Tuning Impala: The top five performance optimizations for the best BI and SQL analytics on Hadoop

The Leader for Analytic SQL on Hadoop
Cloudera Enterprise
Making Hadoop Fast, Easy, and Secure

A new kind of data platform:
• One place for unlimited data
• Unified, multi-framework analytics

Cloudera makes it:
• Fast for business
• Easy to manage
• Secure without compromise
One Platform, Many Workloads

Batch, Interactive, and Real-Time.
Leading performance and usability in one platform.

- End-to-end analytic workflows
- Access more data
- Work with data in new ways
- Enable new users
## Analytic SQL Requirements for Hadoop

**Interactive BI requires:**

<table>
<thead>
<tr>
<th>Multi-User Performance &amp; Usability</th>
<th>Meets user experience expectations at standard load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compatibility</td>
<td>Familiar BI tools/SQL interfaces</td>
</tr>
</tbody>
</table>

**Hadoop requires:**

<table>
<thead>
<tr>
<th>Flexibility</th>
<th>Use SQL to access any type of data, and access any type of data with more than just SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native Integration</td>
<td>Unified resource management, metadata, security, and management across frameworks</td>
</tr>
</tbody>
</table>
Choosing the Right SQL Engine
Know Your Audience, Know Your Use Case

- Hive
- OR
- Spark

Batch Processing
BI and SQL Analytics
Procedural Development
Apache Impala (incubating): Open Source & Open Standard

1. > 1 MM downloads since GA

2. Majority adoption across Cloudera customers

3. Certification across key application partners:
   - MicroStrategy
   - Qlik
   - SAS
   - Tableau
   - SAP
   - IBM
   - Cognos
   - Microsoft
   - Oracle
   - and others

4. De facto standard with multi-vendor support:
Impala Performance trend

- Track record in improving release to release performance
- 5.4x speedup in standard benchmarks over the last 14 months
- Continued to add functionality without introducing regressions
Competitive benchmarking : Definition

7x node cluster each with Hardware
- 256GB memory, 2x sockets, 40x total cores, Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz
- 8 disk drives at 2TB each

Workload
- TPC-DS 1TB & 10TB stored in
  - Parquet for Impala & Spark SQL
  - ORC for LLAP & Presto
  - Column oriented storage for Greenplum
- Ran 75 out of the 99 TPC-DS queries without any modifications, removed 23 queries that required language modification and 1 query that ran excessively slow on multiple engines
- Multi user test consisting of N concurrent streams each running 75 TPC-DS queries
- Queries in streams 1 through N use different parameters (No query is repeated)

Comparative Set
- Impala CDH 5.10
- Greenplum Database 4.3.9.1 (Traditional MPP analytical DB)
- Presto 0.160
- Spark SQL 2.1
- LLAP HDP 2.5
TPC-DS 1TB : Single user

- At 1TB scale with single user, all engines completed all 75 unmodified queries

- Impala and Greenplum are the fastest engines with Impala 2.5x faster than Greeplum on average

- Impala outperforms Presto by 8x, LLAP by 4x, Spark SQL by 2.8x
TPC-DS 1TB : Multi user

- Impala and Greenplum are fastest query engines with Impala ahead of Greenplum by 2.8x for 16 concurrent users
- Presto failed to complete the multi user workload
- Impala outperformed Spark SQL and LLAP by 6.5x for 4 concurrent users and by 20x for 16 concurrent users

<table>
<thead>
<tr>
<th></th>
<th>4 users</th>
<th>8 users</th>
<th>16 users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Impala</td>
<td>495</td>
<td>865</td>
<td>1,315</td>
</tr>
<tr>
<td>Greenplum</td>
<td>381</td>
<td>417</td>
<td>462</td>
</tr>
<tr>
<td>Spark SQL</td>
<td>76</td>
<td>70</td>
<td>61</td>
</tr>
<tr>
<td>LLAP</td>
<td>58</td>
<td>62</td>
<td>66</td>
</tr>
</tbody>
</table>
TPC-DS 10TB : Single user continued…

- At 10TB, scale only Greenplum and Impala are able to finish the unmodified queries within 12 hours per query.

- Impala outperforms Greenplum by 1.8x for Geometric Mean and 2.75x for total execution time when running all queries.
**Impala**  
The Leader in Analytic SQL for Hadoop

Impala delivers the best of both worlds

<table>
<thead>
<tr>
<th>Multi-User Performance &amp; Usability</th>
<th>✔️</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 10x vs alternatives with latest benchmarks</td>
<td></td>
</tr>
<tr>
<td>• Cost-based optimization allows for more users and tools to run a broader range of queries</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>✔️</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Provides both ANSI SQL and vendor-specific extensions</td>
<td></td>
</tr>
<tr>
<td>• Compatibility with the leading BI partners</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flexibility</th>
<th>✔️</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Supports the common native Hadoop file formats</td>
<td></td>
</tr>
<tr>
<td>• Parquet provides best-of-breed columnar performance across Hadoop frameworks</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Native Integration</th>
<th>✔️</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Unified with Hadoop resource management, metadata, security, and management</td>
<td></td>
</tr>
</tbody>
</table>
Performance Tuning: Agenda

Physical data modeling and schema design:
Choosing the best physical representation for your logical data model
- Partitioning
- Sorting
- Runtime filter and dynamic partition pruning
- Data types
- Nested types

Operational optimizations
- Computing statistics
- Admission control and memory management
Performance Tuning Basics: Partitioning

- What it is: physically dividing your data so that queries only need to access a subset
- Partition = minimum unit of work
- Partitioning expressed through DDL, applied automatically when queries contain matching predicates

CREATE TABLE Sales (...)
PARTITIONED BY (INT year,
INT month);

SELECT ...
FROM Sales
WHERE year >= 2012
AND month IN (1, 2, 3)

or

CREATE TABLE Sales (...)
PARTITIONED BY (INT date_key);

SELECT ...
FROM Sales JOIN DateDim d
USING date_key
WHERE d.year >= 2012
AND d.month IN (1, 2, 3)
Performance Tuning Basics: Partitioning

- Choose partition granularity carefully
- Too low:
  - small number of files can hurt parallelism
  - increases minimum unit of work
- Too high:
  - small data files hurt large queries; scans less efficient
  - large number of files can cause metadata bloat and create bottlenecks on HDFS NameNode, Hive Metastore, Impala catalog service
- General guidelines:
  - Regularly compact tables to keep the number of files per partition under control and improve scan and compression efficiency
  - Keep number of partitions under 20K (not a hard limit, mileage will vary)
Performance Tuning Basics: Sorting

- Sorting data files improves the effectiveness of file statistics (min/max) and compression (e.g., delta encoding).
- Sorting can be used on columns which have too many values to qualify for partitioning.
- Create sorted data files by adding the SORT BY clause during table creation.
- SORT BY will be available in Impala 2.9.
- The Parquet community is working on extending the format for efficient point lookups.

CREATE TABLE Sales (...)
PARTITIONED BY (year INT, month INT)
SORT BY (day, hour)
STORED AS Parquet;
Sorting: Augment Partitioning

Business question: *Find top 10 customers in terms of revenue who made purchases on Christmas eve in a given time window.*

SortBy helps meet query SLAs without over-partitioning the table

```
SELECT sum(ss_ext_sales_price) AS revenue, 
    count(ss_quantity) AS quantity_count, 
    customer_name
FROM Sales
WHERE Year = 2016 AND Month=12 AND Day=24 AND hour BETWEEN 01 AND 12
GROUP BY customer_name
ORDER BY revenue DESC LIMIT 10;
```

<table>
<thead>
<tr>
<th>Metric</th>
<th>Partition by Year, Month</th>
<th>Partition by Year, Month + SortBy Day, Hour</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed time (seconds)</td>
<td>1.2</td>
<td>0.4</td>
<td>3x</td>
</tr>
<tr>
<td>CPU time (seconds)</td>
<td>5</td>
<td>1</td>
<td>5x</td>
</tr>
<tr>
<td>HDFS MBytes read</td>
<td>619</td>
<td>37</td>
<td>17x</td>
</tr>
</tbody>
</table>

CREATE TABLE Sales (...)
PARTITIONED BY (year INT, month INT)
SORT BY (day, hour)
Stored as Parquet;
Sorting: Complement Partitioning

Business question: *Find interactions for a specific customer over for given time window*

SortBy helps meet query SLAs without over partitioning the table

```sql
SELECT * 
FROM Sales 
WHERE Year = 2016 AND customer_id = 4976004;
```

<table>
<thead>
<tr>
<th>Metric</th>
<th>Partition by Year, Month</th>
<th>Partition by Year, Month + SortBy Day, Hour</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed time (seconds)</td>
<td>36</td>
<td>2</td>
<td>18x</td>
</tr>
<tr>
<td>HDFS MBytes read</td>
<td>4093</td>
<td>183</td>
<td>22x</td>
</tr>
</tbody>
</table>

CREATE TABLE Sales (...) PARTITIONED BY (year INT, month INT) 
SORT BY (customer_id) 
Stored as Parquet;
Runtime Filters and Dynamic Partition Pruning

Business question: How much was sold in June

CREATE TABLE store_sales (...)
PARTITIONED BY (INT ss_sold_date_sk);

SELECT d_year,
   ,sum(ss_ext_sales_price) sum_agg
FROM DATE_DIM
   ,STORE_SALES
WHERE
   DATE_DIM.d_date_sk = STORE_SALES.ss_sold_date_sk
   AND d_moy = 6
GROUP BY d_year
Business question: *How much was sold in June*

```sql
SELECT d_year,
       sum(ss_ext_sales_price) sum_agg
FROM DATE_DIM,
     STORE_SALES
WHERE DATE_DIM.d_date_sk = STORE_SALES.ss_sold_date_sk
  AND d_moy = 6
GROUP BY d_year
```

The *planner* doesn’t know what the set of `d_date_sk` and `ss_sold_date_sk` contains - even with statistics.

But there’s clearly an opportunity to save some work - why bother sending 28 billion of those rows to the joins?

*Runtime filters* compute this predicate at runtime.
Runtime Filters and Dynamic Partition Pruning

SELECT d_year,
      ,sum(ss_ext_sales_price) sum_agg
FROM DATE_DIM,
      ,STORE_SALES
WHERE
DATE_DIM.d_date_sk = STORE_SALES.ss_sold_date_sk
AND d_moy = 6
GROUP BY d_year

Bloom filter: compact, probabilistic representation of a data set
Essentially a sophisticated bitmap

Step 1: planner tells Join #1 to produce Bloom filter for qualifying distinct values of d_date_sk

Aggregate
1.3 Billion rows
Broadcast
Join #1

28 billion rows
6,087 rows

STORE_SALES
DATE_DIM
Runtime Filters and Dynamic Partition Pruning

SELECT d_year, 
  sum(ss_ext_sales_price) sum_agg
FROM DATE_DIM, 
  STORE_SALES
WHERE
DATE_DIM.d_date_sk = STORE_SALES.ss_sold_date_sk
AND d_moy = 6
GROUP BY d_year

Step 2: Join reads all rows from build side (right input), and populates Bloom filter containing all distinct values of d_date_sk

Aggregate
1.3 Billion rows
Broadcast Join #1
28 billion rows
6,087 rows
STORE_SALES
DATE_DIM
Runtime Filters and Dynamic Partition Pruning

```
SELECT d_year,
       sum(ss_ext_sales_price) sum_agg
FROM DATE_DIM, STORE_SALES
WHERE DATE_DIM.d_date_sk = STORE_SALES.ss_sold_date_sk
     AND d_moy = 6
GROUP BY d_year
```

Step 3: Query coordinator sends filter to store_sales scan before the scan starts.

- Aggregate
  - 1.3 Billion rows
    - Broadcast Join #1
      - 28 billion rows
        - STORE_SALES
      - 6,087 rows
        - DATE_DIM
Runtime Filters and Dynamic Partition Pruning

SELECT d_year,
    sum(ss_ext_sales_price) sum_agg
FROM DATE_DIM,
    STORE_SALES
WHERE DATE_DIM.d_date_sk = STORE_SALES.ss_sold_date_sk
    AND d_moy = 6
GROUP BY d_year

Step 4: Scan eliminates all partitions that don’t have a match in the Bloom filter. Only 150 out of the 1824 partitions are read from disk.
SELECT d_year,
    sum(ss_ext_sales_price) sum_agg
FROM DATE_DIM,
    STORE_SALES
WHERE
    DATE_DIM.d_date_sk = STORE_SALES.ss_sold_date_sk
    AND d_moy = 6
GROUP BY d_year

Step 5: Rows coming out of the scan is reduced from 28 Billion to 1.3 Billion
Performance Tuning Basics: Data Type Selection

Data type selection affects performance:

- computation: numerical types allow direct computation, string types require conversion
- on-disk storage size: numerical types are more compact
- more compact types also require less network traffic
- runtime code generation: some types are not supported (CHAR, TIMESTAMP, TINYINT)

General guidelines:

- choose numerical types over character types for numerical data
- use smallest data type that will accommodate the largest possible value
Performance Tuning Basics: Data Type Selection

Picking the incorrect data type can result in:
- Increase in on-disk storage by 40%
- 80% slower scans
- 80% slower aggregations
- 150% slower joins
- Increase in runtime memory utilization

Data set: L_ORDERKEY column from TPCH 3TB
Values domain: 1-18,000,000,000
Number of distinct values: 4,500,000,000
Performance Tuning Basics: Complex Schemas

- Complex/nested-relational schemas are the most natural way to model most data sources.
- Nested schemas also present an opportunity for performance improvements:
  - turning parent-child hierarchies into nested collections
    - logical hierarchy becomes physical hierarchy
    - nested structure = join index
    - physical clustering of child with parent
- For distributed, big data systems, this matter:
  - distributed join turns into local join
Example: TPC-H, Flat and Nested
Columnar Storage for Complex Schemas

- Columnar storage: a necessity for processing nested data at speed
  - complex schema = really wide tables
  - row wise storage: ends up reading lots of data you don’t care about
- Columnar formats effectively store a join index
A “join” between parent and child is essentially free:
- coordinated scan of parent and child columns
- data effectively presorted in parent’s PK
- merge join beats hash join!

```
SELECT c.id, o.price
FROM customers c, c.orders o
```
Query Execution with Complex Schemas

- Aggregating child data is cheaper
  - data already pre-grouped by the parent
  - amenable to vectorized execution
  - local non-grouping aggregation <<< distributed grouping aggregation

```
SELECT c.id, MAX(o.price)
FROM customers c
JOIN orders o
ON (c.id = o.cid)
```

```
SELECT c.id, MAX(o.price)
FROM customers c,
     (SELECT MAX(price)
      FROM c.orders)
```
Performance Tuning Basics: Nested Queries

Example: find the 10 customers with the highest average per-item price

Flat

```
SELECT c.id, AVG(i.price)
FROM customer c, order o, item i
WHERE c.id = o.cid and o.id = i.oid
GROUP BY c.id
ORDER BY avg_price DESC LIMIT 10
```

Nested

```
SELECT c.id, AVG(orders.items.price)
FROM customer
ORDER BY avg_price DESC LIMIT 10
```
Performance Tuning Basics: Nested Queries
Performance Tuning Basics: Statistics

- Order scan predicates by selectivity and cost
- Compute selectivity of predicates for scans as well as joins
- Determine build and probe side for equi joins
- Select the ideal join type that minimizes resource utilization
  - Broadcast Join
  - Partition Join
- Identify joins which can benefit from Runtime filters
- Detection of common join pattern of Primary key/Foreign key joins
- Determine optimal join order

```
02: HASH JOIN [INNER JOIN, BROADCAST]
| hash predicates: l_orderkey = o_orderkey
| runtime filters: RF000 <= o_orderkey
| tuple-ids=1,0 row-size=113B cardinality=27,381,196

-05: EXCHANGE [BROADCAST]
  | hosts=20 per-host-mem=0B
  | tuple-ids=0 row-size=8B cardinality=68,452,805

00: SCAN HDFS [tpch_3000_parquet.orders, RANDOM]
  | partitions=366/2406 files=366 size=28.83GB
  | predicates: tpch_3000_parquet.orders.o_orderkey < 100
  | table stats: 4,500,000,000 rows total
  | column stats: all
  | tuple-ids=0 row-size=8B cardinality=68,452,805

01: SCAN HDFS [tpch_3000_parquet.lineitem, RANDOM]
  | partitions=2526/2526 files=2526 size=1.36TB
  | predicates: l_orderkey < 100, l_receiptdate >= '1994-01-01', l_comment LIKE '%long string%'
  | runtime filters: RF000 -> l_orderkey
  | table stats: 18,000,048,306 rows total
  | column stats: all
  | tuple-ids=1 row-size=105B cardinality=1,800,004,831
```
Order scan predicates by selectivity and cost

Compute selectivity of predicates for scans as well as joins

Determine build and probe side for equi joins

Select the ideal join type that minimizes resource utilization
  ○ Broadcast Join
  ○ Partition Join

Identify joins which can benefit from Runtime filters

Detection of common join pattern of Primary key/Foreign key joins

Determine optimal join order

02: HASH JOIN [INNER JOIN, BROADCAST]
  hash predicates: l_orderkey = o_orderkey
  runtime filters: RF000 <- o_orderkey
tuple-ids=1,0 row-size=113B cardinality=27,381,196

--05: EXCHANGE [BROADCAST]
  hosts=20 per-host-mem=0B
tuple-ids=0 row-size=8B cardinality=68,452,805

00: SCAN HDFS [tpch_3000_parquet.orders, RANDOM]
  partitions=366/2406 files=366 size=28.83GB
  predicates: tpch_3000_parquet.orders.o_orderkey < 100
table stats: 4,500,000,000 rows total
column stats: all
tuple-ids=0 row-size=8B cardinality=68,452,805

01: SCAN HDFS [tpch_3000_parquet.lineitem, RANDOM]
  partitions=2526/2526 files=2526 size=1.36TB
  predicates: l_orderkey < 100, l_receiptdate >= '1994-01-01', l_comment LIKE '%long string%'
  runtime filters: RF000 -> l_orderkey
table stats: 18,000,048,306 rows total
column stats: all
tuple-ids=1 row-size=105B cardinality=1,800,004,831
Performance Tuning Basics: Statistics

- Order scan predicates by selectivity and cost
- Compute selectivity of predicates for scans as well as joins
- Determine build and probe side for equi joins
- Select the ideal join type that minimizes resource utilization
  - Broadcast Join
  - Partition Join
- Identify joins which can benefit from Runtime filters
- Detection of common join pattern of Primary key/Foreign key joins
- Determine optimal join order

02: HASH JOIN [INNER JOIN, BROADCAST]
- hash predicates: l_orderkey = o_orderkey
- runtime filters: RF000 <- o_orderkey
tuple-ids=1,0 row-size=113B cardinality=27,381,196

05: EXCHANGE [BROADCAST]
- hosts=20 per-host-mem=0B
tuple-ids=0 row-size=8B cardinality=68,452,805

00: SCAN HDFS [tpch_3000_parquet.orders, RANDOM]
- partitions=366/2406 files=366 size=28.83GB
- predicates: tpch_3000_parquet.orders.o_orderkey < 100
table stats: 4,500,000,000 rows total
  column stats: all
tuple-ids=0 row-size=8B cardinality=68,452,805

01: SCAN HDFS [tpch_3000_parquet.lineitem, RANDOM]
- partitions=2526/2526 files=2526 size=1.36TB
- predicates: l_orderkey < 100, l_receiptdate >= '1994-01-01', l_comment LIKE '%long string%'
- runtime filters: RF000 -> l_orderkey
table stats: 18,000,048,306 rows total
column stats: all
tuple-ids=1 row-size=105B cardinality=1,800,004,831
Performance Tuning Basics: Statistics

- Order scan predicates by selectivity and cost
- Compute selectivity of predicates for scans as well as joins
- Determine build and probe side for equi joins
- Select the ideal join type that minimizes resource utilization
  - Broadcast Join
  - Partition Join
- Identify joins which can benefit from Runtime filters
- Detection of common join pattern of Primary key/Foreign key joins
- Determine optimal join order

02: HASH JOIN [INNER JOIN, BROADCAST]
  - hash predicates: l_orderkey = o_orderkey
  - runtime filters: RF000 < o_orderkey
  - tuple-ids=1,0 row-size=113B cardinality=27,381,196

05: EXCHANGE [BROADCAST]
  - hosts=20 per-host-mem=0B
  - tuple-ids=0 row-size=8B cardinality=68,452,805

00: SCAN HDFS [tpch_3000_parquet.orders, RANDOM]
  - partitions=366/2406 files=366 size=28.83GB
  - predicates: tpch_3000_parquet.orders.o_orderkey < 100
  - table stats: 4,500,000,000 rows total
  - column stats: all
  - tuple-ids=0 row-size=8B cardinality=68,452,805

01: SCAN HDFS [tpch_3000_parquet.lineitem, RANDOM]
  - partitions=2526/2526 files=2526 size=1.36TB
  - predicates: l_orderkey < 100, l_receiptdate >= '1994-01-01', l_comment LIKE '%long string%'
  - runtime filters: RF000 -> l_orderkey
  - table stats: 18,000,048,306 rows total
  - column stats: all
  - tuple-ids=1 row-size=105B cardinality=1,800,044,831
Performance Tuning Basics: Statistics

- Order scan predicates by selectivity and cost
- Compute selectivity of predicates for scans as well as joins
- Determine build and probe side for equi joins
- Select the ideal join type that minimizes resource utilization
  - Broadcast Join
  - Partition Join
- Identify joins which can benefit from Runtime filters
- Detection of common join pattern of Primary key/Foreign key joins
- Determine optimal join order

02: HASH JOIN [INNER JOIN, BROADCAST]
| hash predicates: l_orderkey = o_orderkey |
| runtime filters: RF000 <- o_orderkey |
| tuple-ids=1,0 row-size=113B cardinality=27,381,196 |

05: EXCHANGE [BROADCAST]
| hosts=20 per-host-mem=0B |
| tuple-ids=0 row-size=8B cardinality=68,452,805 |

00: SCAN HDFS [tpch_3000_parquet.orders, RANDOM]
| partitions=366/2406 files=366 size=28.83GB |
| predicates: tpch_3000_parquet.orders.o_orderkey < 100 |
| table stats: 4,500,000,000 rows total |
| column stats: all |
| tuple-ids=0 row-size=8B cardinality=68,452,805 |

01: SCAN HDFS [tpch_3000_parquet.lineitem, RANDOM]
| partitions=2526/2526 files=2526 size=1.36TB |
| predicates: l_orderkey < 100, l_receiptdate >= '1994-01-01', l_comment LIKE '%long string%' |
| runtime filters: RF000 -> l_orderkey |
| table stats: 18,000,048,306 rows total |
| column stats: all |
| tuple-ids=1 row-size=105B cardinality=1,800,004,831 |
Performance Tuning Basics: Summary

Pick data types to match schema semantics as closely as possible.

Pick data partitioning to match workload characteristics as closely as possible.

Write queries to match the partitioning.

Express 1-n relationships as nested tables. Use nested aggregates over nested data.

Compute statistics. No, really.
Get Started

• 100% Apache-licensed open source
  • Download: cloudera.com/downloads
  • Project Page: http://impala.apache.org/
  • Join the discussion: user@impala.incubator.apache.org

• Questions/comments?
  • Resources: http://blog.cloudera.com/blog/category/impala/
  • Community: http://impala.apache.org/community.html
  • Email: user@impala.incubator.apache.org.