is catered towards developers
- Run SQL-like queries that get compiled and run as MapReduce jobs
- Data in Hadoop even though generally unstructured has some vague structure associated with it
- Benefits of MapReduce + HDFS (Hadoop)
  - Fault tolerant
  - Robust
  - Scalable

Agenda

- What is Hive?
- Why use Hive?
- Hive features
- Hive architecture
- HCatalog
- Demo!

Preamble

- This is a remote talk
- Feel free to ask questions any time!

Hive features

- Create table, create view, create index - DDL
- Select, where clause, group by, order by, joins
- Pluggable User Defined Functions - UDFs (e.g from_unixtime)
- Pluggable User Defined Aggregate Functions - UDAFs (e.g. count, avg)
- Pluggable User Defined Table Generating Functions - UDTFs (e.g. explode)

Hive features

- Pluggable custom Input/Output format
- Pluggable Serialization Deserialization libraries (SerDes)
- Pluggable custom map and reduce scripts

What Hive does NOT support

- OLTP workloads - low latency
- Correlated subqueries
- Not super performant with small amounts of data
  - How much data do you need to call it “Big Data”?

Other Hive features

- Partitioning
- Sampling
- Bucketing
- Various join optimizations
- Integration with HBase and other storage handlers
- Views – Unmaterialized
- Complex data types – arrays, structs, maps

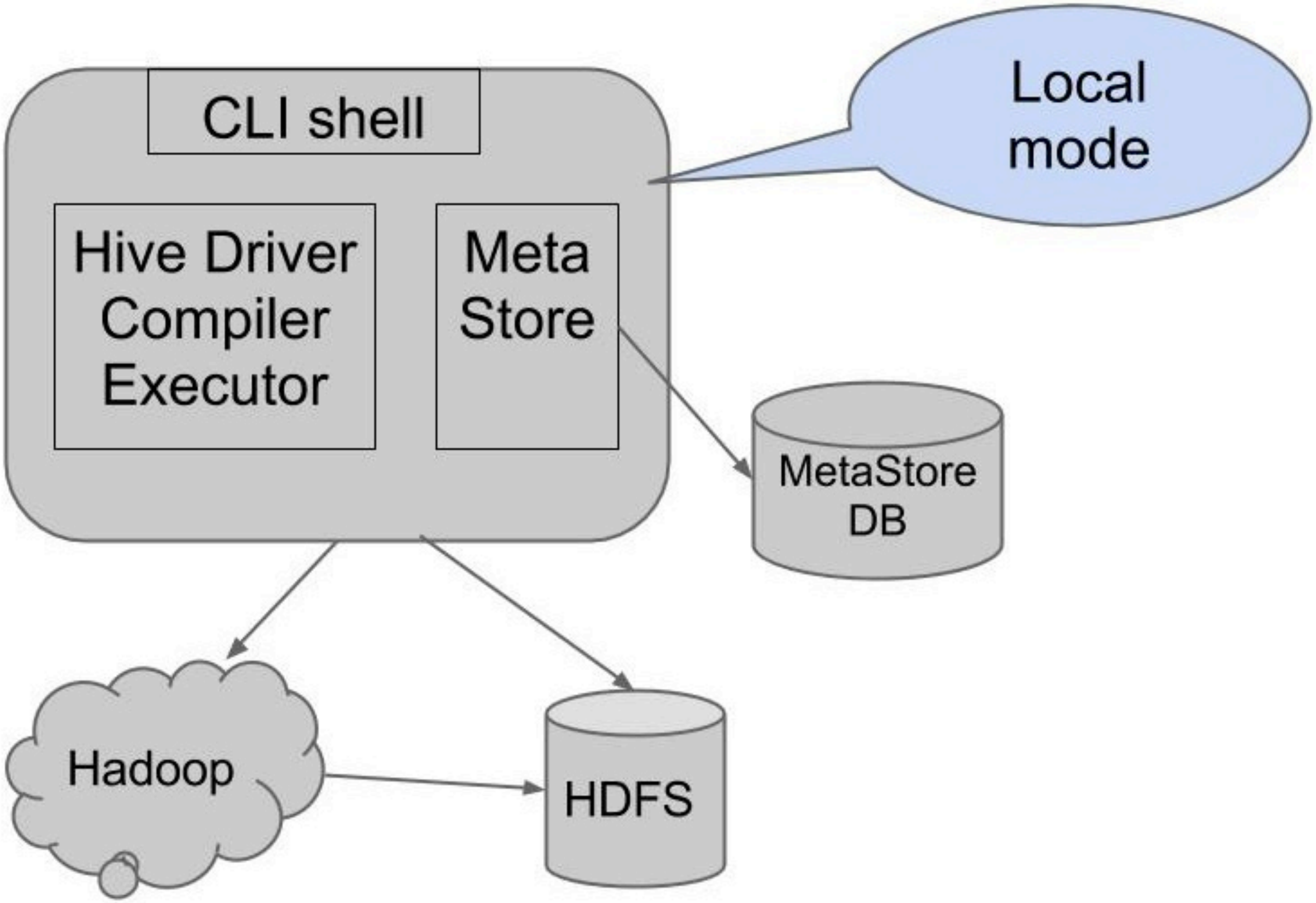
Connecting to Hive

- Hive Shell
- JDBC driver
- ODBC driver
- Thrift client

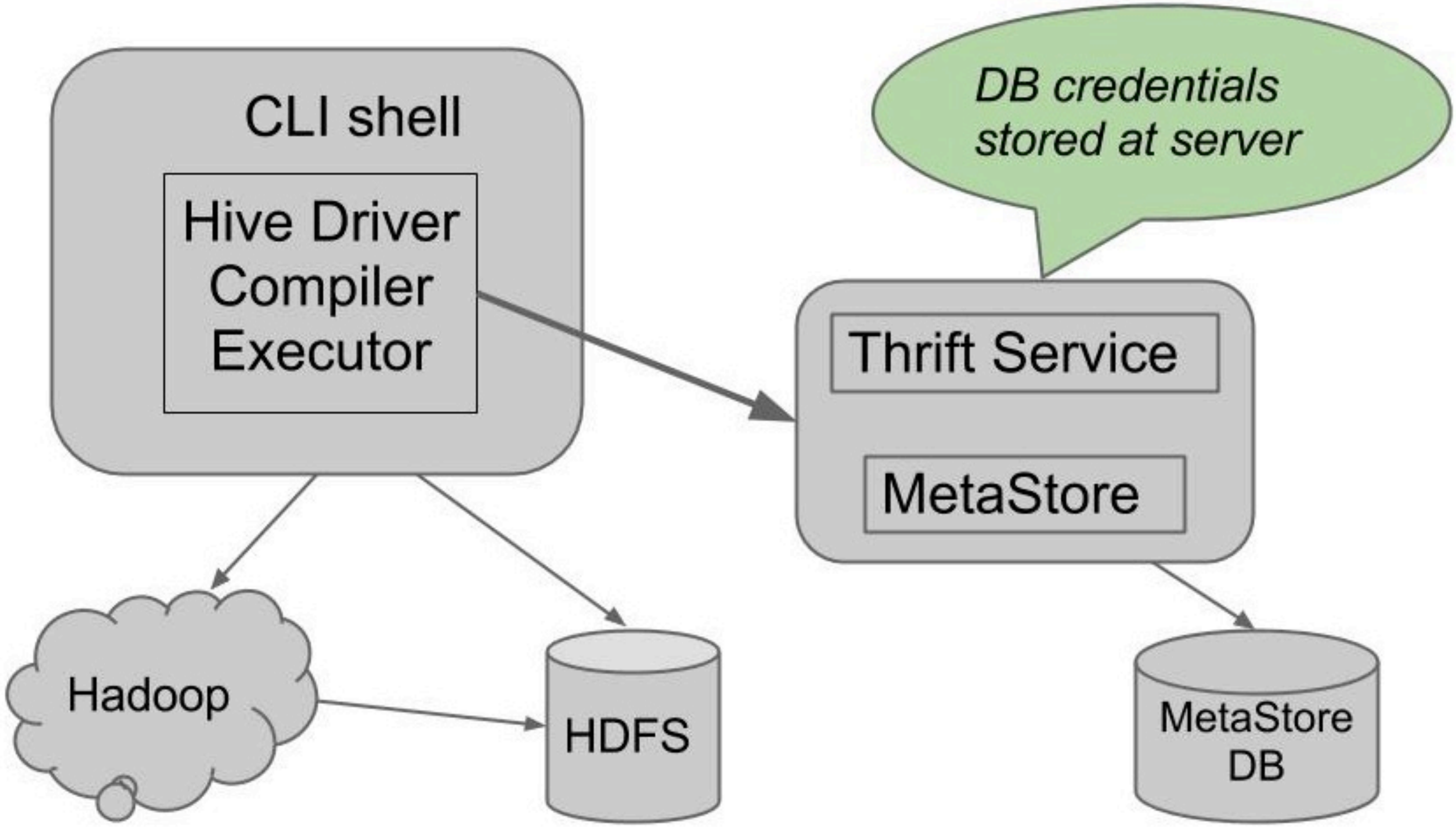
Hive metastore

- Backed by RDBMS
  - Derby, MySQL, PostgreSQL, etc. supported
- Default Embedded Derby
  - Not recommend for anything but a quick Proof of Concept
- 3 different modes of operation:
  - Embedded Derby (default)
  - Local
  - Remote

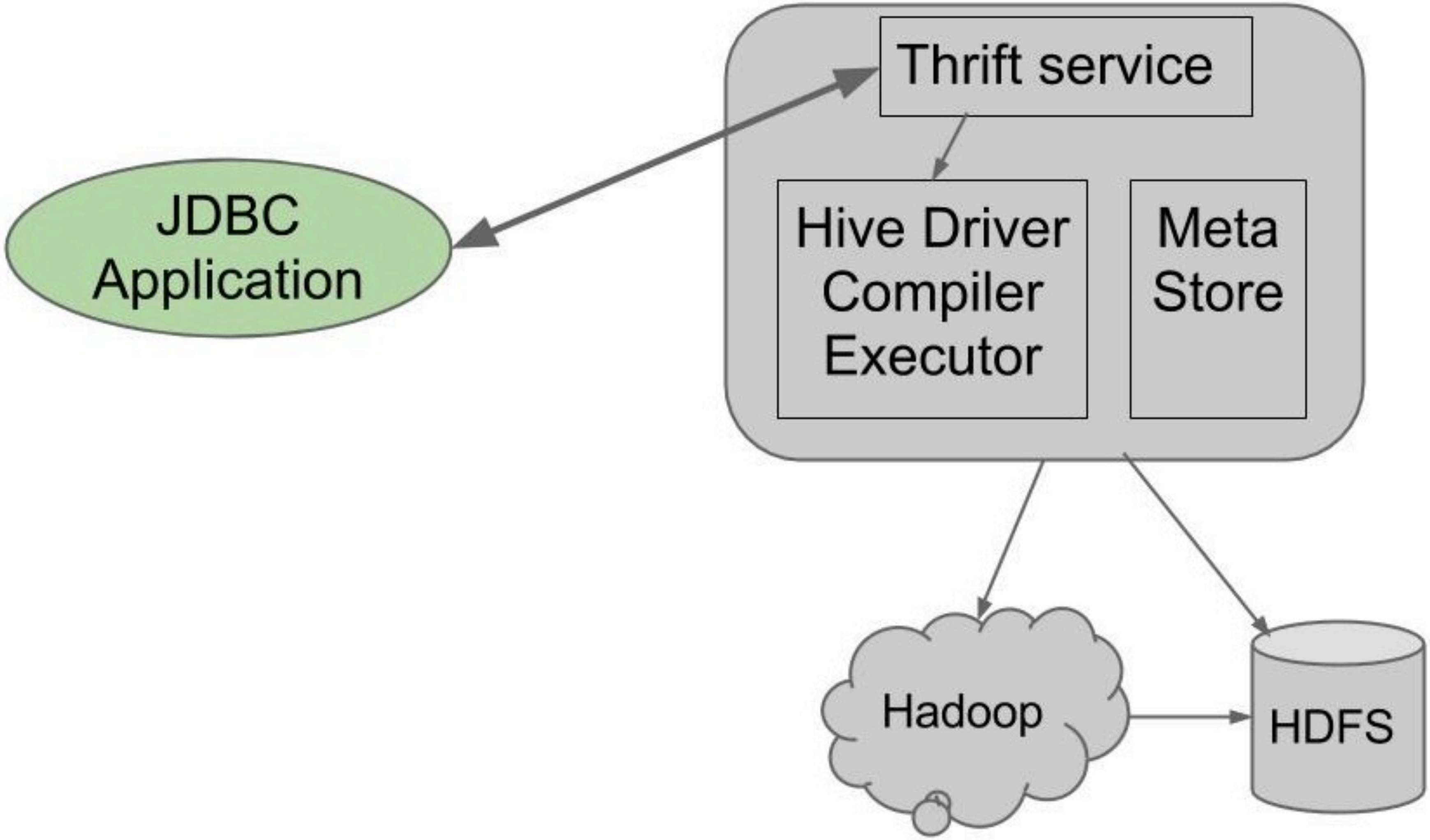
Hive architecture



Hive Remote Mode



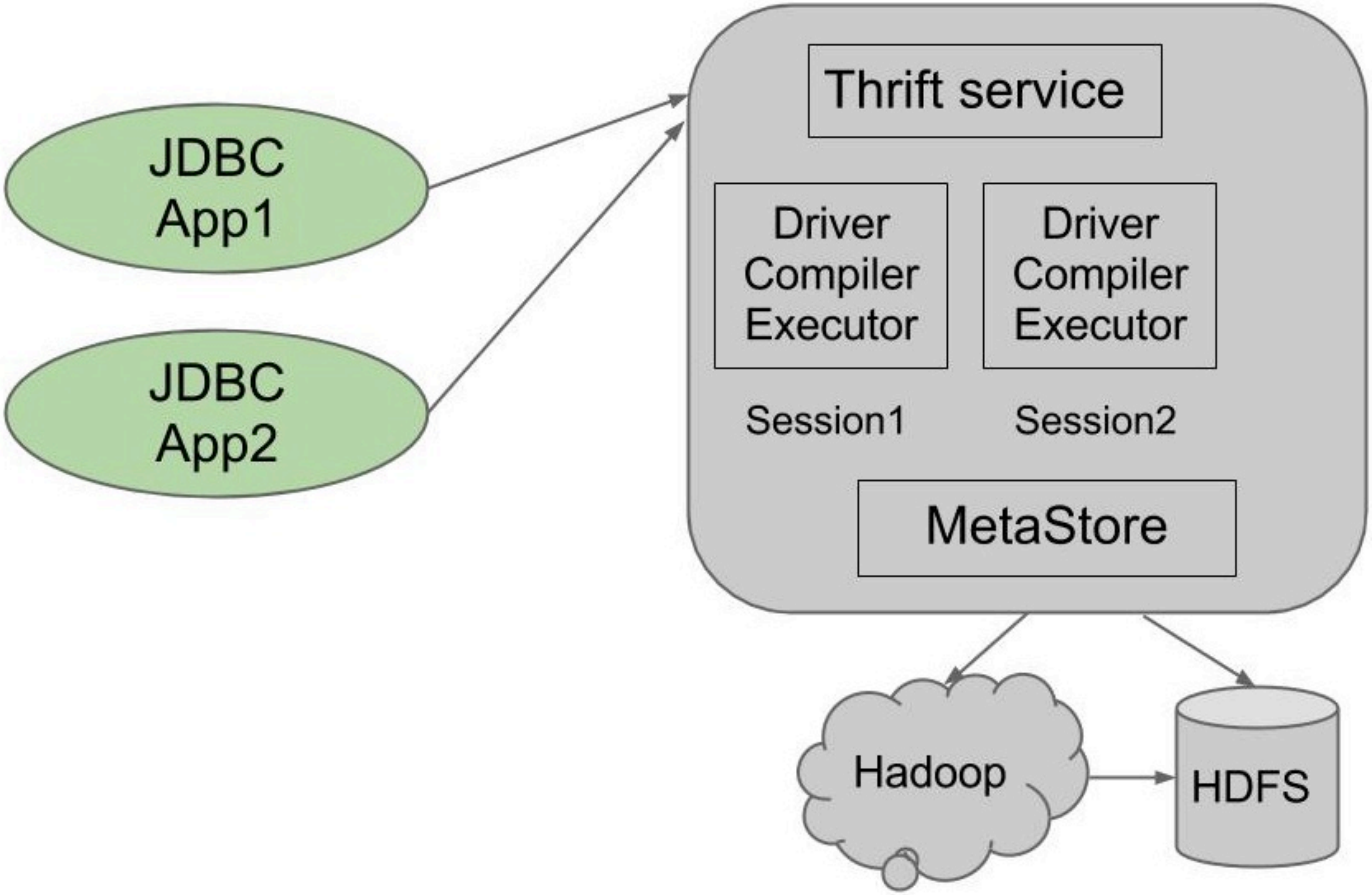
Hive server


Thrift service
JDBC Application
Hive Driver Compiler Executor
Meta Store
Hadoop
HDFS

Problems with Hive Server

- No sessions/concurrency
- Essentially need 1 server per client
- Security
- Auding/Logging

Hive server 2



Architecture Summary

- Use remote metastore service for sharing the metastore with HCatalog and other tools
- Use Hive Server2 for concurrent queries

Hive Server2 topology

Demo!

Why Impala?

- General-purpose SQL engine
- Real-time queries in Apache Hadoop

Impala Overview: Goals

- General-purpose SQL query engine:
  - Works for 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 – No MapReduce

User View of Impala: Overview

- Runs as a distributed service in cluster: one Impala daemon on each node with data
- Highly available: no single point of failure
- User submits query via ODBC/JDBC, Impala CLI or Hue to any of the daemons
- Query is distributed to all nodes with relevant data
- Impala uses Hive's metadata interface, connects to Hive metastore

User View of Impala: Overview

- There is no 'Impala format'!
- 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:
  - 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
  - SQL-style authorization via Apache Sentry (incubating)
  - UDFs and UDAFs are supported

User View of Impala: SQL

- Functional limitations:
  - No file formats, SerDes
  - no beyond SQL (buckets, samples, transforms, arrays, structs, maps, xpath, json)
  - Broadcast joins and partitioned hash joins supported
    - Smaller 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 rows
  - Predicates on other columns are mapped into SingleColumnValueFilters
- But: mapping of HBase tables metastore table patterned after Hive
  - All data stored as scalars and in ascii
  - The rowkey needs to be mapped into 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 Architecture

- Three binaries: impalad, statestored, catalogd
- 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
- Catalog daemon (catalogd) – 1 instance
  - Relays metadata changes to all impalad's

Impala Architecture

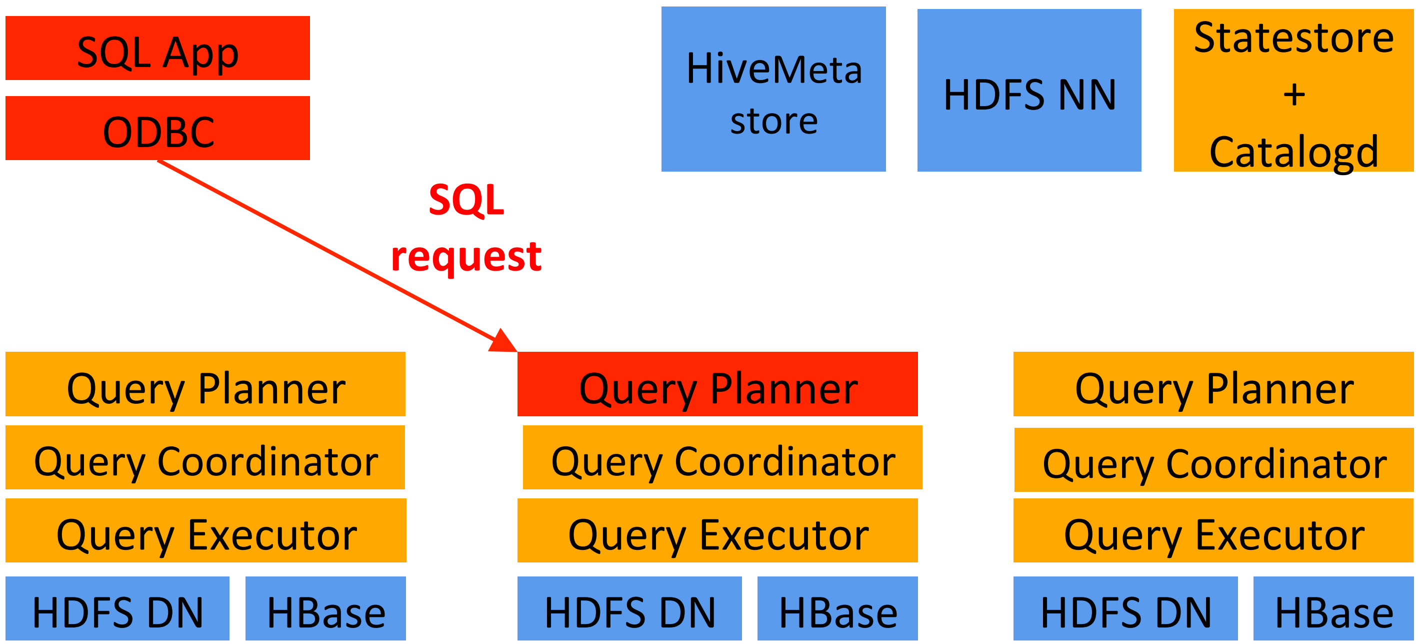
- Query execution phases
  - request arrives via odbc/jdbc
  - planner turns request into collections of plan fragments
  - coordinator initiates execution on remote impalad's

Impala Architecture

- 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

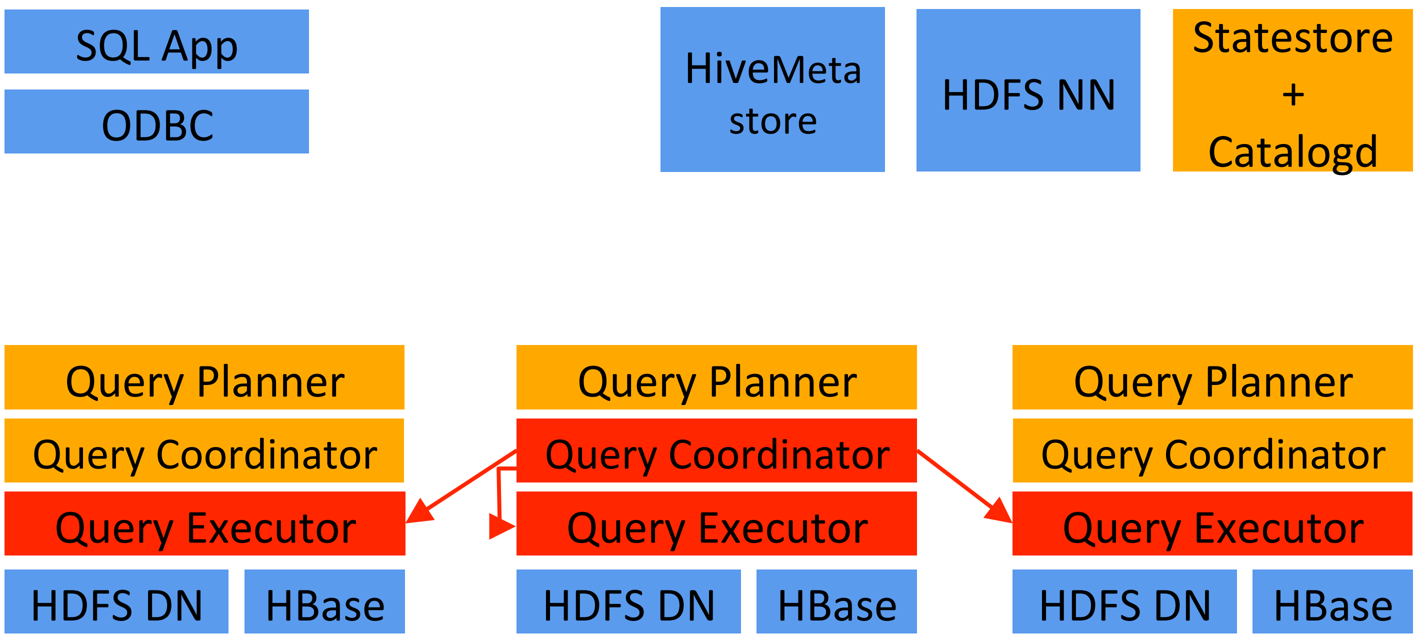
Request arrives via odbc/jdbc



Impala Architecture: Query Execution

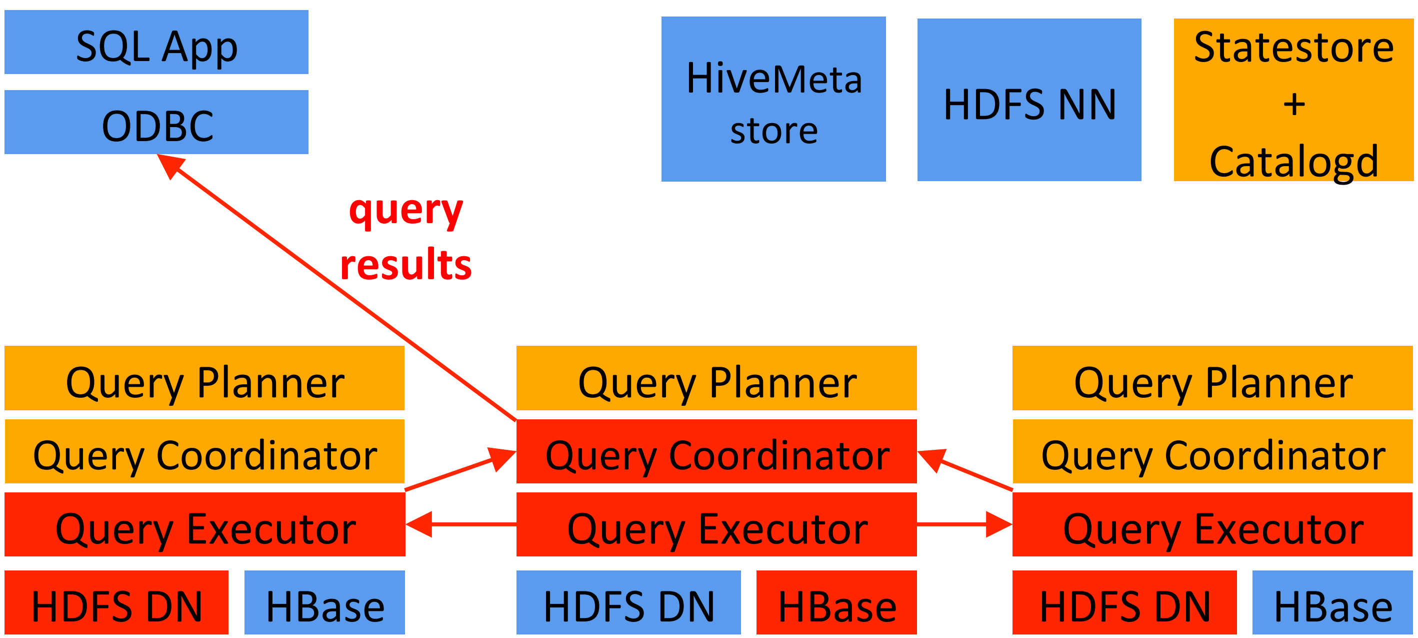
Planner turns request into collections of plan fragments

Coordinator initiates execution on remote impalad's



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
  - plan partitioning: partition single-node plan to maximize scan locality, minimize data movement
- Parallelization of operators:
  - All query operators are fully distributed

Query Planning: Single-Node Plan

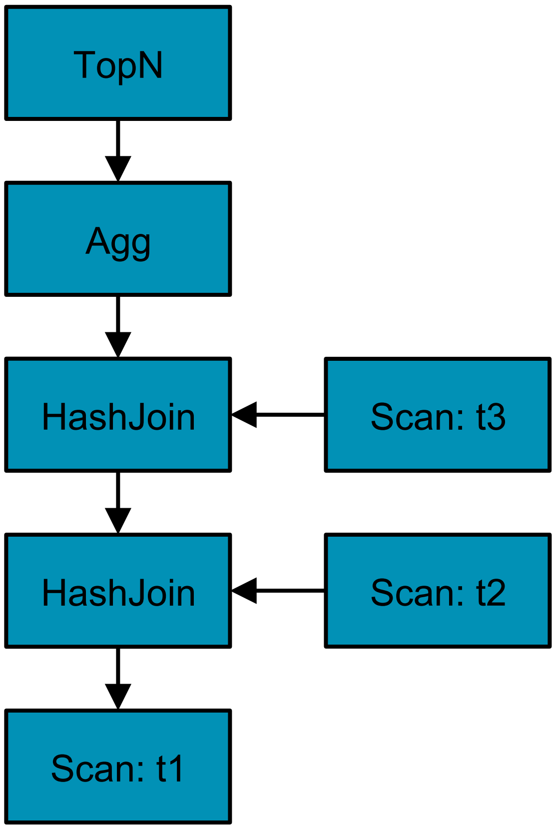
- Plan operators: Scan, HashJoin, HashAggregation, Union, TopN, Exchange

Single-Node Plan: 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: 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

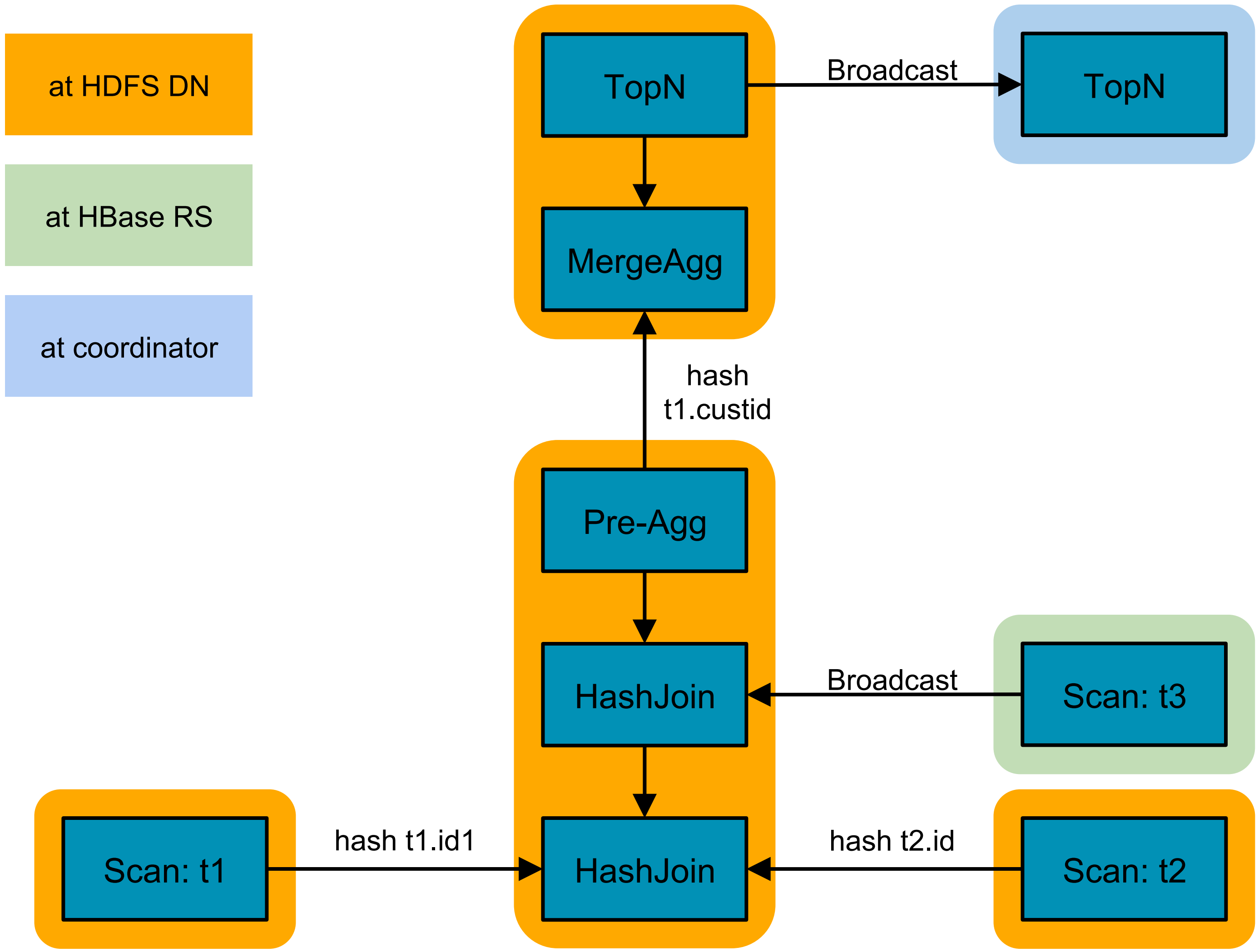
Query Planning: Distributed Plans

- 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

Metadata Handling

- Caches metadata: no synchronous metastore API calls during query execution
- impalad instances read metadata from metastore at startup
- Catalog Service relays metadata when you run DDL or update metadata on one of Impalad's
- `REFRESH [<tbl>]`: reloads metadata on all impalad's (if you added new files via Hive)
- `INVALIDATE METADATA`: reloads metadata for all tables
- Roadmap: HCatalog

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 a batch, tuple layout, column types, etc.
  - generate at compile time: unrolled 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)
  - Metadata
  - Roadmap: 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
  - 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)

More about Parquet

- What is it:
  - container format for all popular serialization formats: Avro, Thrift, Protocol Buffers
  - Successor to Trevni
  - jointly developed between Cloudera and Twitter
  - open source; hosted on github
- Features
  - rowgroup format: file contains multiple horiz. slices
  - supports storing each column in separate file
  - supports 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

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

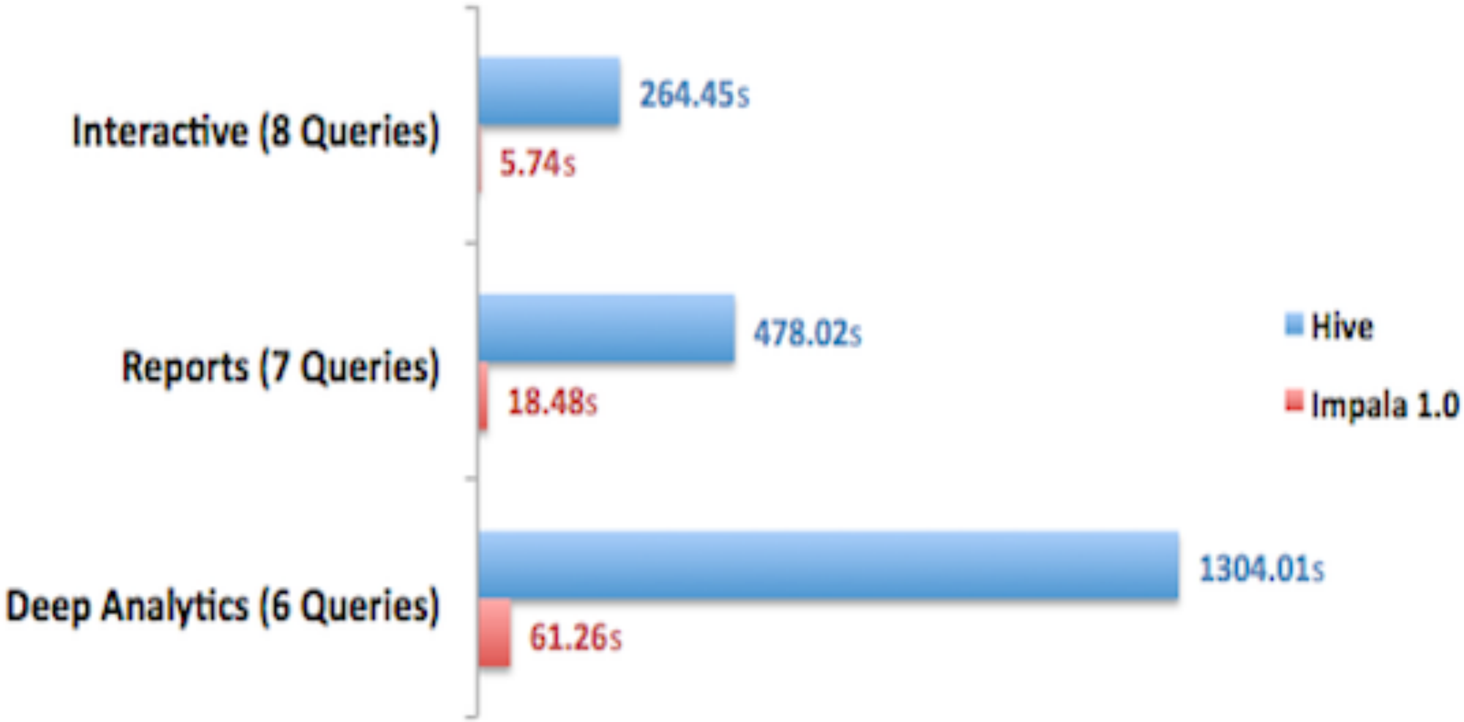
Comparing Impala to Hive

- 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 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

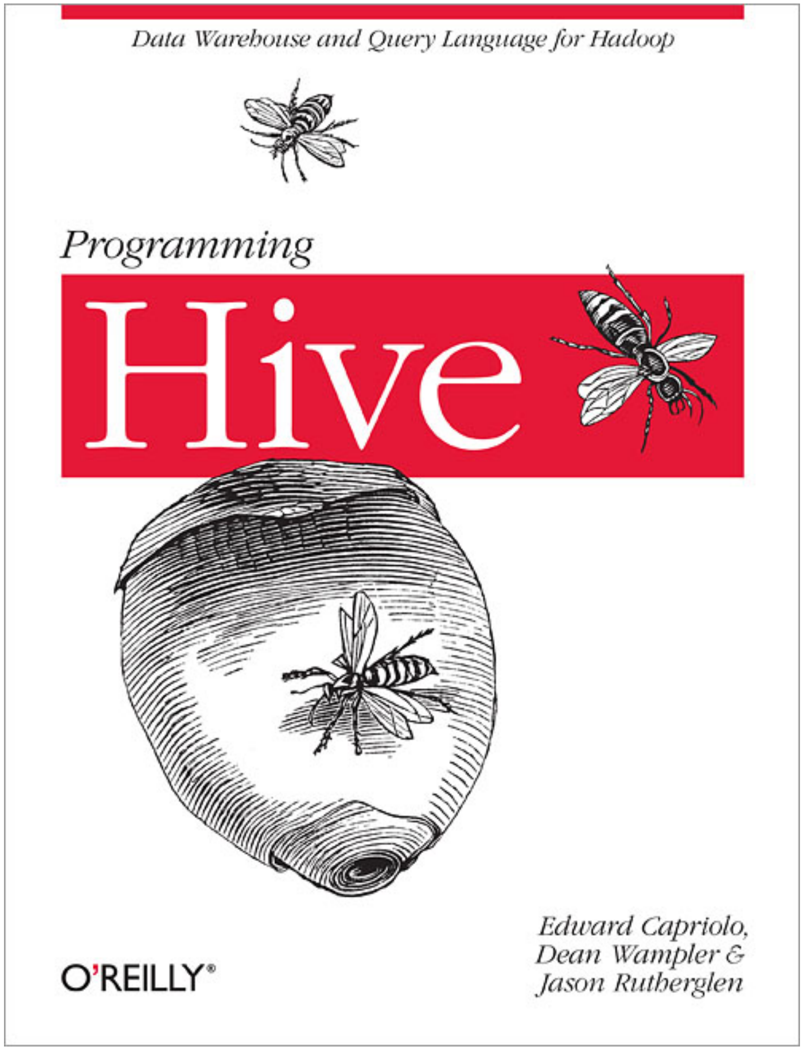
Impala Single-User Performance



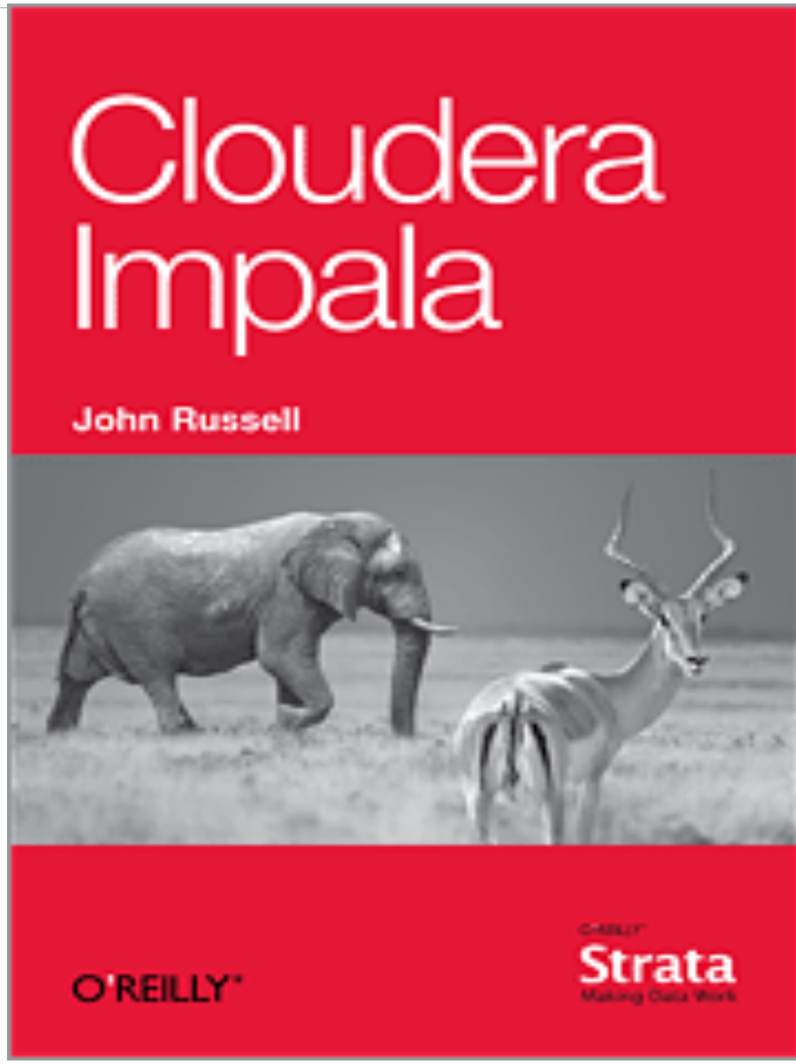
Impala Single-User Performance

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

Want to learn more about Hive?



Want to learn more about Impala?



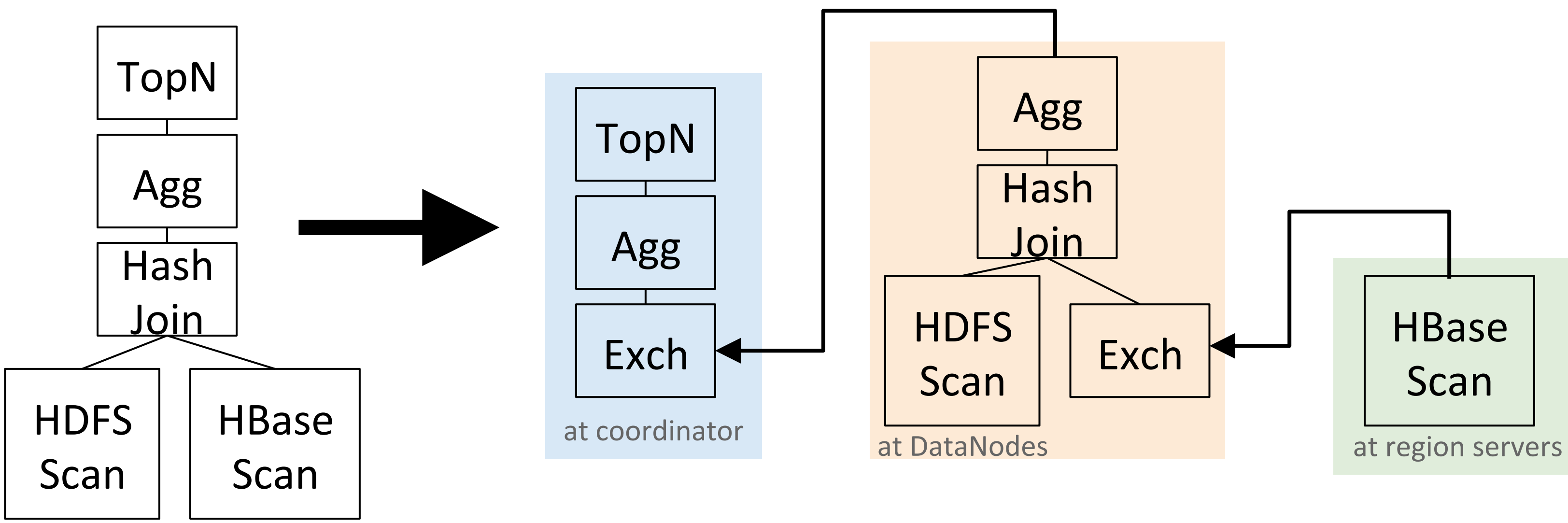
Impala Architecture: Planner

- Example: query with join and aggregation

  SELECT state, SUM(revenue)

  FROM HdfsTbl h JOIN HbaseTbl b ON (...)

  GROUP BY 1 ORDER BY 2 desc LIMIT 10



Demo!

ETL and real-time access

- Insert diagram here

Where to get Hive and Impala?

- Open source under ASL v2
- Open source distribution: CDH
- QuickStart VM: tiny.cloudera.com/quick-start

Contact info

Feedback: tiny.cloudera.com/mark

@mark_grover

github.com/markgrover

linkedin.com/in/grovermark

mgrover@cloudera.com

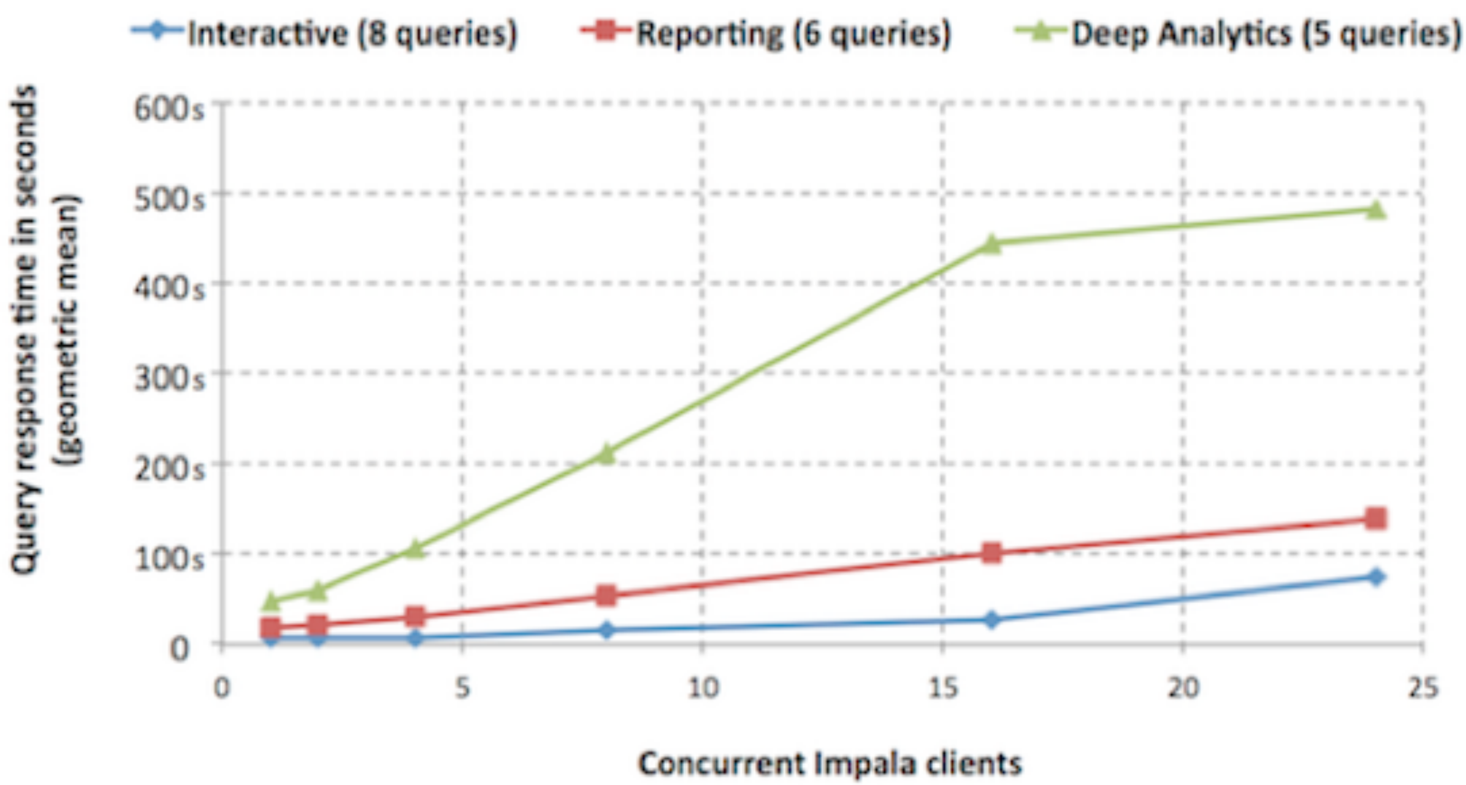
Impala Roadmap: 2014

- Additional SQL:
  - 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

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

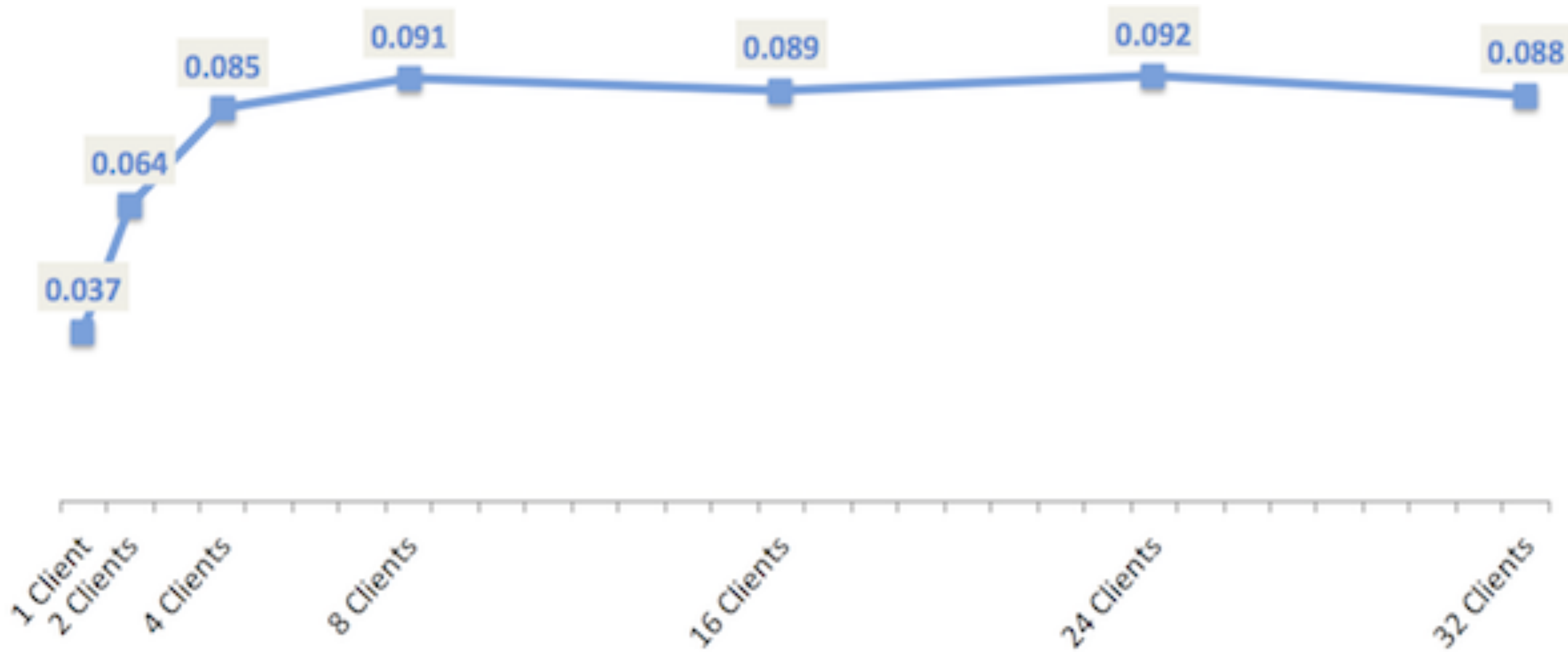
Impala Multi-User Performance



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 Multi-User Performance



Applications of Hive

- Web Analytics
- Retail
- Healthcare
- Spam detection
- Data Mining
- Ad optimization
- ETL workloads

Hive architecture

- Compiler
  - Parser
  - Type checking
  - Semantic Analyzer
  - Plan Generation
  - Task Generation

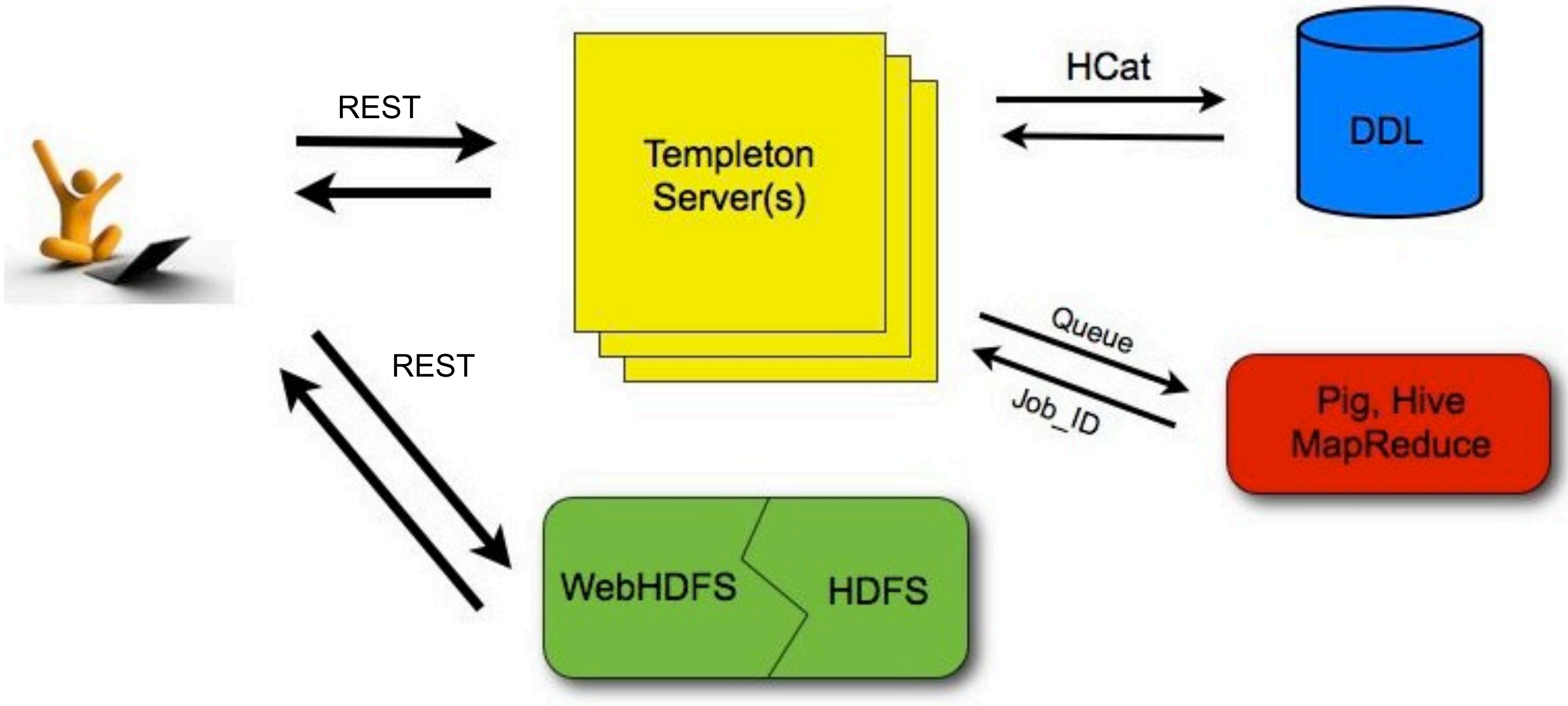
Hive architecture

- Execution Engine
  - Plan
  - Operators
  - SerDes
  - UDFs/UDAFs/UDTFs
- Metastore
  - Stores schema of data
  - HCatalog

HCatalog

- Sub-component of Hive
- Table and storage management service
- Public APIs and webservice wrappers for accessing metadata in Hive metastore
- Metastore contains information of interest to other tools (Pig, MapReduce jobs)
- Expose that information as REST interface
- WebHCat: Web Server for engaging with the Hive metastore

WebHCat



REST
Templeton Server(s)
HCat
DDL
REST
Queue
Job_ID
Pig, Hive MapReduce
WebHDFS
HDFS

Impala Roadmap: 2014

- Runtime optimizations:
  - straggler handling
  - improved cache management
  - data collocation for improved join performance
- Resource management:
  - goal: run exploratory and production workloads in same cluster, against same data, w/o impacting production jobs