TECHNICAL BRIEFING BOOK

cloudera®

Apache Impala (incubating): Analytic Database for Apache Hadoop

Highlights from the Cloudera Engineering Blog

Table of Contents

What is Impala?

Apache Impala (curently an ASF incubator project) provides high-performance, low-latency SQL queries on data stored in popular Apache Hadoop file formats. The fast response for queries enables interactive exploration and fine-tuning of analytic queries, rather than long batch jobs traditionally associated with SQL-on-Hadoop technologies. (You will often see the term "interactive" applied to these kinds of fast queries with human-scale response times.)

Impala integrates with the Apache Hive metastore database, to share databases and tables between both components. The high level of integration with Apache Hive, and compatibility with the HiveQL syntax, lets you use either Impala or Hive to create tables, issue queries, load data, and so on.

The following are some of the key advantages of Impala:

- Impala integrates with the existing CDH ecosystem, meaning data can be stored, shared, and accessed using the various solutions included with CDH, including components such as Apache Hive, Apache Spark, Apache Solr, and others. This integration also avoids data silos and minimizes expensive data movement, all with shared security, governance, and administration.
- Impala provides access to data stored in CDH without requiring the Java skills required for MapReduce jobs. Impala can access data directly from the HDFS filesystem. Impala also provides a SQL front-end to access data in Apache HBase or in the Amazon Simple Storage System (S3), and offers deep integrations with popular third-party BI tools.
- Impala returns results typically within seconds or a few minutes, rather than the many minutes or hours that are often required for Hive queries to complete.
- Impala is pioneering the use of the Apache Parquet file format, a columnar storage layout that is optimized for large-scale queries typical in data warehouse scenarios.

This Technical Briefing Book contains featured blog posts from the Cloudera Engineering Blog about key Impala concepts, Impala performance, and best practices. For more complete technical information, see:

- The *Impala Guide* in the Cloudera documentation
- Big Data Analyst courses from Cloudera University
- *Getting Started with Impala*, an O'Reilly Media book
- *The Impala Cookbook*

About Cloudera
Cloudera provides the world's fastest, easiest, and most secure data platform built on Apache Hadoop. We help solve your most demanding business challenges with data.

New SQL Benchmarks: Impala Uniquely Delivers Analytic Database Performance

By Devadutta Ghat, David Rorke, and Dileep Kumar (Feb. 2016)

New testing results show a significant difference between the analytic database performance of Impala compared to batch and procedural development engines, as well as Impala running all 99 TPC-DS-derived queries in the benchmark workload.

2015 was an exciting year for Apache Impala (incubating). Cloudera's Impala team significantly improved Impala's scale and stability, which enabled many customers to deploy Impala clusters with hundreds of nodes, run millions of queries, and even push Impala concurrency to thousands of users. We introduced highly anticipated features like nested data types (see below), unified fine-grained security with RecordService, and Apache Sentry (incubating). We unveiled Apache Kudu (incubating), which enables Impala to query fast-changing data and provide updatability, and Kudu and Impala both became Apache Incubator projects.

As noted in our recent roadmap update, 2016 is slated to be the most exciting year yet for Impala. With features like dynamic partition pruning, improved YARN integration, Amazon S3 support, and even better performance via multi-core joins and aggregations and increased runtime code-generation, Impala is set to take on even more use cases and greater concurrency as the analytic database for Apache Hadoop.

Over the past few years, the distinction has widened between systems designed as analytic databases, as compared to SQL interfaces exposed for easier development (such as Apache Hive and the Spark SQL module in Apache Spark). Cloudera's performance engineering team recently completed a new round of benchmarks comparing the most recent, stable releases of those SQL-on-Hadoop engines. For the low-latency and multi-user throughput performance requirements of BI and SQL analytics, there are clear, substantial differences between Impala as an analytic database compared to its batch and procedural-development engine peers:

- For multi-user queries, Impala is on average 16.4x faster than Hive-on-Tez and 7.6x faster than Spark SQL with Tungsten, with an average response time of 12.8s compared to over 1.6 minutes or more.
- All 99 TPC-DS-derived queries in the benchmark run on Impala, with similar results in Impala's favor.

These results demonstrate Impala's leadership in delivering the low-latency and multi-user throughput for running SQL analytics and BI on Hadoop. Even after the large investments that improved the performance in Hive with Stinger and Spark with Tungsten, there is nearly an order-of-magnitude difference in performance when compared to an analytic database like Impala.

We unfortunately cannot publish results against Impala's analytic database competitors due to their proprietary licensing restrictions, but we have published the benchmarking queries so you can replicate the results that customers such as Quaero, Epsilon, and many others have seen.

Now, for the details.

Configuration

All tests were run on the same 21-node cluster. For the previous round of testing, we ran on a smaller (64GB per node) memory footprint to correct a misperception that Impala only works well with lots of memory. But, this time, we ran on larger memory (384GB per node) to provide ample memory for Spark to perform at its best.

Each node was configured as follows:

- CPU: 2 sockets, 12 total cores, Intel Xeon CPU E5-2630L 0 at 2.00GHz
- 12 disk drives at 932GB each (one for the OS, the rest for HDFS)
- 384GB memory

Comparative Set

- Impala 2.3
- Hive-on-Tez: Hive 2.0 on Tez 0.5.2 (aka Stinger; see appendix for results on Tez 0.8.2)
- Spark SQL 1.5 with Tungsten

This time, we dropped Presto because it has not kept up with the SQL language functionality to run the latest benchmarks in an apples-to-apples comparison (due to its lack of `DECIMAL` support, for example).

Queries

In this post, we demonstrate how to run all 99 queries derived from TPC-DS on Impala. Given the significant advancements Impala and other engines have made since our last performance update, we dropped previously used rewrites with `DOUBLE` (as Impala and other engines support the `DECIMAL` datatype now) and also dropped SQL-92 style join rewrites (as other engines are now capable of running without them). Full queries are published here.

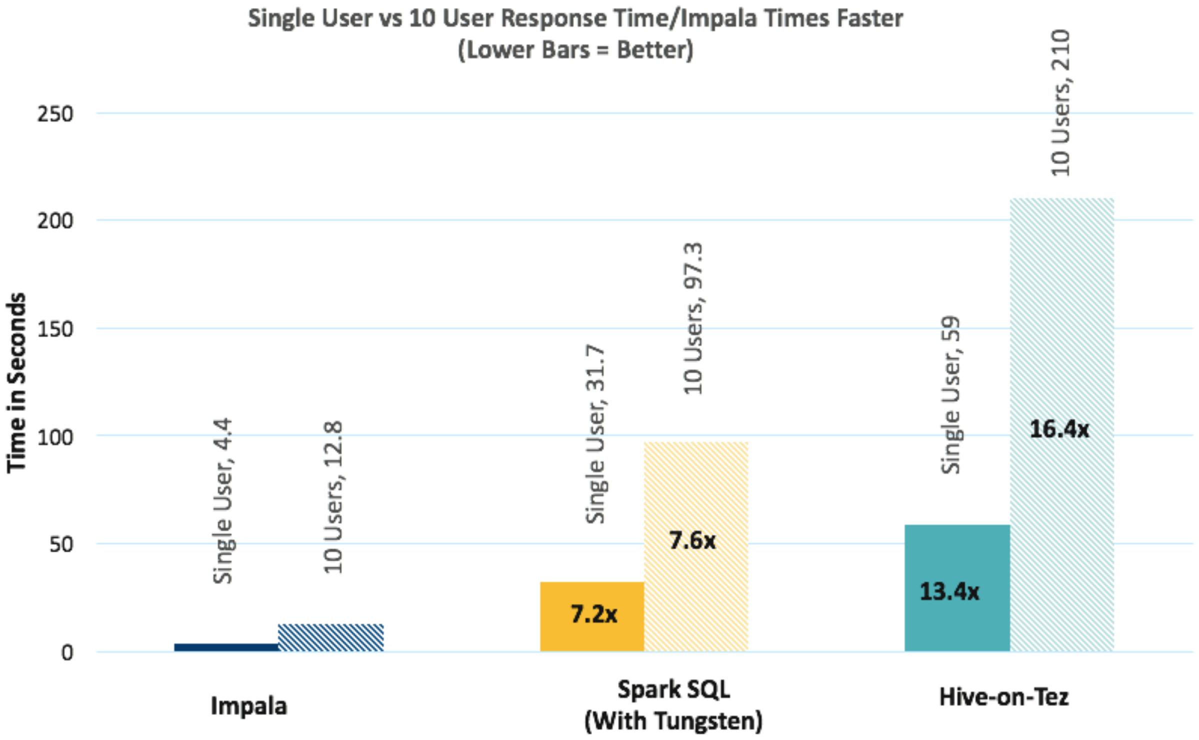
The testing used a 15TB scale-factor dataset. Each engine was assessed on a file format and compression that ensured the best possible performance and a fair, consistent comparison: Impala and Spark SQL on Apache Parquet with Snappy, and Hive-on-Tez on ORC with Zlib. The standard rigorous testing techniques (multiple runs, tuning, and so on) were used for each of the engines involved.

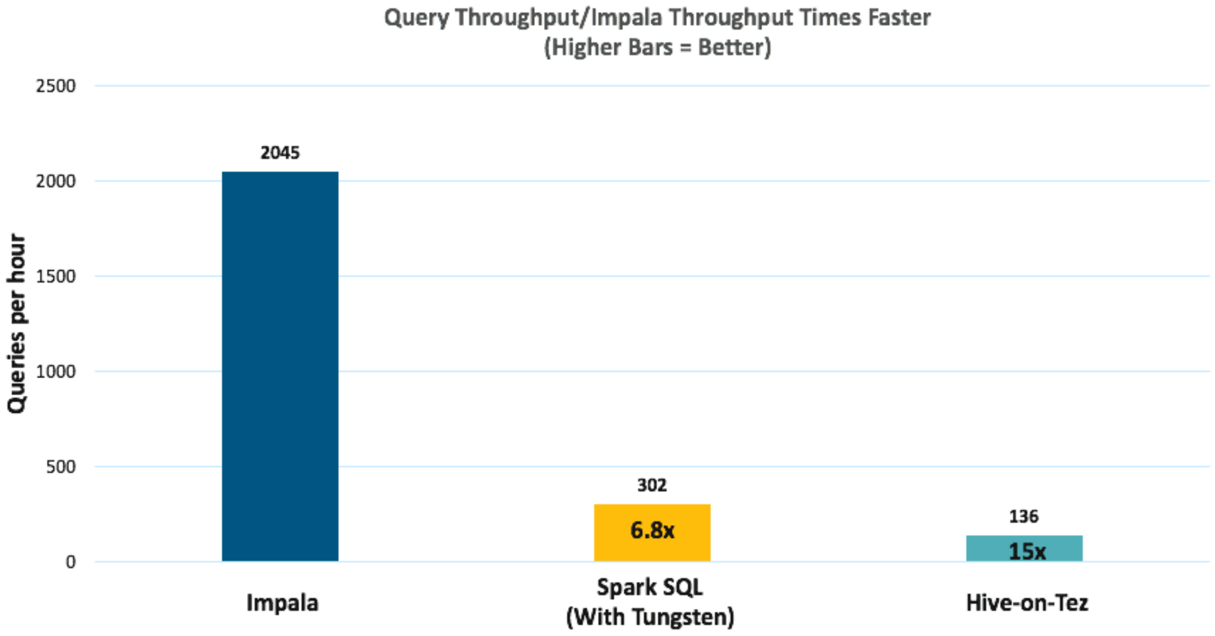
Results: Multiple Users

Our performance comparison focuses on multi-user results to simulate real-world workloads (as running a dedicated cluster per BI user is clearly impractical). We re-ran the same Interactive queries as in previous testing, running 10 users at the same time. To summarize the results:

- Impala is the only engine that provides interactive responses for the Interactive query set with an average of 12.8s, compared to the next nearest alternative with an average of over 1.5 minutes (7.6x slower).
- Impala delivers 6.8x – 15x higher throughput under concurrent load.

These results show that Impala is the only engine that consistently provides the interactive latency and concurrency needed for BI and SQL analytics, despite significant performance improvement efforts from Spark SQL with Tungsten as well as Hive-on-Tez with the Stinger initiative. We're excited to see these results as more significant investments in performance and concurrency are still under away.

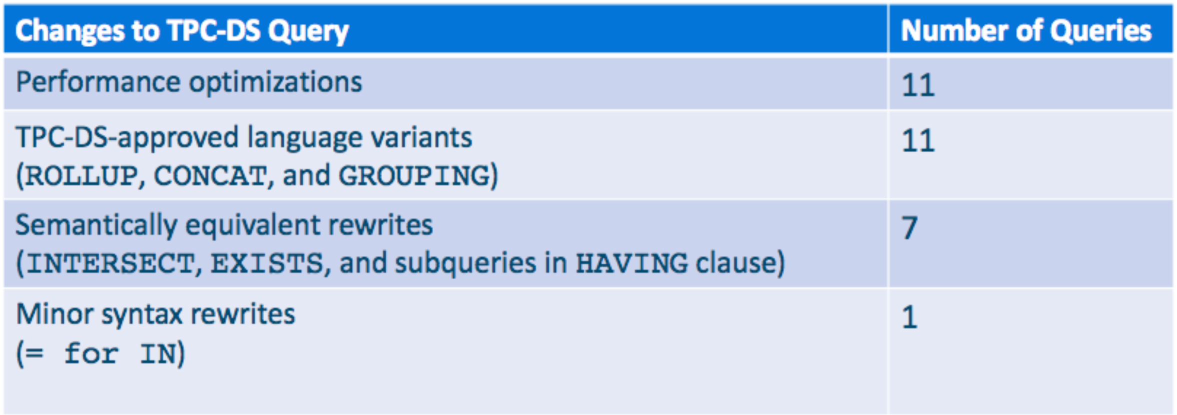




The astute reader will note that the performance numbers are slower than what was reflected in the previous testing. We attribute that result to the fact that, as noted previously, the originally specified `DECIMAL` types from the TPC-DS spec were included in the new round of testing, now that all the engines support `DECIMAL` (and `DECIMAL` incurs more computational complexity than `DOUBLE`.)

Running the Full 99 Queries Based on TPC-DS

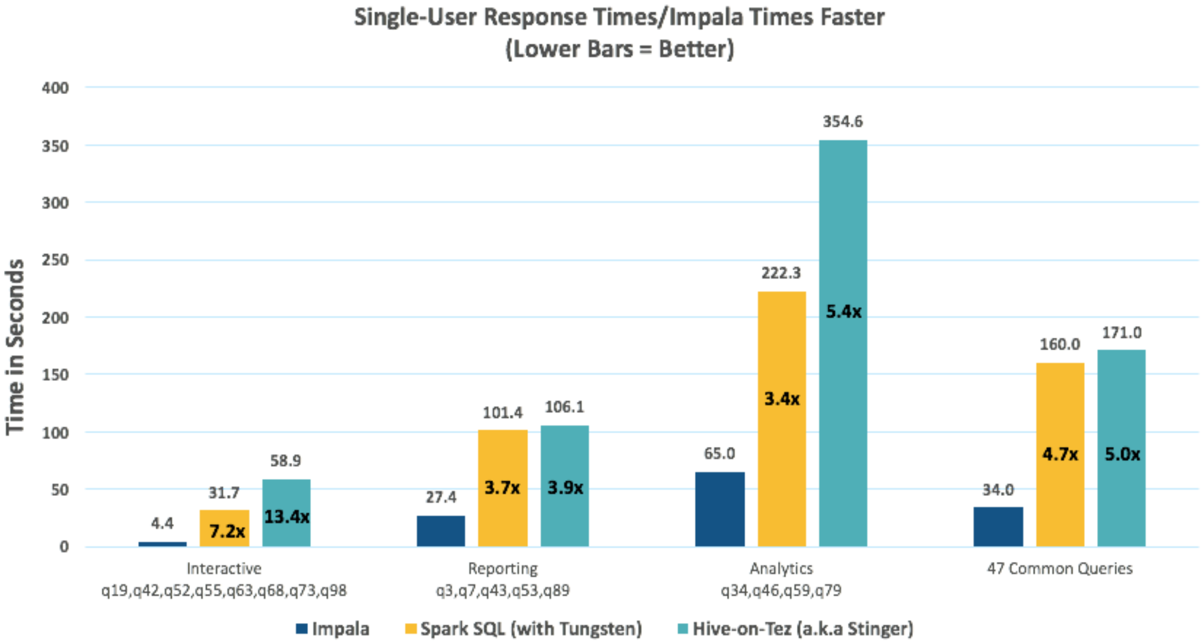
Next, we will demonstrate how to run all 99 queries derived from the TPC-DS spec on Impala. While most of the queries ran with no changes beyond the addition of simple partition filters, we modified some queries for language differences and additional optimizations, specifically:



Changes to TPC-DS Query	Number of Queries
Performance optimizations	11
TPC-DS-approved language variants (`ROLLUP`, `CONCAT`, and `GROUPING`)	11
Semantically equivalent rewrites (`INTERSECT`, `EXISTS`, and subqueries in `HAVING` clause)	7
Minor syntax rewrites (`= for IN`)	1

The 99 queries that run on Impala are published in the benchmark kit here. As you can see in the above table, most of the changes are due to a small number of language features used in multiple queries. For a fair apples-to-apples comparison across engines, we excluded these 30 queries and also eliminated queries that could not run on Spark SQL and Hive-on-Tez, which left a total of 47 remaining queries.

The chart below illustrates the breakdown and results of the same single-user buckets/queries as before, plus the 47 common queries. As noted previously, it's uncommon for real-world deployments to run a cluster with a single BI user, so the multiuser results from the previous section are a better comparison.



TPC-DS was designed to be representative of a traditional report-based workload, rather than the more common self-service and exploratory BI workloads you see in Hadoop today. The latter doesn't have a canonical industry benchmark. When assessing for your environment it's important to look at the characteristics of your workload that are best suited for particular systems. For Cloudera customers, the Cloudera Navigator Optimizer tool is designed to help with that assessment.

Conclusion

Cloudera's vision for Impala is for it to become the most performant, compatible, and usable analytic database. These results, which demonstrate how to run all 99 queries derived from TPC-DS on Impala, is another important milestone on the way to its status as the leader and open standard for BI and SQL analytics on modern big data architecture.

Choosing the right engine for the right job is very important. Despite Impala's significant performance lead as an analytic database, Hive and Spark SQL continue to provide important capabilities for other use cases and users alongside Impala:

- **Hive is designed to make batch processing jobs like data preparation and ETL more accessible than raw MapReduce via a SQL-like language.** Most data served to BI users in Impala today is prepared by ETL developers in Hive. Hive-on-Tez and Hive-on-Spark provide the same great Hive capabilities, yet use Tez or Spark as the execution engines for incrementally faster processing.
- **Spark SQL is an API within Spark that is designed for Scala or Java developers to embed SQL queries into their Spark programs.** This API enables common data engineering like aggregations, filters, joins, and so on to be simply expressed in SQL as part of a broader procedural Spark application. For example, data engineers and data scientists commonly use Spark for feature engineering and model development.
- **Impala is modern MPP query engine purpose-built for Hadoop to provide BI and SQL analytics at interactive latencies.** For BI users, there's a big difference between clicking on a report or visualization and getting a response in seconds, versus having to wait minutes. Interactivity is critical for BI users and that interactivity must be maintained as these tools scale to more users.

With multi-core joins and aggregates, dynamic partition pruning, and a variety of other performance enhancements coming up, we expect Impala's performance lead to widen further. Be on the lookout for new performance testing results that showcase these enhancements in future posts.

(As usual, we encourage you to independently verify these results by running your own benchmarks based on the open toolkit.)

This benchmark is derived from the TPC-DS benchmark and, as such, is incomparable to published TPC-DS results.

Devadutta Ghat is Senior Product Manager at Cloudera.

David Rorke is a Performance Engineer at Cloudera.

Dileep Kumar is a Performance Engineer at Cloudera.

Support for Complex Types in Impala

By Alex Behm (Nov. 2015)

The new support for complex types in Impala makes running analytic workloads considerably simpler.

Impala 2.3 (shipping starting in Cloudera Enterprise 5.5) contains support for querying complex types in Apache Parquet tables, specifically `ARRAY`, `MAP`, and `STRUCT`s. This capability enables users to query against naturally nested data sets without having to perform ETL to flatten them. This feature provides a few major benefits, including:

It removes additional ETL and data modeling work to flatten data sets.

It makes queries easier by maintaining the natural relationship between nested data elements.

It boosts performance by removing joins.

This post is a gentle introduction to querying Impala tables with complex types; we will have follow-up posts that will go into more depth. Our goal is give you a quick understanding of Impala's design philosophy and querying capabilities in Impala 2.3. We will focus on the SQL syntax extensions by presenting a series of simple SQL examples.

The Impala team is excited about the first Impala release with complex types support, and we hope to get you excited, too!

Design Goals

The new SQL language extensions were designed with a few principles in mind:

- The syntax should feel natural to the user, and should be a natural extension of SQL.
- It should also allow the full expressiveness of SQL with complex types.
- Common querying patterns can be expressed concisely.
- Queries can be executed efficiently.

Consequently, we came up with the following main ideas and extensions:

- Nested fields within a `STRUCT` are referenced via the familiar `.` notation.
- The collection types `ARRAY` and `MAP` are referenced in the `FROM` clause, just like conventional tables. This exposes the nested data as columns that can be referenced as usual.

- Subqueries (including inline views) can reference nested `ARRAY`s and `MAP`s from tables in enclosing `SELECT` blocks. This allows a SQL-in-SQL pattern where the data in nested collections can be filtered/joined/aggregated with the full expressiveness of SQL.

Onward to the examples!

Example Schema

The following schema models a hypothetical customer data warehouse that contains data that might have been assembled from various data sources. There is routine customer data like name, address, orders, and so on, but also data about website and call-center interactions, all in a single table. The schema presents a customer-centric view of the data with the intent of performing customer-centric analyses.

We have highlighted the complex types below; the column/field names should be more-or-less self-explanatory. Their meaning will become clear in the examples to follow.

```
CREATE TABLE customers (
  cid BIGINT,
  name STRING,
  mktsegment STRING,
  address STRUCT<
    city: STRING,
    street: STRING,
    street_number: INT,
    zip: SMALLINT
  >,
  phone_numbers ARRAY<STRING>,
  orders ARRAY<STRUCT<
    oid: BIGINT,
    status: STRING,
    totalprice: DECIMAL(12,2),
    order_date: STRING,
    items: ARRAY<STRUCT<
      iid: BIGINT,
      name: STRING,
      price: DECIMAL(12,2),
      discount_perc: DECIMAL(2,2),
      shipdate: STRING
    >>
  >>,
  /* Keyed on web URL */
  web_visits MAP<STRING,STRUCT<
    user_agent: STRING,
    client_ip: STRING,
    visit_date: STRING,
    duration_ms: INT
  >>,
  support_calls ARRAY<STRUCT<
    agent_id: BIGINT,
    call_date: STRING,
    duration_ms: BIGINT,
    issue_resolved: BOOLEAN,
    agent_comment: STRING
  >>
)
STORED AS PARQUET;
```

Query A: Referencing STRUCT fields

Find all customer names and their zip in the automotive market segment in San Francisco.

```
SELECT name, address.zip FROM customers
WHERE mktsegment = "AUTOMOTIVE" AND address.city = "SAN FRANCISCO"
```

The nested fields of the `STRUCT`-typed address column are accessed via `.` notation.

Query B: Accessing a Nested Collection

List all orders from 11/27/2015 with a total price over $1000.

```
SELECT oid, status, totalprice, order_date FROM customers.orders
WHERE order_date = "11-27-2015" and totalprice > 1000
```

The `.` notation in the `FROM` clause is used to expose the nested `orders` collection as a table that contains all orders of all customers. By referencing the nested `orders` collection in the `FROM` clause, we can use its fields anywhere a conventional column reference could appear. Notice that in this query we do not reference any of the top-level customer fields.

Query C: Accessing a Deeply Nested Collection in a Single Path

Compute the average item price over all items.

```
SELECT AVG(price) FROM customers.orders.items
```

Here the `FROM` clause produces a table with all items of all orders of all customers. A single dotted path in the `FROM` clause can traverse any number of collections, flattening all of them.

Query D: Using Relative Table References

Compute the minimum and maximum duration of all successfully resolved support calls by market segment.

```
SELECT MIN(duration_ms), MAX(duration_ms) FROM customers c, c.support_calls s
WHERE s.issue_resolved = true
GROUP BY c.mktsegment
```

Unlike the previous examples, this query references top-level customer columns, as well as nested fields from the `support_calls` collection. For every table or nested collection we wish to use fields/columns from, we need to establish a table alias in the `FROM` clause. In this example, we have an alias `c` that exposes the top-level columns `cid`, `name`, etc. and another alias `s` that exposes the columns `agent_id`, `order_date`, `duration_ms`, and so on. You can think of the `FROM` clause as a join between all customers and all support calls on the implicit is-nested-in relationship. This implicit join condition is expressed by virtue of the `c.support_calls` reference that is relative to the alias `c`.

Query E: ANSI-92 Joins with Nested Collections

List all customers and their orders from a specific zip code, including customers that have no orders.

```
SELECT cid, name, oid, status, totalprice, order_date
FROM customers c LEFT OUTER JOIN c.orders
WHERE address.zip = 92309
```

The `LEFT OUTER JOIN` flattens the nested orders collection while preserving those customers that have no orders. The orders columns are set to `NULL` for customers that have no orders. Notice that an `ON`-clause is not required here due to the implicit is-nested-in join condition.

Query F: Pseudocolumns of Arrays

List all phone numbers of a specific customer.

```
SELECT c.cid, c.name, p.item, p.pos FROM customers c, c.phone_numbers p
WHERE c.cid = 12345
```

We use the pseudocolumn `item` to refer to the column exposed by `c.phone_numbers` because the element type of the `phone_numbers` array is an anonymous scalar. The `pos` pseudocolumn contains the ordinal position of `item` in the corresponding array.

Query G: Pseudocolumns of Maps

Count the number of distinct user agents that accessed a purchasing-related URL.

```
SELECT COUNT(DISTINCT w.user_agent) FROM customers.web_visits w
WHERE w.key LIKE "%purchase%"
```

We use the pseudocolumn `key` to refer to the key portion of the `web_visits` map. Similarly, the `value` pseudocolumn can be used to access the value portion of the map (e.g., if the map's value was an anonymous scalar type).

Query H: Correlated Table References in Inline Views

Count the number of pending orders and show their total value for every customer living in "Palo Alto".

```
SELECT cid, name, order_cnt, price_sum
  FROM customers c,
    (SELECT COUNT(*) order_cnt, SUM(totalprice) price_sum FROM c.orders
     WHERE status = "PENDING") v
  WHERE address.city = "PALO ALTO"
```

This example shows how a relative reference to a nested collection `c.orders` can be made inside an inline view. We say the reference is correlated because it references a table alias from an enclosing query block, akin to conventional correlated subqueries in SQL. One way to think about the meaning of this query is that the query inside `v` is evaluated for every customer `c` due to the correlated reference. The result of each `v` evaluation is then joined with the corresponding `c` row. Such inline views can contain arbitrary SQL for operating on nested collections. Note that this query does per-customer aggregation, but does not require a `GROUP BY` clause.

Query I: Correlated Table References in Subqueries

Count the number of customers who have no orders but at least one web visit, grouped by market segment.

```
SELECT mktsegment, COUNT(*)
  FROM customers c
  WHERE NOT EXISTS (SELECT oid FROM c.orders)
    AND EXISTS (SELECT key from c.web_visits)
  GROUP BY mktsegment
```

This example shows two `EXISTS` subqueries that have correlated table references to filter a customer based on evaluating some SQL over the nested `orders` and `web_visits` collections of that particular customer. Note that in a flat schema, this query would require explicit and potentially expensive distributed joins between `customers`, `orders`, and `web_visits`.

Query J: Deeply Nested Subqueries

List all customers that have at least five orders that were ordered on a date where the customer also had at least one support call and web interaction.

```
SELECT cid, name, cnt
  FROM customers c,
    (SELECT COUNT(oid) cnt FROM c.orders o
     WHERE order_date IN (SELECT visit_date FROM c.web_visits)
       AND order_date IN (SELECT call_date FROM c.support_calls)
     HAVING cnt >= 5
    ) v
```

The purpose of this final example is to show a more complicated query with multiple nested subqueries that contain correlated table references. Subqueries can be arbitrarily nested just as in the conventional "flat" SQL.

Useful Utility Commands

When working with data that has complex types, you might find the following utility commands useful.

- `CREATE TABLE <tbl> LIKE PARQUET 'path_to_file'`. See the documentation.
- `DESCRIBE <path>`. See `ARRAY` type and `MAP` type.

What's Next

While the core built-in language extensions are available in CDH 5.5 today, we are continuing to improve the complex types feature on several dimensions, and have the following items on the roadmap for future releases:

- Support for Apache Avro
- Support for JSON files
- Syntactic sugar for even more concise aggregates over collections
- `INSERT` into tables with complex types
- Builtin functions and user-defined functions that return and/or operate on complex types
- Performance improvements

This post was intended to be a short introduction and reference, so we inevitably omitted many interesting details. For more information and examples, try the following resources:

- Impala Documentation (Complex Types)
- Presentation: “Nested Types in Impala“
- Presentation: “Data Modeling for Data Science: Simplify Your Workload with Complex Types in Impala“

Alex Behm is a Software Engineer at Cloudera, working on the Impala team.

How-to: Use Impala with Kudu

By Misty Stanley-Jones (Nov. 2015)

Learn the details about using Impala alongside Kudu.

Kudu (currently in beta), the new storage layer for the Apache Hadoop ecosystem, is tightly integrated with Impala, allowing you to insert, query, update, and delete data from Kudu tablets using Impala's SQL syntax, as an alternative to using the Kudu APIs to build a custom Kudu application. In addition, you can use JDBC or ODBC to connect existing or new applications written in any language, framework, or business intelligence tool to your Kudu data, using Impala as the broker. This integration relies on features that released versions of Impala do not have yet, as of Impala 2.3, which is expected to ship in CDH 5.5. In the interim, you need to install a fork of Impala called Impala_Kudu.

In this post, you will learn about the various ways to create and partition tables as well as currently supported SQL operators. This post assumes a successful install of the Impala_Kudu package via Cloudera Manager or command line; see the docs for instructions. Note these prerequisites:

- Impala_Kudu depends upon CDH 5.4 or later. To use Cloudera Manager with Impala_Kudu, you need Cloudera Manager 5.4.3 or later. Cloudera Manager 5.4.7 is recommended, as it adds support for collecting metrics from Kudu.
- If you have an existing Impala instance on your cluster, you can install Impala_Kudu alongside the existing Impala instance *if you use parcels*. The new instance does not share configurations with the existing instance and is completely independent. A script is provided to automate this type of installation.
- It is especially important that the cluster has adequate unreserved RAM for the Impala_Kudu instance.
- Consider shutting down the original Impala service when testing Impala_Kudu if you want to be sure it is not impacted.
- Before installing Impala_Kudu, you must have already installed and configured services for HDFS, Apache Hive, and Kudu. You may need Apache HBase, YARN, Apache Sentry, and Apache ZooKeeper services as well.

Using Impala with Kudu

Neither Kudu nor Impala need special configuration for you to use the Impala Shell or the Impala API to insert, update, delete, or query Kudu data using Impala. However, you do need to create a mapping between the Impala and Kudu tables. Kudu provides the Impala query to map to an existing Kudu table in the web UI.

- Be sure you are using the impala-shell binary provided by the Impala_Kudu package, rather than the default CDH Impala binary. The following shows how to verify this using the alternatives command on a RHEL 6 host. Do not copy and paste the alternatives `--set` command directly, because the file names are likely to differ.

```
$ sudo alternatives --display impala-shell

impala-shell - status is auto. link currently points to
/opt/cloudera/parcels/CDH-5.4.6-1.cdh5.4.6.p0.1007/bin/impala-shell
/opt/cloudera/parcels/CDH-5.4.6-1.cdh5.4.6.p0.1007/bin/impala-shell -
priority 10
/opt/cloudera/parcels/IMPALA_KUDU-2.3.0-1.cdh5.4.6.p0.119/bin/impala-
shell - priority 5
Current `best' version is /opt/cloudera/parcels/CDH-5.4.0-
1.cdh5.6.0.p0.1007/bin/impala-shell.

$ sudo alternatives --set impala-shell
/opt/cloudera/parcels/IMPALA_KUDU-2.3.0-1.cdh5.4.6.p0.119/bin/impala-
shell
```

- Start Impala Shell using the `impala-shell` command. By default, `impala-shell` attempts to connect to the Impala daemon on localhost on port 21000. To connect to a different host, use the `-i <host:port>` option. To automatically connect to a specific Impala database, use the `-d <database>` option. For instance, if all your Kudu tables are in Impala in the database impala_kudu, use `-d impala_kudu` to use this database.

- To quit the Impala Shell, use the following command: `quit;`

Internal and External Impala Tables

When creating a new Kudu table using Impala, you can create the table as an internal table or an external table.

- Internal: An internal table (created by `CREATE TABLE`) is managed by Impala, and can be dropped by Impala. When you create a new table using Impala, it is generally a internal table.

- External: An external table (created by `CREATE EXTERNAL TABLE`) is not managed by Impala, and dropping such a table does not drop the table from its source location (here, Kudu).

Instead, it only removes the mapping between Impala and Kudu. This is the mode used in the syntax provided by Kudu for mapping an existing table to Impala.

See the Impala documentation for more information about internal and external tables.

Querying an Existing Kudu Table In Impala

- Go to http://kudu-master.example.com:8051/tables/, where kudu-master.example.com is the address of your Kudu master.
- Click the table ID link for the relevant table.
- Scroll to the bottom of the page, or search for the text Impala `CREATE TABLE` statement. Copy the entire statement.
- Paste the statement into Impala Shell. Impala now has a mapping to your Kudu table.

Creating a New Kudu Table From Impala

Creating a new table in Kudu from Impala is similar to mapping an existing Kudu table to an Impala table, except that you need to write the `CREATE` statement yourself. Use the following example as a guideline. Impala first creates the table, then creates the mapping.

This example does not use a partitioning schema. However, you will almost always want to define a schema to pre-split your table.

```
CREATE TABLE `my_first_table` (
`id` BIGINT,
`name` STRING
)
TBLPROPERTIES(
  'storage_handler' = 'com.cloudera.kudu.hive.KuduStorageHandler',
  'kudu.table_name' = 'my_first_table',
  'kudu.master_addresses' = 'kudu-master.example.com:7051',
  'kudu.key_columns' = 'id'
);
```

In the `CREATE TABLE` statement, the columns that comprise the primary key must be listed first. Additionally, primary key columns are implicitly marked `NOT NULL`.

The following table properties are required, and the `kudu.key_columns` property must contain at least one column.

- `storage_handler`: the mechanism used by Impala to determine the type of data source. For Kudu tables, this must be com.cloudera.kudu.hive.KuduStorageHandler.

- `kudu.table_name`: the name of the table that Impala will create (or map to) in Kudu
- `kudu.master_addresses`: the list of Kudu masters with which Impala should communicate
- `kudu.key_columns`: the comma-separated list of primary key columns, whose contents should not be nullable

CREATE TABLE AS SELECT. You can create a table by querying any other table or tables in Impala, using a `CREATE TABLE AS SELECT` query.

The following example imports all rows from an existing table old_table into a Kudu table new_table. The columns in new_table will have the same names and types as the columns in old_table, but you need to populate the `kudu.key_columns` property. In this example, the primary key columns are ts and name.

```
CREATE TABLE new_table AS
SELECT * FROM old_table
TBLPROPERTIES(
  'storage_handler' = 'com.cloudera.kudu.hive.KuduStorageHandler',
  'kudu.table_name' = 'new_table',
  'kudu.master_addresses' = 'kudu-master.example.com:7051',
  'kudu.key_columns' = 'ts, name'
);
```

You can refine the `SELECT` statement to only match the rows and columns you want to be inserted into the new table. You can also rename the columns by using syntax like `SELECT` name as new_name.

Partitioning Tables

Tables are partitioned into tablets according to a partition schema on the primary key columns. Each tablet is served by at least one tablet server. Ideally, a table should be split into tablets that are distributed across a number of tablet servers to maximize parallel operations. The details of the partitioning schema you use will depend entirely on the type of data you store and how you access it.

Kudu currently has no mechanism for splitting or merging tablets after the table has been created. Until this feature has been implemented, you must provide a partition schema for your table when you create it. When designing your tables, consider using primary keys that will allow you to partition your table into tablets which grow at similar rates.

You can partition your table using Impala's `DISTRIBUTE BY` keyword, which supports distribution by `RANGE` or `HASH`. The partition scheme can contain zero or more `HASH` definitions, followed by an optional `RANGE` definition. The `RANGE` definition can refer to one or more primary key columns. Examples of basic and advanced partitioning are shown below. **Note**: Impala keywords, such as group, are enclosed by back-tick characters when they are used as identifiers, rather than as keywords.

Basic Partitioning

`DISTRIBUTE BY RANGE`. You can specify split rows for one or more primary key columns that contain integer or string values. Range partitioning in Kudu allows splitting a table based on the lexicographic order of its primary keys. This allows you to balance parallelism in writes with scan efficiency.

The split row does not need to exist. It defines an exclusive bound in the form of:

```
(START_KEY, SplitRow), [SplitRow, STOP_KEY)
```

|

In other words, the split row, if it exists, is included in the tablet after the split point. For instance, if you specify a split row abc, a row abca would be in the second tablet, while a row abb would be in the first.

Suppose you have a table that has columns state, name, and purchase_count. The following example creates 50 tablets, one per US state. **Note:** If you partition by range on a column whose values are monotonically increasing, the last tablet will grow much larger than the others. Additionally, all data being inserted will be written to a single tablet at a time, limiting the scalability of data ingest. In that case, consider distributing by HASH instead of, or in addition to, RANGE.

```
CREATE TABLE customers (
  state STRING,
  name STRING,
  purchase_count int32,
) DISTRIBUTE BY RANGE(state)
SPLIT ROWS(('al'), ('ak'), ('ar'), .., ('wv'), ('wy'))
TBLPROPERTIES(
'storage_handler' = 'com.cloudera.kudu.hive.KuduStorageHandler',
'kudu.table_name' = 'customers',
'kudu.master_addresses' = 'kudu-master.example.com:7051',
'kudu.key_columns' = 'state, name'
);
```

`DISTRIBUTE BY HASH`. Instead of distributing by an explicit range, or in combination with range distribution, you can distribute into a specific number of "buckets" by hash. You specify the primary key columns you want to partition by, and the number of buckets you want to use. Rows are distributed by hashing the specified key columns. Assuming that the values being hashed do not themselves exhibit significant skew, this will serve to distribute the data evenly across buckets.

You can specify multiple definitions, and you can specify definitions which use compound primary keys. However, one column cannot be mentioned in multiple hash definitions. Consider two columns, a and b:

- HASH(a), HASH(b) — will succeed
- HASH(a,b) — will succeed
- HASH(a), HASH(a,b) — will fail

Note: `DISTRIBUTE BY HASH` with no column specified is a shortcut to create the desired number of buckets by hashing all primary key columns.

Hash partitioning is a reasonable approach if primary key values are evenly distributed in their domain and no data skew is apparent, such as timestamps or serial IDs.

The following example creates 16 tablets by hashing the id column. A maximum of 16 tablets can be written to in parallel. In this example, a query for a range of sku values is likely to need to read from all 16 tablets, so this may not be the optimum schema for this table. See Advanced Partitioning for an extended example.

```
CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT
)
DISTRIBUTE BY HASH (id) INTO 16 BUCKETS
TBLPROPERTIES(
'storage_handler' = 'com.cloudera.kudu.hive.KuduStorageHandler',
'kudu.table_name' = 'cust_behavior',
'kudu.master_addresses' = 'kudu-master.example.com:7051',
'kudu.key_columns' = 'id, sku'
);
```

Advanced Partitioning

You can use zero or more `HASH` definitions, followed by zero or one `RANGE` definitions to partition a table. Each definition can encompass one or more columns. While every possible distribution schema is out of the scope of this document, a few demonstrations follow.

`DISTRIBUTE BY RANGE` Using Compound Split Rows. This example creates 100 tablets, two for each US state. Per state, the first tablet holds names starting with characters before m, and the second tablet holds names starting with m-z. At least 50 tablets (and up to 100) can be written to in parallel. A query for a range of names in a given state is likely to only need to read from one tablet, while a query for a range of names across every state will likely only read from 50 tablets.

```
CREATE TABLE customers (
  state STRING,
  name STRING,
  purchase_count int32,
) DISTRIBUTE BY RANGE(state, name)
SPLIT ROWS(('al', ''), ('al', 'm'), ('ak', ''), ('ak', 'm'),
  ..,
  ('wy', ''), ('wy', 'm'))
TBLPROPERTIES(
'storage_handler' = 'com.cloudera.kudu.hive.KuduStorageHandler',
'kudu.table_name' = 'customers',
'kudu.master_addresses' = 'kudu-master.example.com:7051',
'kudu.key_columns' = 'state, name'
);
```

`DISTRIBUTE BY HASH and RANGE`. Let's go back to the hashing example above. If you often query for a range of sku values, you can optimize the example by combining hash partitioning with range partitioning. The following example still creates 16 tablets, by first hashing the `id` column into 4 buckets, and then applying range partitioning to split each bucket into four tablets, based upon the value of the skustring. At least four tablets (and possibly up to 16) can be written to in parallel, and when you query for a contiguous range of sku values, you have a good chance of only needing to read from 1/4 of the tablets to fulfill the query.

```
CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT
)
DISTRIBUTE BY HASH (id) INTO 4 BUCKETS,
RANGE (sku) SPLIT ROWS(('g'), ('o'), ('u'))
TBLPROPERTIES(
'storage_handler' = 'com.cloudera.kudu.hive.KuduStorageHandler',
'kudu.table_name' = 'cust_behavior',
'kudu.master_addresses' = 'kudu-master.example.com:7051',
'kudu.key_columns' = 'id, sku'
);
```

Multiple `DISTRIBUTE BY HASH` Definitions. Again expanding the example above, suppose that the query pattern will be unpredictable, but you want to maximize parallelism of writes. You can achieve even distribution across the entire primary key by hashing on both primary key columns.

```
CREATE TABLE cust_behavior (
  id BIGINT,
  sku STRING,
  salary STRING,
  edu_level INT,
  usergender STRING,
  `group` STRING,
  city STRING,
  postcode STRING,
  last_purchase_price FLOAT,
  last_purchase_date BIGINT,
  category STRING,
  rating INT,
  fulfilled_date BIGINT
)
DISTRIBUTE BY HASH (id) INTO 4 BUCKETS, HASH (sku) INTO 4 BUCKETS
TBLPROPERTIES(
'storage_handler' = 'com.cloudera.kudu.hive.KuduStorageHandler',
'kudu.table_name' = 'cust_behavior',
'kudu.master_addresses' = 'kudu-master.example.com:7051',
'kudu.key_columns' = 'id, sku'
);
```

The example creates 16 buckets. You could also use `HASH (id, sku) INTO 16 BUCKETS`. However, a scan for sku values would almost always impact all 16 buckets, rather than possibly being limited to 4.

Impala Database Containment Model

Impala uses a database containment model. You can create a table within a specific scope, referred to as a database. To create the database, use a `CREATE DATABASE` statement. To use the database for further Impala operations such as `CREATE TABLE`, use the `USE` statement. For example, to create a table in a database called impala_kudu, use the following statements:

```
CREATE DATABASE impala_kudu;
USE impala_kudu;
CREATE TABLE my_first_table (
...
```

The my_first_table table is created within the impala_kudu database. To refer to this database in the future, without using a specific USE statement, you can refer to the table using<database>:<table> syntax. For example, to specify the my_first_table table in database impala_kudu, as opposed to any other table with the same name in another database, refer to the table as impala_kudu:my_first_table. This also applies to INSERT, UPDATE, DELETE, and DROP statements.

(**Warning**: As of this writing, Kudu does not encode the Impala database into the table name in any way. This means that even though you can create Kudu tables within Impala databases, the actual Kudu tables need to be unique within Kudu. For example, if you create database_1:my_kudu_table and database_2:my_kudu_table, you will have a naming collision within Kudu, even though this would not cause a problem in Impala.)

Impala Keywords Not Support for Kudu Tables

The following Impala keywords are not supported for Kudu tables:

- `PARTITIONED`
- `STORED AS`
- `LOCATION`
- `ROWFORMAT`

Understanding SQL Operators and Kudu

If your query includes the operators =, <=, or >=, Kudu evaluates the condition directly and only returns the relevant results. Kudu does not yet support <, >, !=, or any other operator not listed.

For these unsupported operations, Kudu returns all results regardless of the condition, and Impala performs the filtering. Since Impala must receive a larger amount of data from Kudu, these operations are less efficient. In some cases, creating and periodically updating materialized views may be the right solution to work around these inefficiencies.

Inserting a Row

The syntax for inserting one or more rows using Impala is shown below.

```
INSERT INTO my_first_table VALUES (99, "sarah");
INSERT INTO my_first_table VALUES (1, "john"), (2, "jane"), (3, "jim");
```

The primary key must not be null.

Inserting in Bulk

When insert in bulk, there are at least three common choices. Each may have advantages and disadvantages, depending on your data and circumstances.

- Multiple Single `INSERT` statements: This approach has the advantage of being easy to understand and implement. This approach is likely to be inefficient because Impala has a high query start-up cost compared to Kudu's insertion performance. This will lead to relatively high latency and poor throughput.
- Single `INSERT` statement with multiple `VALUES` subclauses: If you include more than 1024 `VALUES` statements, Impala batches them into groups of 1024 (or the value of batch_size) before sending the requests to Kudu. This approach may perform slightly better than multiple sequential `INSERT` statements by amortizing the query start-up penalties on the Impala side. To set the batch size for the current Impala Shell session, use this syntax: `set batch_size=10000;` . **Note**: Increasing the Impala batch size causes Impala to use more memory. You should verify the impact on your cluster and tune accordingly.)
  The "batch insert' approach that usually performs best, from the standpoint of both Impala and Kudu, is usually to import the data using a `SELECT FROM` subclause in Impala.
- If your data is not already in Impala, one strategy is to import it from a text file, such as a TSV or CSV file.
- Create the Kudu table, being mindful that the columns designated as primary keys cannot have null values.

- Insert values into the Kudu table by querying the table containing the original data, as in the following example:

  ```
  INSERT INTO my_kudu_table
  SELECT * FROM legacy_data_import_table;
  ```

- Ingest using the C++ or Java API: In many cases, the appropriate ingest path is to use the C++ or Java API to insert directly into Kudu tables. Unlike other Impala tables, data inserted into Kudu tables via the API becomes available for query in Impala without the need for any `INVALIDATE METADATA` statements or other statements needed for other Impala storage types.

INSERT and the IGNORE Keyword

Normally, if you try to insert a row that has already been inserted, the insertion will fail because the primary key would be duplicated (see "Failures During INSERT, UPDATE, and DELETE Operations".) If an insert fails part of the way through, you can re-run the insert, using the `IGNORE` keyword, which will ignore only those errors returned from Kudu indicating a duplicate key.

The first example will cause an error if a row with the primary key `99` already exists. The second example will still not insert the row, but will ignore any error and continue on to the next SQL statement.

```
INSERT INTO my_kudu_table
  SELECT * FROM legacy_data_import_table;
```

Updating a Row

The syntax for updating one or more rows using Impala is shown below.

```
UPDATE my_first_table SET name="bob" where id = 3;
```

You cannot change or null the primary key value. (**Important**: The `UPDATE` statement only works in Impala when the underlying data source is Kudu.)

Updating in Bulk

You can update in bulk using the same approaches outlined in "Inserting in Bulk" above.

UPDATE and the IGNORE Keyword

Similar to INSERT and the IGNORE Keyword, you can use the `IGNORE` operation to ignore an `UPDATE` which would otherwise fail. For instance, a row may be deleted while you are attempting to update it. In Impala, this would cause an error. The IGNORE keyword causes the error to be ignored.

```
DELETE IGNORE FROM my_first_table WHERE id < 3;
```

Deleting a Row

You can delete Kudu rows in near real time using Impala. You can even use more complex joins when deleting.

```
DELETE FROM my_first_table WHERE id < 3;
DELETE c FROM my_second_table c, stock_symbols s WHERE c.name = s.symbol;
```

Important: The DELETE statement only works in Impala when the underlying data source is Kudu.

Deleting in Bulk

You can delete in bulk using the same approaches outlined in “Inserting in Bulk” above.

DELETE and the IGNORE Keyword

Similar to INSERT and the IGNORE Keyword, you can use the `IGNORE` operation to ignore an `DELETE` which would otherwise fail. For instance, a row may be deleted by another process while you are attempting to delete it. In Impala, this would cause an error. The `IGNORE` keyword causes the error to be ignored.

```
DELETE IGNORE FROM my_first_table WHERE id > 3;
```

Failures During INSERT, UPDATE, and DELETE Operations

`INSERT`, `UPDATE`, and `DELETE` statements cannot be considered transactional as a whole. If one of these operations fails part of the way through, the keys may have already been created (in the case of `INSERT`) or the records may have already been modified or removed by another process (in the case of `UPDATE` or `DELETE`). You should design your application with this in mind. See INSERT and the IGNORE Keyword.

Altering Table Properties

You can change Impala’s metadata relating to a given Kudu table by altering the table’s properties. These properties include the table name, the list of Kudu master addresses, and whether the table is managed by Impala (internal) or externally. You cannot modify a table’s split rows after table creation. (**Important**: Altering table properties only changes Impala’s metadata about the table, not the underlying table itself. These statements do not modify any Kudu data.)

Rename a Table

```
ALTER TABLE my_table RENAME TO my_new_table;
```

Change the Kudu Master Addresses

```
ALTER TABLE my_table SET TBLPROPERTIES('kudu.master_addresses' = 'kudu-original-master.example.com:7051,kudu-new-master.example.com:7051');
```

Change an Internally-Managed Table to External

```
ALTER TABLE my_table SET TBLPROPERTIES('EXTERNAL' = 'TRUE');
```

Dropping a Table

If the table was created as an internal table in Impala, using `CREATE TABLE`, the standard `DROP TABLE` syntax drops the underlying Kudu table and all its data. If the table was created as an external table, using `CREATE EXTERNAL TABLE`, the mapping between Impala and Kudu is dropped, but the Kudu table is left intact, with all its data. To change an external table to internal, or vice versa, see Altering Table Properties.

```
DROP TABLE my_first_table;
```

Next Steps

The examples above have only explored a fraction of what you can do with Impala Shell. Read about Impala internals or learn how to contribute to Impala on the Impala Wiki.

Misty Stanley-Jones is a Technical Writer at Cloudera, and an Apache HBase committer.

How Impala Scales for Business Intelligence

By Yanpei Chen, Alan Choi, Dileep Kumar, David Rorke, Silvius Rus, and Devadutta Ghat (Oct. 2015)

Recent Impala testing demonstrates its scalability to a large number of concurrent users.

Impala, the open source MPP query engine designed for high-concurrency SQL over Apache Hadoop, has seen tremendous adoption across enterprises in industries such as financial services, telecom, healthcare, retail, gaming, government, and advertising. Impala has unlocked the ability to use business intelligence (BI) applications on Hadoop; these applications support critical business needs such as data discovery, operational dashboards, and reporting. For example, one customer has proven that Impala scales to 80 queries/second, supporting 1,000+ web dashboard end-users with sub-second response time. Clearly, BI applications represent a good fit for Impala, and customers can support more users simply by enlarging their clusters.

Cloudera's previous testing already established that Impala is the clear winner among analytic SQL-on-Hadoop alternatives, and we will provide additional support for this claim soon. We also showed that Impala scales across cluster sizes for stand-alone queries. Future roadmap also aims to deliver significant performance improvements.

That said, there is scant public data about how Impala scales across a large range of concurrency and cluster sizes. The results described in this post aim to close that knowledge gap by demonstrating that Impala clusters of 5, 10, 20, 40, and 80 nodes will support an increasing number of users at interactive latency.

To summarize the findings:

- Enlarging the cluster proportionally increases the users supported and the system throughput.
- Once the cluster saturates, adding more users leads to proportional increase in query latency.
- Scaling is bound by CPU, memory capacity, and workload skew.

The following describes the technical details surrounding our results. We'll also cover the relevant metrics, desired behavior, and configurations required for concurrency scalability for analytic SQL-on-Hadoop. As always, we strongly encourage you to do your own testing to confirm these results, and all the tools you need to do so have been provided.

Test Setup

Cluster

For this round of testing, we used a 5-rack, 80-node cluster. To investigate behavior across different cluster sizes, we divided these machines into clusters of 5, 10, 20, 40, and 80 nodes. Each node has hardware considered typical for Hadoop:

- CPU: Dual-socket, 12-core, 24-threads Intel Xeon E5-2630L 2.00GHz processor
- Disk: 12 Hewlett-Packard spinning SATA disks with 7200 RPM and 2TB each
- Memory: 64GB RAM (below what Cloudera recommends to demonstrate that Impala scales out well even with moderate resources per node)
- Network: 10 Gbps Ethernet

The cluster runs on RHEL 6.4 with CDH 5.3.3 and Impala 2.1.3, which were the newest CDH and Impala versions available when this project began. Every worker node runs an Impala Daemon and a HDFS DataNode.

Workload

The workload involved is derived from TPC-DS. Although this workload is not an ideal match for self-service BI and analytics, and more closely mimics reporting use cases, TPC-DS is publicly available and thus allows others to reproduce our results.

- Data schema: Generated by TPC-DS data generator
- Data size: 15TB (scale factor 15,000 in TPC-DS data generator)
- Data format: Apache Parquet file, Snappy compression
- Queries: The "Interactive" queries from our previous blog posts about Impala performance. (See our post from May 2013 for details and Github for the queries themselves.)
- Load: Reproduces the spirit of the TPC-DS "throughput run"
- A configurable number of concurrent, continuous query streams
- Each stream submits queries one after another
- Different streams run the set of all queries once in randomized order

Each query stream corresponds to a user. This workload mimics a scenario where many concurrent users continuously submit queries in random order, one after another. This concurrency model is actually more demanding than is typical in real life, where there is usually a gap between successive queries as users spend some time thinking about query output before writing a new query.

We intentionally constrained the query set to interactive queries for two reasons:

An analysis of customer workloads across industry verticals indicates that interactive queries make up more than 90% of queries for analytic SQL-on-Hadoop. Thus, concurrency scale will be driven by the concurrency level of interactive queries.

- Each of these interactive queries touches 80GB of data on average after partition pruning. These queries are non-trivial.
- We recorded the average of three repeated measurements for each concurrency setting.

Performance Goals

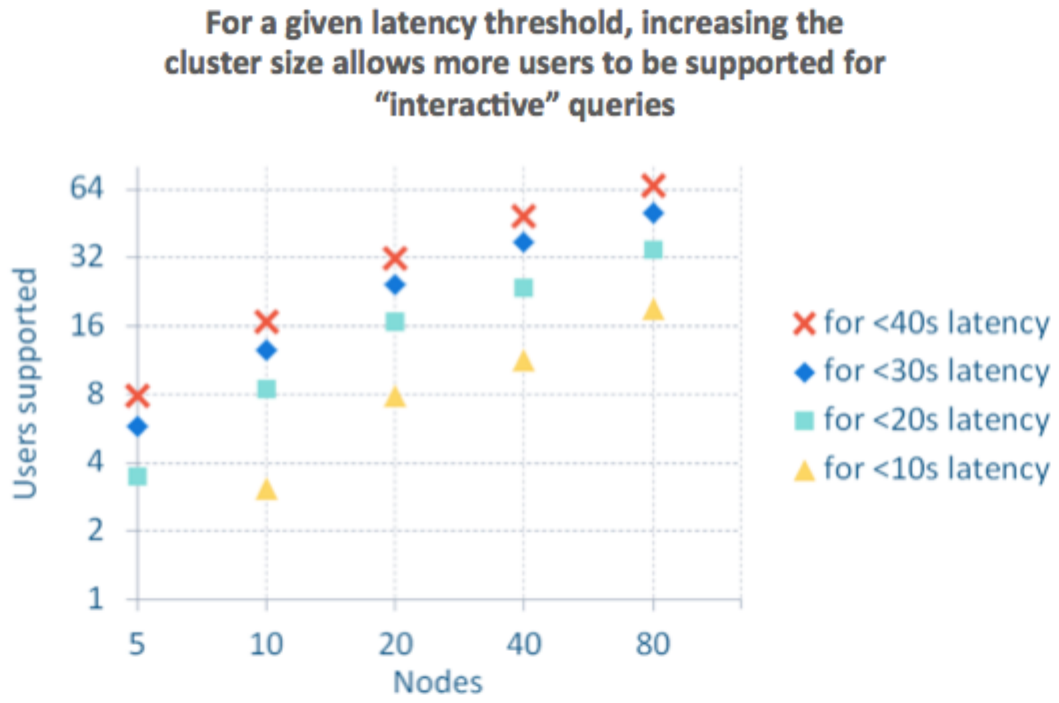
A system with good concurrency scalability should have the following characteristics:

- Uses all available hardware resources: As more users access the cluster, query throughput maximizes at a saturation point where some cluster hardware resource is fully utilized.
- High performance: At saturation point, query latency is low and throughput is high.
- Scalable in the number of users: Adding users after saturation leads to proportionally increasing latency without compromising throughput.
- Scalable in cluster size: Adding hardware to the system leads to proportionally increasing throughput and decreasing latency.

Results

Users Supported Scales with Cluster Size

For each cluster size, we continuously added more users until the latency increased beyond a set threshold of “interactive latency.” For each cluster size, we documented the number of users that makes latency cross the threshold.

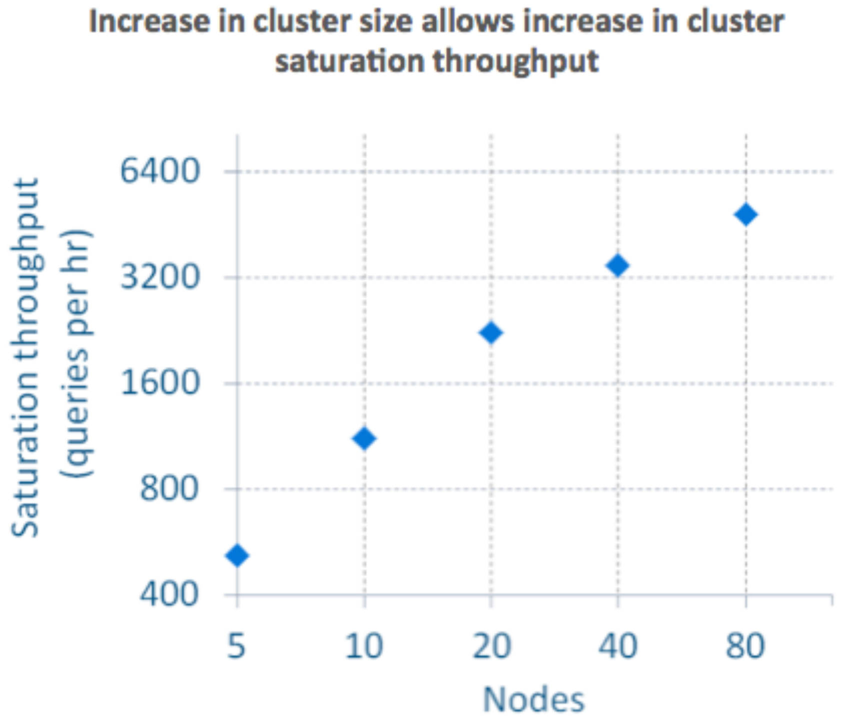


As you can see, for a given threshold, increasing the cluster size results in increasing the number of users supported. The general shape of the graphs is identical for different thresholds. For a fixed size, a cluster can add more users with increasing latency, allowing the cluster to support many users before exceeding a given threshold.

This result is impressive. It supports the contention that to maintain low query latency while adding more users, you would simply add more nodes to the cluster.

Saturation Throughput Scales with Cluster Size

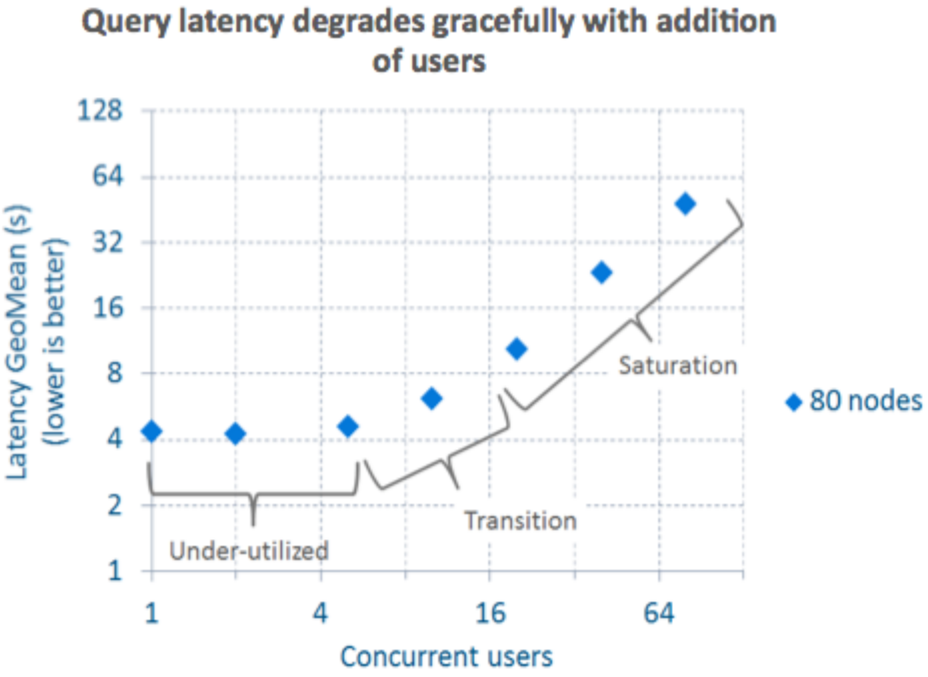
Cluster saturation throughput is another important performance metric. When there is a large number of users (the cluster is saturated), you want larger clusters to run through the queries proportionally faster. The results indicate that this is indeed the case with Impala.



Again, the results indicate that increasing the cluster size will allow you to increase the cluster saturation throughput.

Cluster Behavior as More Users are Added

A different view on the data allows one to identify distinct cluster operating regions as more users are added. The results below show query latency on the 80-node cluster as we progressively add more users.



There is initially a region where the cluster is under-utilized, and query latency remains constant even as we add more users. At the extreme right of the graph, there is a saturation region, where adding more users results in proportionally longer query latency. There is also a transition region in between.

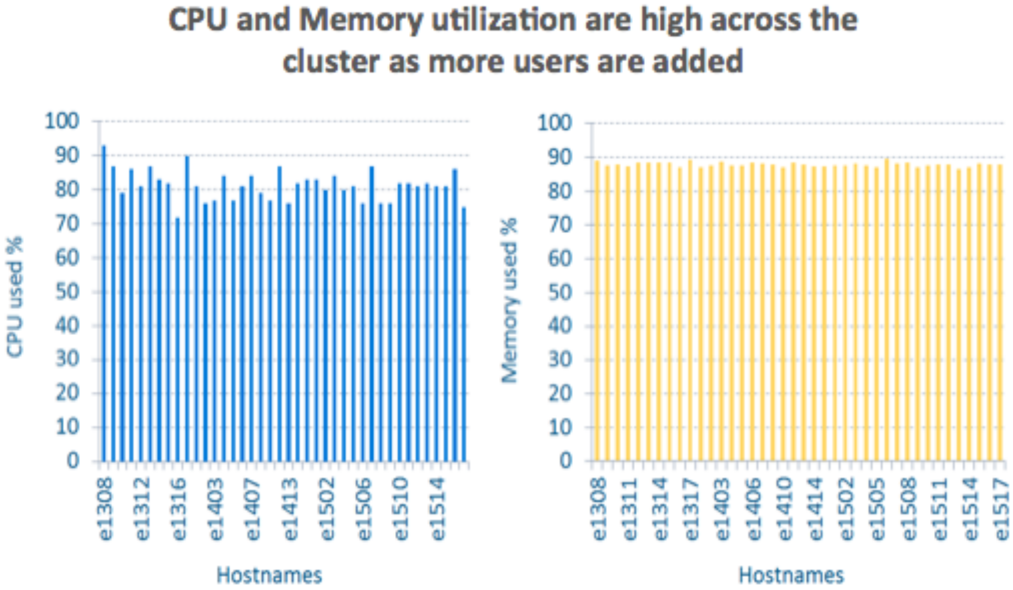
The shape of the graph on the right side is important because it indicates gracefully degrading query latency. So although fluctuations in real-life workloads can often take the cluster beyond saturation, in those conditions, query latencies would not become pathologically high.

Performance Limited by CPU and Memory Capacity

The cluster is both CPU and memory bound. Thus, Impala is efficiently utilizing the available hardware resources.

The graph below on the left shows the CPU utilization across the 40-nodes cluster when we're running 40 concurrent users or query streams, when the cluster is saturated. The cluster is CPU bound on several nodes that have 90% or higher CPU utilization. The variation in CPU utilization is due to execution skew (more about that below).

The graph on the right shows the memory utilization. It's more uniform across the cluster, and caps at around 90% utilized.



CPU and Memory utilization are high across the cluster as more users are added

Why Admission Control is Necessary

Previously you saw that when we add more users to a cluster, we get gracefully degrading query latency. We achieve this behavior by configuring Admission Control.

Impala's Admission Control feature maintains the cluster in an efficient and not-overwhelmed state by limiting the number of queries that can run at the same time. When users submit queries beyond the limit, the cluster puts the additional queries in a waiting queue. More users means a longer queue and longer waiting time, but the actual "running time" is the same. That is how we achieve graceful latency degradation.

The following is a conservative heuristic to set the admission control limit. It requires running typical queries in the workload in a stand-alone fashion, then finding the per-node peak memory use from the query profiles.

```
Set admission control limit =
(RAM size – headroom for OS and other CDH services) /
(max per-node peak memory use across all queries) *
(safety factor < 1)
```

We used 5GB for headroom for OS and other CDH services, and a safety factor value of 0.7.

On larger clusters, the same queries result in lower values for the per-node peak memory use, and this heuristic will give higher limits for Admission Control. Again, this heuristic is conservative.

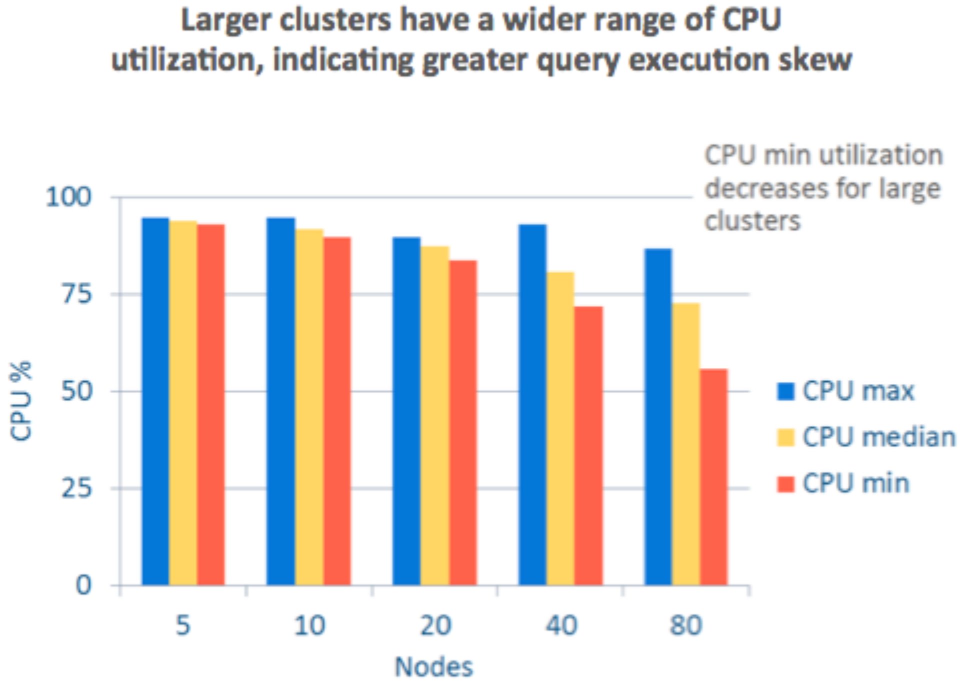
Scaling Overhead and Execution Skew

A query engine that scales simplifies cluster planning and operations: To add more users while maintaining query latency, just add more nodes.

However, the behavior is not ideal. In our tests, increasing the cluster size Nx resulted in below-Nx increase in the number of users supported at the same latency.

One reason for this scaling overhead is skew in how the workload is executed. This skew is visible in the CPU utilization graph above, where some nodes are CPU-bound while others have spare CPU capacity.

The graph below shows what happens when we vary the cluster size. On small clusters, there is almost no gap between maximum and minimum CPU utilization across the cluster. On large clusters, there is a large gap. The overall performance is bottlenecked on the slowest node. The bigger the gap, the bigger the skew, and the bigger the scaling overhead.

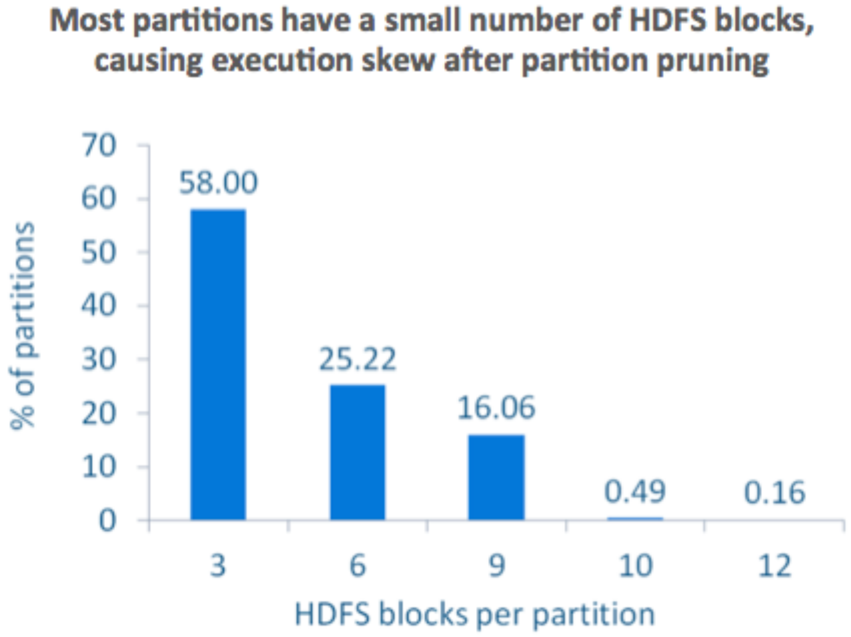


Larger clusters have a wider range of CPU utilization, indicating greater query execution skew

CPU min utilization decreases for large clusters

The primary source of execution skew occurs during the fact table scan HDFS operator. It arises out of uneven data placement across different nodes.

The fact table is partitioned by date. Many of the queries filter by date ranges in a month, so all but 30 partitions will be pruned. Data in partitions are stored as Parquet files, each a 256MB granularity HDFS block. Most partitions have three blocks, and the average is 4.7 blocks per partition (see graph below).



The net effect is that after partition pruning, there will be around 30 partitions remaining and 90-150 Parquet blocks that will be scanned across the cluster. On an 80-node cluster, most nodes will scan one or two blocks, and some nodes could end up scanning three or more blocks or zero blocks. This is a source of heavy skew. Smaller clusters would have on average more blocks per node, and statistically smooth out the node-to-node variation.

We verified the behavior by examining the query profiles. This skew impacts all queries, and propagates through the rest of the query execution after the fact table scan HDFS operator.

Summary and Recommendations

For customers running BI applications, a good analytic SQL-on-Hadoop backend should have the following properties:

- **Scales better to large clusters.** As datasets and the number of users grow, clusters will also grow. Solutions that can prove themselves at large cluster sizes are better.
- **Achieves fast, interactive latency.** This enables human specialists to explore the data, discover new ideas, then validate those ideas, all without waiting and losing their train of thought.
- **Makes efficient use of hardware – CPU as well as memory.** One can always buy bigger machines and build larger clusters. However, an efficient solution will support more users from the hardware available.

- **Simplifies planning**. Adding more users should not require a complete redesign of the system, or migrate wholesale to a different platform. Supporting more users should be a simple matter of adding more nodes to your cluster.

The results presented here show that Impala achieves the above goals. In these tests, we found that Impala shows good scaling behavior, with increasing cluster sizes being able to support increasing throughput and number of users. For this workload, Impala performance is limited by available CPU and memory capacity, and skew in how data is placed across the cluster. These results fall in line with the scaling behavior that most customers see.

We also have some general recommendations for analytic SQL-on-Hadoop software and hardware vendors:

- **Concurrency scale** should get more attention. BI use cases highlight the need to design for a large number of users. The technical challenges at high concurrency, such as load balancing to address execution skew, are only starting to be discovered.

- **Admission Control** is an important design point. Our heuristic is based on the number of queries, and would be less effective when some queries require a lot more processing than others and are a lot "bigger" than others.

- Analytic SQL-on-Hadoop engines should prioritize **CPU efficiency**. Each user adds incremental CPU demands, so an engine should aim to execute each query with as little CPU work as possible.

- Hardware needs **memory as well as CPU capacity**. CPU capacity is necessary as just discussed. Memory capacity is also needed, because data sizes and working sets will increase over time.

- Overall, concurrency and cluster scalability involves an interplay between hardware properties, software configurations, and the workload to be serviced.

We have seen only the tip of the iceberg for this complex problem space. Look for our future posts for more information!

Acknowledgements

The authors would like to thank Arun Singla for contributing to an earlier version of this work, and Ippokratis Pandis for in-depth comments during review.

Yanpei Chen, Alan Choi, Dileep Kumar, and David Rorke are Software Engineers at Cloudera.

Silvius Rus is a Director of Engineering at Cloudera.

Devadutta Ghat is a Senior Product Manager at Cloudera.

How-to: Prepare Unstructured Data in Impala for Analysis

By John Russell (Sept. 2015)

Learn how to build an Impala table around data that comes from non-Impala, or even non-SQL, sources.

As data pipelines start to include more aspects such as NoSQL or loosely specified schemas, you might encounter situations where you have data files (particularly in Apache Parquet format) where you do not know the precise table definition. This tutorial shows how you can build an Impala table around data that comes from non-Impala or even non-SQL sources, where you do not have control of the table layout and might not be familiar with the characteristics of the data.

The data used in this tutorial represents airline on-time arrival statistics, from October 1987 through April 2008. (See the details on the 2009 ASA Data Expo web site.) You can also see the explanations of the columns; for purposes of this exercise, wait until after following the tutorial before examining the schema, to better simulate a real-life situation where you cannot rely on assumptions and assertions about the ranges and representations of data values.

Wrangling the Data

First, we download and unpack the data files. There are eight files totalling 1.4GB. Each file is less than 256MB in size.

```
$ wget -O airlines_parquet.tar.gz
https://www.dropbox.com/s/20ycbvhqsy2uaqv/airlines_parquet.tar.gz
...
Length: 1245204740 (1.2G) [application/octet-stream]
Saving to: “airlines_parquet.tar.gz”

2015-08-12 17:14:24 (23.6 MB/s) - “airlines_parquet.tar.gz” saved
[1245204740/1245204740]

$ tar xvzf airlines_parquet.tar.gz
airlines_parquet/
airlines_parquet/93459d994898a9ba-77674173b331fa9a_2073981944_data.0.parq
airlines_parquet/93459d994898a9ba-77674173b331fa99_1555718317_data.1.parq
airlines_parquet/93459d994898a9ba-77674173b331fa99_1555718317_data.0.parq
airlines_parquet/93459d994898a9ba-77674173b331fa96_2118228804_data.0.parq
airlines_parquet/93459d994898a9ba-77674173b331fa97_574780876_data.0.parq
airlines_parquet/93459d994898a9ba-77674173b331fa96_2118228804_data.1.parq
airlines_parquet/93459d994898a9ba-77674173b331fa98_1194408366_data.0.parq
airlines_parquet/93459d994898a9ba-77674173b331fa9b_1413430552_data.0.parq
$ cd airlines_parquet/
$ du -kch *.parq
253M  93459d994898a9ba-77674173b331fa96_2118228804_data.0.parq
```

```
65M 93459d994898a9ba-77674173b331fa96_2118228804_data.1.parq
156M  93459d994898a9ba-77674173b331fa97_574780876_data.0.parq
240M  93459d994898a9ba-77674173b331fa98_1194408366_data.0.parq
253M  93459d994898a9ba-77674173b331fa99_1555718317_data.0.parq
16M 93459d994898a9ba-77674173b331fa99_1555718317_data.1.parq
177M  93459d994898a9ba-77674173b331fa9a_2073981944_data.0.parq
213M  93459d994898a9ba-77674173b331fa9b_1413430552_data.0.parq
1.4G  total
```

Next, we put the Parquet data files in HDFS, all together in a single directory, with permissions on the directory and the files so that the `impala` user will be able to read them. (Note: After unpacking, we saw the largest Parquet file was 253MB. When copying Parquet files into HDFS for Impala to use, for maximum query performance, make sure that each file resides in a single HDFS data block. Therefore, we pick a size larger than any single file and specify that as the block size, using the argument `-Ddfs.block.size=256m` on the `hdfs dfs -put` command.)

```
$ hdfs dfs -mkdir -p
hdfs://demo_host.example.com:8020/user/impala/staging/airlines
$ hdfs dfs -Ddfs.block.size=256m -put *.parq /user/impala/staging/airlines
$ hdfs dfs -ls /user/impala/staging
Found 1 items
drwxrwxrwx   - hdfs supergroup          0 2015-08-12 13:52
/user/impala/staging/airlines
$ hdfs dfs -ls hdfs://demo_host.example.com:8020/user/impala/staging/airlines
Found 8 items
-rw-r--r--   3 jrussell supergroup  265107489 2015-08-12 17:18
/user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa96_2118228804_data.0.parq
-rw-r--r--   3 jrussell supergroup   67544715 2015-08-12 17:18
/user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa96_2118228804_data.1.parq
-rw-r--r--   3 jrussell supergroup  162556490 2015-08-12 17:18
/user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa97_574780876_data.0.parq
-rw-r--r--   3 jrussell supergroup  251603518 2015-08-12 17:18
/user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa98_1194408366_data.0.parq
-rw-r--r--   3 jrussell supergroup  265186603 2015-08-12 17:18
/user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa99_1555718317_data.0.parq
-rw-r--r--   3 jrussell supergroup   16663754 2015-08-12 17:18
/user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa99_1555718317_data.1.parq
-rw-r--r--   3 jrussell supergroup  185511677 2015-08-12 17:18
```

```
 /user/impala/staging/airlines/93459d994898a9ba-
 77674173b331fa9a_2073981944_data.0.parq
 -rw-r--r--   3 jrussell supergroup  222794621 2015-08-12 17:18
 /user/impala/staging/airlines/93459d994898a9ba-
 77674173b331fa9b_1413430552_data.0.parq
```

With the files in an accessible location in HDFS, we create a database table that uses the data in those files. The `CREATE EXTERNAL` syntax and the `LOCATION` attribute point Impala at the appropriate HDFS directory. The `LIKE PARQUET '`*path_to_any_parquet_file*`'` clause means we skip the list of column names and types; Impala automatically gets the column names and data types straight from the data files. (Currently, this technique only works for Parquet files.) We ignore the warning about lack of `READ_WRITE` access to the files in HDFS; the `impala` user can read the files, which will be sufficient for us to experiment with queries and perform some copy and transform operations into other tables.

```
$ impala-shell -i localhost
Starting Impala Shell without Kerberos authentication
Connected to localhost:21000
Server version: impalad version 2.2.0-cdh5 RELEASE (build
2ffd73a4255cefd521362ffe1cfb37463f67f75c)
Welcome to the Impala shell. Press TAB twice to see a list of available
commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v2.1.2-cdh5 (92438b7) built on Tue Feb 24
12:36:33 PST 2015)
[localhost:21000] > create database airline_data;
[localhost:21000] > use airline_data;
[localhost:21000] > create external table airlines_external
                  > like parquet
'hdfs://demo_host.example.com:8020/user/impala/staging/airlines/93459d994898a
9ba-77674173b331fa96_2118228804_data.0.parq'
                  > stored as parquet location
'hdfs://demo_host.example.com:8020/user/impala/staging/airlines';
WARNINGS: Impala does not have READ_WRITE access to path
'hdfs://demo_host.example.com:8020/user/impala/staging'
```

Confirming the Data

With the table created, we examine its physical and logical characteristics to confirm that the data is really there and in a format and shape that we can work with. The `SHOW TABLE STATS` statement gives a very high-level summary of the table, showing how many files and how much total data it contains. Also, it confirms that the table is expecting all the associated data files to be in Parquet format. (The ability to work with all kinds of HDFS data files in different formats means that it is possible to have a mismatch between the format of the data files, and the format that the table expects the data files to be in.)

The `SHOW FILES` statement confirms that the data in the table has the expected number, names, and sizes of the original Parquet files. The `DESCRIBE` statement (or its abbreviation `DESC`) confirms the names and types of the columns that Impala automatically created after reading that metadata from the Parquet file. The `DESCRIBE FORMATTED` statement prints out some extra detail along with the column definitions; the pieces we care about for this exercise are the containing database for the table, the location of the associated data files in HDFS, the fact that it's an external table so Impala will not delete the HDFS files when we finish the experiments and drop the table, and the fact that the table is set up to work exclusively with files in the Parquet format.

```
[localhost:21000] > show table stats airlines_external;
+-------+--------+--------+--------------+-------------------+---------+----+
| #Rows | #Files | Size   | Bytes Cached | Cache Replication | Format  |
Incremental stats |
+-------+--------+--------+--------------+-------------------+---------+----+
| -1    | 8      | 1.34GB | NOT CACHED   | NOT CACHED        | PARQUET |
false             |
+-------+--------+--------+--------------+-------------------+---------+----+
[localhost:21000] > show files in airlines_external;
+---------------------------------------------------------------------------+
| path
| size     | partition |
+---------------------------------------------------------------------------+
| /user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa96_2118228804_data.0.parq | 252.83MB |           |
| /user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa96_2118228804_data.1.parq | 64.42MB  |           |
| /user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa97_574780876_data.0.parq  | 155.03MB |           |
| /user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa98_1194408366_data.0.parq | 239.95MB |           |
| /user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa99_1555718317_data.0.parq | 252.90MB |           |
| /user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa99_1555718317_data.1.parq | 15.89MB  |           |
| /user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa9a_2073981944_data.0.parq | 176.92MB |           |
| /user/impala/staging/airlines/93459d994898a9ba-
77674173b331fa9b_1413430552_data.0.parq | 212.47MB |           |
+---------------------------------------------------------------------------+
[localhost:21000] > describe airlines_external;
+---------------------+--------+---------------------------------------------+
| name                | type   | comment
|
+---------------------+--------+---------------------------------------------+
| year                | int    | inferred from: optional int32 year
|
| month               | int    | inferred from: optional int32 month
|
| day                 | int    | inferred from: optional int32 day
|
| dayofweek           | int    | inferred from: optional int32 dayofweek
|
| dep_time            | int    | inferred from: optional int32 dep_time
|
| crs_dep_time        | int    | inferred from: optional int32 crs_dep_time
|
| arr_time            | int    | inferred from: optional int32 arr_time
|
| crs_arr_time        | int    | inferred from: optional int32 crs_arr_time
|
| carrier             | string | inferred from: optional binary carrier
|
| flight_num          | int    | inferred from: optional int32 flight_num
|
| tail_num            | int    | inferred from: optional int32 tail_num
|
| actual_elapsed_time | int    | inferred from: optional int32
actual_elapsed_time |
| crs_elapsed_time    | int    | inferred from: optional int32
crs_elapsed_time    |
| airtime             | int    | inferred from: optional int32 airtime
|
| arrdelay            | int    | inferred from: optional int32 arrdelay
|
| depdelay            | int    | inferred from: optional int32 depdelay
|
| origin              | string | inferred from: optional binary origin
|
| dest                | string | inferred from: optional binary dest
|
| distance            | int    | inferred from: optional int32 distance
|
| taxi_in             | int    | inferred from: optional int32 taxi_in
|
| taxi_out            | int    | inferred from: optional int32 taxi_out
|
| cancelled           | int    | inferred from: optional int32 cancelled
|
| cancellation_code   | string | inferred from: optional binary
cancellation_code  |
| diverted            | int    | inferred from: optional int32 diverted
|
| carrier_delay       | int    | inferred from: optional int32 carrier_delay
|
| weather_delay       | int    | inferred from: optional int32 weather_delay
|
| nas_delay           | int    | inferred from: optional int32 nas_delay
|
| security_delay      | int    | inferred from: optional int32 security_delay
|
| late_aircraft_delay | int    | inferred from: optional int32
late_aircraft_delay |
+---------------------+--------+---------------------------------------------+
[localhost:21000] > desc formatted airlines_external;
+------------------------------+-------------------------------
| name                         | type
+------------------------------+-------------------------------
...
| # Detailed Table Information | NULL
| Database:                    | airline_data
| Owner:                       | jrussell
...
| Location:                    | /user/impala/staging/airlines
| Table Type:                  | EXTERNAL_TABLE
...
| # Storage Information        | NULL
| SerDe Library:               | parquet.hive.serde.ParquetHiveSerDe
| InputFormat:                 | parquet.hive.DeprecatedParquetInputFormat
| OutputFormat:                | parquet.hive.DeprecatedParquetOutputFormat
...
```

Running Queries

Now that we are confident that the connections are solid between the Impala table and the underlying Parquet files, we run some initial queries to understand the characteristics of the data: the overall number of rows, and the ranges and how many different values are in certain columns. For convenience in understanding the magnitude of the `COUNT(*)` result, we run another query dividing the number of rows by 1 million, demonstrating that there are 123 million rows in the table.

```
[localhost:21000] > select count(*) from airlines_external;
+-----------+
| count(*)  |
+-----------+
| 123534969 |
+-----------+
Fetched 1 row(s) in 1.32s
[localhost:21000] > select count(*) / 1e6 as 'millions of rows' from
airlines_external;
+------------------+
| millions of rows |
+------------------+
| 123.534969       |
+------------------+
Fetched 1 row(s) in 1.24s
```

The `NDV()` function stands for `number of distinct values`, which for performance reasons is an estimate when there are lots of different values in the column, but is precise when the cardinality is less than 16KB. Use `NDV()` calls for this kind of exploration rather than `COUNT(DISTINCT colname)`, because Impala can evaluate multiple `NDV()` functions in a single query, but only a single instance of `COUNT DISTINCT`.

Here we see that there are modest numbers of different airlines, flight numbers, and origin and destination airports. Two things jump out from this query: the number of `tail_num` values is much smaller than we might have expected, and there are more destination airports than origin airports. Let's dig further.

What we find is that most `tail_num` values are `NULL`. It looks like this was an experimental column that wasn't filled in accurately. We make a mental note that if we use this data as a starting point, we'll ignore this column. We also find that certain airports are represented in the `ORIGIN` column but not the `DEST` column; now we know that we can't rely on the assumption that those sets of airport codes are identical.

(Note: A slight digression for some performance tuning. Notice how the first `SELECT DISTINCT DEST` query takes almost 40 seconds. We expect all queries on such a small data set, less than 2GB, to take a few seconds at most. The reason is because the expression `NOT IN (SELECT origin FROM airlines_external)` produces an intermediate result set of 123 million rows, then runs 123 million comparisons on each data node against the tiny set of destination airports. The way the `NOT IN` operator works internally means that this intermediate result set with 123 million rows might be transmitted across the network to each data node in the cluster. Applying another `DISTINCT` inside the `NOT IN` subquery means that the intermediate result set is only 340 items, resulting in much less network traffic and fewer comparison operations. The more efficient query with the added `DISTINCT` is approximately seven times as fast.)

```
[localhost:21000] > select ndv(carrier), ndv(flight_num), ndv(tail_num),
                  >   ndv(origin), ndv(dest) from airlines_external;
+--------------+-----------------+---------------+-------------+-----------+
| ndv(carrier) | ndv(flight_num) | ndv(tail_num) | ndv(origin) | ndv(dest) |
+--------------+-----------------+---------------+-------------+-----------+
| 29           | 9086            | 3             | 340         | 347       |
+--------------+-----------------+---------------+-------------+-----------+
[localhost:21000] > select tail_num, count(*) as howmany from
airlines_external
                  >   group by tail_num;
+----------+-----------+
| tail_num | howmany   |
+----------+-----------+
| 715      | 1         |
| 0        | 406405    |
| 112      | 6562      |
| NULL     | 123122001 |
+----------+-----------+
Fetched 1 row(s) in 5.18s
[localhost:21000] > select distinct dest from airlines_external
                  >   where dest not in (select origin from
airlines_external);
+------+
| dest |
+------+
| LBF  |
| CBM  |
| RCA  |
| SKA  |
| LAR  |
+------+
Fetched 5 row(s) in 39.64s
```

```
[localhost:21000] > select distinct dest from airlines_external
                  >   where dest not in (select distinct origin from
airlines_external);
+-------+
| dest  |
+-------+
| LBF   |
| RCA   |
| CBM   |
| SKA   |
| LAR   |
+-------+
Fetched 5 row(s) in 5.59s
[localhost:21000] > select distinct origin from airlines_external
                  >   where origin not in (select distinct dest from
airlines_external);
Fetched 0 row(s) in 5.37s
```

Initial Exploration

Next, we try doing a simple calculation, with results broken down by year. This reveals that some years have no data in the `AIRTIME` column. That means we might be able to use that column in queries involving certain date ranges, but we can't count on it to always be reliable. The question of whether a column contains any `NULL` values, and if so what is their number, proportion, and distribution, comes up again and again when doing initial exploration of a data set.

```
[localhost:21000] > select year, sum(airtime) from airlines_external
                  >   group by year order by year desc;
+------+--------------+
| year | sum(airtime) |
+------+--------------+
| 2008 | 713050445    |
| 2007 | 748015545    |
| 2006 | 720372850    |
| 2005 | 708204026    |
| 2004 | 714276973    |
| 2003 | 665706940    |
| 2002 | 549761849    |
| 2001 | 590867745    |
| 2000 | 583537683    |
| 1999 | 561219227    |
| 1998 | 538050663    |
| 1997 | 536991229    |
| 1996 | 519440044    |
| 1995 | 513364265    |
| 1994 | NULL         |
| 1993 | NULL          |
| 1992 | NULL          |
| 1991 | NULL          |
| 1990 | NULL          |
| 1989 | NULL          |
| 1988 | NULL          |
| 1987 | NULL          |
```

With the notion of `NULL` values in mind, let's come back to the `TAILNUM` column that we discovered had a lot of `NULL`s. Let's quantify the `NULL` and non-`NULL` values in that column for better understanding.

First, we just count the overall number of rows versus the non-`NULL` values in that column. That initial result gives the appearance of relatively few non-`NULL` values, but we can break it down more clearly in a single query. Once we have the `COUNT(*)` and the `COUNT(`*colname*`)` numbers, we can encode that initial query in a `WITH` clause, then run a followon query that performs multiple arithmetic operations on those values. Seeing that only one-third of one percent of all rows have non-`NULL` values for the `TAILNUM` column clearly illustrates that that column won't be of much use.

```
[localhost:21000] > select count(*) as 'rows', count(tail_num) as 'non-null
tail numbers'
                  >   from airlines_external;
+-----------+-----------------------+
| rows      | non-null tail numbers |
+-----------+-----------------------+
| 123534969 | 412968                |
+-----------+-----------------------+
Fetched 1 row(s) in 1.51s
[localhost:21000] > with t1 as
                  >   (select count(*) as 'rows', count(tail_num) as
'nonnull'
                  >   from airlines_external)
                  > select `rows`, `nonnull`, `rows` - `nonnull` as 'nulls',
                  >   (`nonnull` / `rows`) * 100 as 'percentage non-null'
                  > from t1;
+-----------+---------+-----------+---------------------+
| rows      | nonnull | nulls     | percentage non-null |
+-----------+---------+-----------+---------------------+
| 123534969 | 412968  | 123122001 | 0.3342923897119365  |
+-----------+---------+-----------+---------------------+
```

By examining other columns using these techniques, we can form a mental picture of the way data is distributed throughout the table, and which columns are most significant for query purposes. For this tutorial, we focus mostly on the fields likely to hold discrete values, rather than columns such as `ACTUAL_ELAPSED_TIME` whose names suggest they hold measurements. We would dig deeper into those columns once we had a clear picture of which questions were worthwhile to ask, and what kinds of trends we might look for.

For the final piece of initial exploration, let's look at the `YEAR` column. A simple `GROUP BY` query shows that it has a well-defined range, a manageable number of distinct values, and relatively even distribution of rows across the different years.

```
[localhost:21000] > select min(year), max(year), ndv(year) from
airlines_external;
+-----------+-----------+-----------+
| min(year) | max(year) | ndv(year) |
+-----------+-----------+-----------+
| 1987      | 2008      | 22        |
+-----------+-----------+-----------+
Fetched 1 row(s) in 2.03s
[localhost:21000] > select year, count(*) howmany from airlines_external
                  >   group by year order by year desc;
+------+---------+
| year | howmany |
+------+---------+
| 2008 | 7009728 |
| 2007 | 7453215 |
| 2006 | 7141922 |
| 2005 | 7140596 |
| 2004 | 7129270 |
| 2003 | 6488540 |
| 2002 | 5271359 |
| 2001 | 5967780 |
| 2000 | 5683047 |
| 1999 | 5527884 |
| 1998 | 5384721 |
| 1997 | 5411843 |
| 1996 | 5351983 |
| 1995 | 5327435 |
| 1994 | 5180048 |
| 1993 | 5070501 |
| 1992 | 5092157 |
| 1991 | 5076925 |
| 1990 | 5270893 |
| 1989 | 5041200 |
| 1988 | 5202096 |
| 1987 | 1311826 |
+-------+---------+
Fetched 22 row(s) in 2.13s
```

We could go quite far with the data in this initial raw format, just as we downloaded it from the web. If the data set proved to be useful and worth persisting in Impala for extensive queries, we might want to copy it to an internal table, letting Impala manage the data files and perhaps reorganizing a little for higher efficiency.

Copying the Data into a Partitioned Table

Partitioning based on the `YEAR` column lets us run queries with clauses such as `WHERE year = 2001` or `WHERE year BETWEEN 1989 AND 1999`, which can dramatically cut down on I/O by ignoring all the data from years outside the desired range. Rather than reading all the data and then deciding which rows are in the matching years, Impala can zero in on only the data files from specific `YEAR` partitions. To do this, Impala physically reorganizes the data files, putting the rows from each year into data files in a separate HDFS directory for each `YEAR` value. Along the way, we'll also get rid of the `TAIL_NUM` column that proved to be almost entirely `NULL`.

The first step is to create a new table with a layout very similar to the original `AIRLINES_EXTERNAL` table. We'll do that by reverse-engineering a `CREATE TABLE` statement for the first table, then tweaking it slightly to include a `PARTITION BY` clause for `YEAR`, and excluding the `TAIL_NUM` column. The `SHOW CREATE TABLE` statement gives us the starting point.

```
[localhost:21000] > show create table airlines_external;
+----------------------------------------------------------------------------
| result
+----------------------------------------------------------------------------
| CREATE EXTERNAL TABLE airline_data.airlines_external (
|   year INT COMMENT 'inferred from: optional int32 year',
|   month INT COMMENT 'inferred from: optional int32 month',
|   day INT COMMENT 'inferred from: optional int32 day',
|   dayofweek INT COMMENT 'inferred from: optional int32 dayofweek',
|   dep_time INT COMMENT 'inferred from: optional int32 dep_time',
|   crs_dep_time INT COMMENT 'inferred from: optional int32 crs_dep_time',
|   arr_time INT COMMENT 'inferred from: optional int32 arr_time',
|   crs_arr_time INT COMMENT 'inferred from: optional int32 crs_arr_time',
|   carrier STRING COMMENT 'inferred from: optional binary carrier',
|   flight_num INT COMMENT 'inferred from: optional int32 flight_num',
|   tail_num INT COMMENT 'inferred from: optional int32 tail_num',
|   actual_elapsed_time INT COMMENT 'inferred from: optional int32
actual_elapsed_time',
|   crs_elapsed_time INT COMMENT 'inferred from: optional int32
crs_elapsed_time',
|   airtime INT COMMENT 'inferred from: optional int32 airtime',
|   arrdelay INT COMMENT 'inferred from: optional int32 arrdelay',
|   depdelay INT COMMENT 'inferred from: optional int32 depdelay',
|   origin STRING COMMENT 'inferred from: optional binary origin',
|   dest STRING COMMENT 'inferred from: optional binary dest',
|   distince INT COMMENT 'inferred from: optional int32 distince',
|   taxi_in INT COMMENT 'inferred from: optional int32 taxi_in',
|   taxi_out INT COMMENT 'inferred from: optional int32 taxi_out',
|   cancelled INT COMMENT 'inferred from: optional int32 cancelled',
|   cancellation_code STRING COMMENT 'inferred from: optional binary
cancellation_code',
|   diverted INT COMMENT 'inferred from: optional int32 diverted',
|   carrier_delay INT COMMENT 'inferred from: optional int32 carrier_delay',
|   weather_delay INT COMMENT 'inferred from: optional int32 weather_delay',
|   nas_delay INT COMMENT 'inferred from: optional int32 nas_delay',
|   security_delay INT COMMENT 'inferred from: optional int32
security_delay',
|   late_aircraft_delay INT COMMENT 'inferred from: optional int32
late_aircraft_delay'
| )
| STORED AS PARQUET
| LOCATION
'hdfs://a1730.halxg.cloudera.com:8020/user/impala/staging/airlines'
| TBLPROPERTIES ('numFiles'='0', 'COLUMN_STATS_ACCURATE'='false',
|   'transient_lastDdlTime'='1439425228', 'numRows'='-1', 'totalSize'='0',
|   'rawDataSize'='-1')
+-----------------------------------------------------------------------------
Fetched 1 row(s) in 0.03s
[localhost:21000] > quit;
```

Although we could edit that output into a new SQL statement, all the ASCII box characters make such editing inconvenient. To get a more stripped-down `CREATE TABLE` to start with, we restart the `impala-shell` command with the `-B` option, which turns off the box-drawing behavior.

```
[localhost:21000] > quit;
Goodbye jrussell
$ impala-shell -i localhost -B -d airline_data;
Starting Impala Shell without Kerberos authentication
Connected to localhost:21000
Server version: impalad version 2.2.0-cdh5 RELEASE (build
2ffd73a4255cefd521362ffe1cfb37463f67f75c)
Welcome to the Impala shell. Press TAB twice to see a list of available
commands.
```

```
Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v2.1.2-cdh5 (92438b7) built on Tue Feb 24
12:36:33 PST 2015)
[localhost:21000] > show create table airlines_external;
"CREATE EXTERNAL TABLE airline_data.airlines_external (
  year INT COMMENT 'inferred from: optional int32 year',
  month INT COMMENT 'inferred from: optional int32 month',
  day INT COMMENT 'inferred from: optional int32 day',
  dayofweek INT COMMENT 'inferred from: optional int32 dayofweek',
  dep_time INT COMMENT 'inferred from: optional int32 dep_time',
  crs_dep_time INT COMMENT 'inferred from: optional int32 crs_dep_time',
  arr_time INT COMMENT 'inferred from: optional int32 arr_time',
  crs_arr_time INT COMMENT 'inferred from: optional int32 crs_arr_time',
  carrier STRING COMMENT 'inferred from: optional binary carrier',
  flight_num INT COMMENT 'inferred from: optional int32 flight_num',
  tail_num INT COMMENT 'inferred from: optional int32 tail_num',
  actual_elapsed_time INT COMMENT 'inferred from: optional int32
actual_elapsed_time',
  crs_elapsed_time INT COMMENT 'inferred from: optional int32
crs_elapsed_time',
  airtime INT COMMENT 'inferred from: optional int32 airtime',
  arrdelay INT COMMENT 'inferred from: optional int32 arrdelay',
  depdelay INT COMMENT 'inferred from: optional int32 depdelay',
  origin STRING COMMENT 'inferred from: optional binary origin',
  dest STRING COMMENT 'inferred from: optional binary dest',
  distance INT COMMENT 'inferred from: optional int32 distance',
  taxi_in INT COMMENT 'inferred from: optional int32 taxi_in',
  taxi_out INT COMMENT 'inferred from: optional int32 taxi_out',
  cancelled INT COMMENT 'inferred from: optional int32 cancelled',
  cancellation_code STRING COMMENT 'inferred from: optional binary
cancellation_code',
  diverted INT COMMENT 'inferred from: optional int32 diverted',
  carrier_delay INT COMMENT 'inferred from: optional int32 carrier_delay',
  weather_delay INT COMMENT 'inferred from: optional int32 weather_delay',
  nas_delay INT COMMENT 'inferred from: optional int32 nas_delay',
  security_delay INT COMMENT 'inferred from: optional int32 security_delay',
  late_aircraft_delay INT COMMENT 'inferred from: optional int32
late_aircraft_delay'
)
STORED AS PARQUET
LOCATION 'hdfs://a1730.halxg.cloudera.com:8020/user/impala/staging/airlines'
TBLPROPERTIES ('numFiles'='0', 'COLUMN_STATS_ACCURATE'='false',
  'transient_lastDdlTime'='1439425228', 'numRows'='-1', 'totalSize'='0',
  'rawDataSize'='-1')"
Fetched 1 row(s) in 0.01s
```

After copying and pasting the `CREATE TABLE` statement into a text editor for fine-tuning, we quit and restart `impala-shell` without the `-B` option, to switch back to regular output.

Next we run the `CREATE TABLE` statement that we adapted from the `SHOW CREATE TABLE` output. We kept the `STORED AS PARQUET` clause because we want to rearrange the data somewhat but still keep it in the high-performance Parquet format.
The `LOCATION` and `TBLPROPERTIES` clauses are not relevant for this new table, so we edit those out. Because we are going to partition the new table based on the `YEAR` column, we move that column name (and its type) into a new `PARTITIONED BY` clause.

```
[localhost:21000] > CREATE TABLE airline_data.airlines
                  > (
                  >   month INT,
                  >   day INT,
                  >   dayofweek INT,
                  >   dep_time INT,
                  >   crs_dep_time INT,
                  >   arr_time INT,
                  >   crs_arr_time INT,
                  >   carrier STRING,
                  >   flight_num INT,
                  >   actual_elapsed_time INT,
                  >   crs_elapsed_time INT,
                  >   airtime INT,
                  >   arrdelay INT,
                  >   depdelay INT,
                  >   origin STRING,
                  >   dest STRING,
                  >   distance INT,
                  >   taxi_in INT,
                  >   taxi_out INT,
                  >   cancelled INT,
                  >   cancellation_code STRING,
                  >   diverted INT,
                  >   carrier_delay INT,
                  >   weather_delay INT,
                  >   nas_delay INT,
                  >   security_delay INT,
                  >   late_aircraft_delay INT
                  > )
                  > STORED AS PARQUET
                  > PARTITIONED BY (year INT);
Fetched 0 row(s) in 0.10s
```

Next, we copy all the rows from the original table into this new one with an `INSERT` statement. (We edited the `CREATE TABLE` statement to make an `INSERT` statement with the column names in the same order.) The only change is to add a `PARTITION(year)` clause, and move the `YEAR` column to the very end of the `SELECT` list of the `INSERT` statement. Specifying `PARTITION(year)`, rather than a fixed value such as `PARTITION(year=2000)`, means that Impala figures out the partition value for each row based on the value of the very last column in the `SELECT` list.

This is the first SQL statement that legitimately takes any substantial time, because the rows from different years are shuffled around the cluster; the rows that go into each partition are collected on one node, before being written to one or more new data files.

```
[localhost:21000] > INSERT INTO airline_data.airlines
                  > PARTITION (year)
                  > SELECT
                  >   month,
                  >   day,
                  >   dayofweek,
                  >   dep_time,
                  >   crs_dep_time,
                  >   arr_time,
                  >   crs_arr_time,
                  >   carrier,
                  >   flight_num,
                  >   actual_elapsed_time,
                  >   crs_elapsed_time,
                  >   airtime,
                  >   arrdelay,
                  >   depdelay,
                  >   origin,
                  >   dest,
                  >   distance,
                  >   taxi_in,
                  >   taxi_out,
                  >   cancelled,
                  >   cancellation_code,
                  >   diverted,
                  >   carrier_delay,
                  >   weather_delay,
                  >   nas_delay,
                  >   security_delay,
                  >   late_aircraft_delay,
                  >   year
                  > FROM airline_data.airlines_external;
Inserted 123534969 row(s) in 202.70s
```

Once partitioning or join queries come into play, it's important to have statistics that Impala can use to optimize queries on the corresponding tables. The `COMPUTE INCREMENTAL STATS` statement is the way to collect statistics for partitioned tables. Then the `SHOW TABLE STATS` statement confirms that the statistics are in place for each partition, and also illustrates how many files and how much raw data is in each partition.

```
[localhost:21000] > compute incremental stats airlines;
+-------------------------------------------+
| summary                                   |
+-------------------------------------------+
| Updated 22 partition(s) and 27 column(s). |
+-------------------------------------------+
[localhost:21000] > show table stats airlines;
+-------+-----------+--------+----------+--------------+------------+--------
-+--------------------+
| year  | #Rows     | #Files | Size     | Bytes Cached | Cache Repl | Format
| Incremental stats |
+-------+-----------+--------+----------+--------------+------------+--------
-+-----
| 1987  | 1311826   | 1      | 9.32MB   | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1988  | 5202096   | 1      | 37.04MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1989  | 5041200   | 1      | 36.25MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1990  | 5270893   | 1      | 38.39MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1991  | 5076925   | 1      | 37.23MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1992  | 5092157   | 1      | 36.85MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1993  | 5070501   | 1      | 37.16MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1994  | 5180048   | 1      | 38.31MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1995  | 5327435   | 1      | 53.14MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1996  | 5351983   | 1      | 53.64MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1997  | 5411843   | 1      | 54.41MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1998  | 5384721   | 1      | 54.01MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 1999  | 5527884   | 1       | 56.32MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 2000  | 5683047   | 1       | 58.15MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 2001  | 5967780   | 1       | 60.65MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 2002  | 5271359   | 1       | 57.99MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 2003  | 6488540   | 1       | 81.33MB  | NOT CACHED   | NOT CACHED | PARQUET
| true
| 2004  | 7129270   | 1       | 103.19MB | NOT CACHED   | NOT CACHED | PARQUET
| true
| 2005  | 7140596   | 1       | 102.61MB | NOT CACHED   | NOT CACHED | PARQUET
| true
| 2006  | 7141922   | 1       | 106.03MB | NOT CACHED   | NOT CACHED | PARQUET
| true
| 2007  | 7453215   | 1       | 112.15MB | NOT CACHED   | NOT CACHED | PARQUET
| true
| 2008  | 7009728   | 1       | 105.76MB | NOT CACHED   | NOT CACHED | PARQUET
| true
| Total | 123534969 | 22      | 1.30GB   | 0B           |            |
|
+-------+-----------+---------+----------+--------------+------------+-------+
```

At this point, we go through a quick thought process to sanity check the partitioning we did. All the partitions have exactly one file, which is on the low side. A query that includes a clause `WHERE year=2004` will only read a single data block; that data block will be read and processed by a single data node; therefore, for a query targeting a single year, all the other nodes in the cluster will sit idle while all the work happens on a single machine. It's even possible that by chance (depending on HDFS replication factor and the way data blocks are distributed across the cluster), that multiple year partitions selected by a filter such as `WHERE year BETWEEN 1999 AND 2001` could all be read and processed by the same data node. The more data files each partition has, the more parallelism you can get and the less probability of "hotspots" occurring on particular nodes, therefore a bigger performance boost by having a big CDH cluster.

However, the more data files, the less data goes in each one. The overhead of dividing the work in a parallel query might not be worth it if each node is only reading a few megabytes. 50 or 100MB is a decent size for a Parquet data block; 9 or 37MB is on the small side. Which is to say, the data distribution we ended up with based on this partitioning scheme is on the borderline between sensible (reasonably large files) and suboptimal (few files in each partition). The way to see how well it works in practice is to run the same queries against the original flat table and the new partitioned table, and compare times.

Spoiler: in this case, with my particular four-node cluster with its specific distribution of data blocks and my particular exploratory queries, queries against the partitioned table do consistently run faster than the same queries against the unpartitioned table. But I could not be sure that would be the case without some real measurements. Here are some queries I ran to draw that conclusion, first against `AIRLINES_EXTERNAL` (no partitioning), then against `AIRLINES` (partitioned by year). The `AIRLINES` queries are consistently faster. Changing the volume of data, changing the size of the cluster, running queries that did or didn't refer to the partition key columns, or other factors could change the results to favor one table layout or the other.

(Note: If you find the volume of each partition is only in the low tens of megabytes, consider lowering the granularity of partitioning. For example, instead of partitioning by year, month, and day, partition by year and month or even just by year. The ideal layout to distribute work efficiently in a parallel query is many tens or even hundreds of megabytes per Parquet file, and the number of Parquet files in each partition somewhat higher than the number of data nodes.)

```
[localhost:21000] > select sum(airtime) from airlines_external;
+--------------+
| sum(airtime) |
+--------------+
| 8662859484   |
+--------------+
Fetched 1 row(s) in 2.02s
[localhost:21000] > select sum(airtime) from airlines;
+--------------+
| sum(airtime) |
+--------------+
| 8662859484   |
+--------------+
Fetched 1 row(s) in 1.21s

[localhost:21000] > select sum(airtime) from airlines_external where year =
2005;
+--------------+
| sum(airtime) |
+--------------+
| 708204026    |
+--------------+
Fetched 1 row(s) in 2.61s
[localhost:21000] > select sum(airtime) from airlines where year = 2005;
+--------------+
| sum(airtime) |
+--------------+
| 708204026    |
+--------------+
Fetched 1 row(s) in 1.19s
```

```
[localhost:21000] > select sum(airtime) from airlines_external where
dayofweek = 1;
+--------------+
| sum(airtime) |
+--------------+
| 1264945051   |
+--------------+
Fetched 1 row(s) in 2.82s
[localhost:21000] > select sum(airtime) from airlines where dayofweek = 1;
+--------------+
| sum(airtime) |
+--------------+
| 1264945051   |
+--------------+
Fetched 1 row(s) in 1.61s
```

Diving into Real Analysis

Now we can finally do some serious analysis with this data set that, remember, a few minutes ago all we had were some raw data files and we didn't even know what columns they contained. Let's see whether the "air time" of a flight tends to be different depending on the day of the week. We can see that the average is a little higher on day number 6; perhaps Saturday is a busy flying day and planes have to circle for longer at the destination airport before landing.

```
[localhost:21000] > select dayofweek, avg(airtime) from airlines
                  > group by dayofweek order by dayofweek;
+-----------+-------------------+
| dayofweek | avg(airtime)      |
+-----------+-------------------+
| 1         | 102.1560425016671 |
| 2         | 102.1582931538807 |
| 3         | 102.2170009256653 |
| 4         | 102.37477661846   |
| 5         | 102.2697358763511 |
| 6         | 105.3627448363705 |
| 7         | 103.4144351202054 |
+-----------+-------------------+
Fetched 7 row(s) in 2.25s
```

To see if the apparent trend holds up over time, let's do the same breakdown by day of week, but also split up by year. Now we can see that day number 6 consistently has a higher average air time in each year. We can also see that the average air time increased over time across the board. And the presence of `NULL` for this column in years 1987 to 1994 shows that queries involving this column need to be restricted to a date range of 1995 and higher.

```
[localhost:21000] > select year, dayofweek, avg(airtime) from airlines
                  > group by year, dayofweek order by year desc, dayofweek;
+------+-----------+-------------------+
| year | dayofweek | avg(airtime)      |
+------+-----------+-------------------+
| 2008 | 1         | 103.1821651651355 |
| 2008 | 2         | 103.2149301386094 |
| 2008 | 3         | 103.0585076622796 |
| 2008 | 4         | 103.4671383539038 |
| 2008 | 5         | 103.5575385182659 |
| 2008 | 6         | 107.4006306562128 |
| 2008 | 7         | 104.8648851041755 |
| 2007 | 1         | 102.2196114337825 |
| 2007 | 2         | 101.9317791906348 |
| 2007 | 3         | 102.0964767689043 |
| 2007 | 4         | 102.6215927201686 |
| 2007 | 5         | 102.4289399000661 |
| 2007 | 6         | 105.1477448215756 |
| 2007 | 7         | 103.6305945644095 |
...
| 1996 | 1         | 99.33860750862108 |
| 1996 | 2         | 99.54225446396656 |
| 1996 | 3         | 99.41129336113134 |
| 1996 | 4         | 99.5110373340348  |
| 1996 | 5         | 99.22120745027595 |
| 1996 | 6         | 101.1717447111921 |
| 1996 | 7         | 99.95410136133704 |
| 1995 | 1         | 96.93779698300494 |
| 1995 | 2         | 96.93458674589712 |
| 1995 | 3         | 97.00972311337051 |
| 1995 | 4         | 96.90843832024412 |
| 1995 | 5         | 96.78382115425562 |
| 1995 | 6         | 98.70872826057003 |
| 1995 | 7         | 97.85570478374616 |
| 1994 | 1         | NULL              |
| 1994 | 2         | NULL              |
| 1994 | 3         | NULL              |
...
| 1987 | 5         | NULL              |
| 1987 | 6         | NULL              |
| 1987 | 7          | NULL                |
+------+-----------+--------------------+
```

Conclusion

I hope this use case has provided a good example of how to prepare data originating from outside Impala (and even non-SQL data) for analysis. Crunch away!

John Russell is the technical writer of the Impala docs. He is also the author of the O'Reilly Media book, Getting Started with Impala.