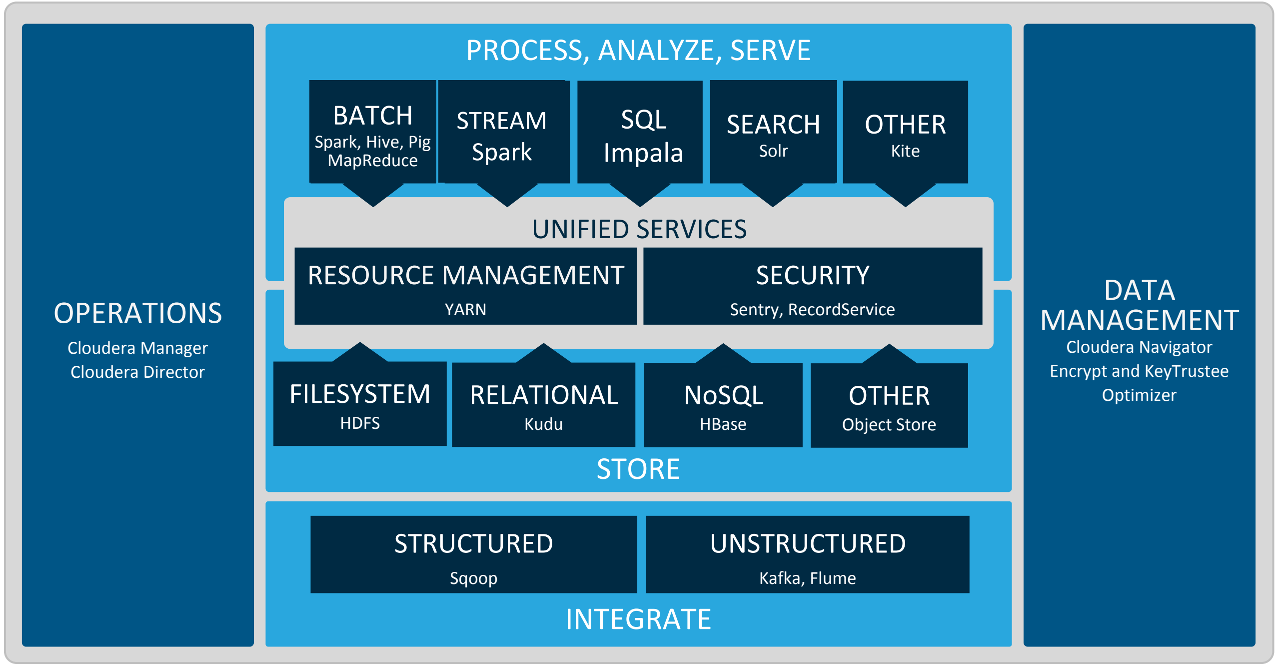
cloudera®

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



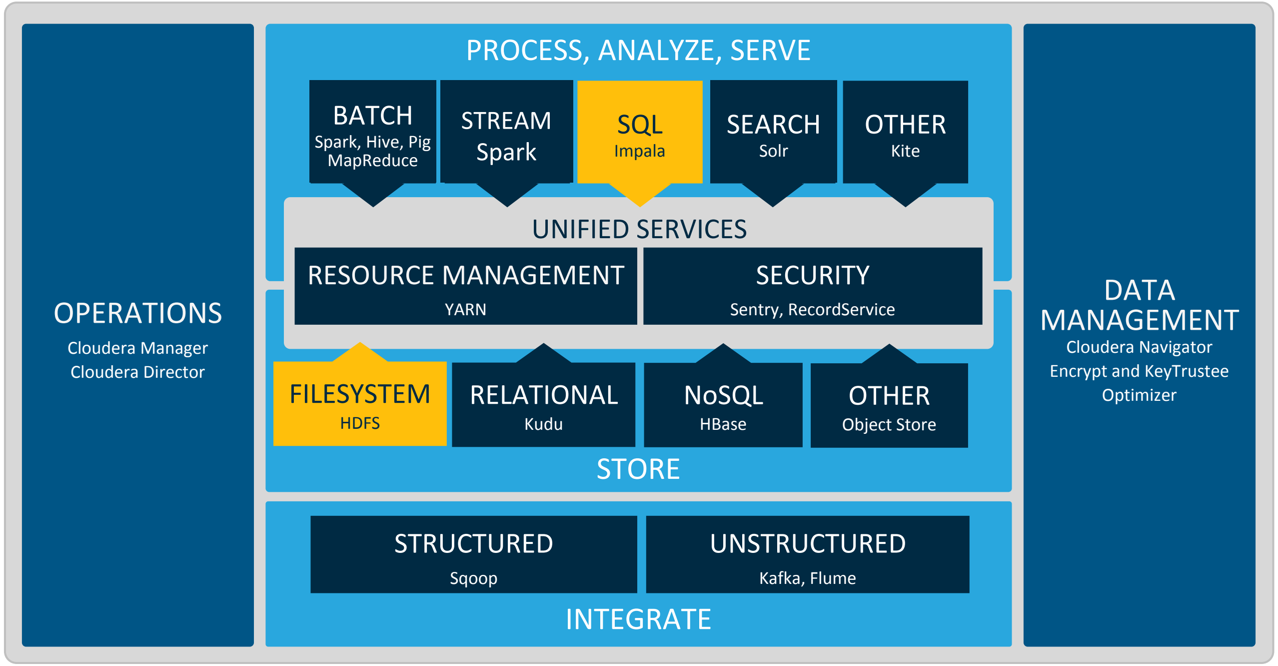
A new kind of data platform:

- 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:

Multi-User Performance & Usability	Meets user experience expectations at standard load
Compatibility	Familiar BI tools/SQL interfaces

Hadoop requires:

Flexibility	Use SQL to access any type of data, and access any type of data with more than just SQL
Native Integration	Unified resource management, metadata, security, and management across frameworks

cloudera

Choosing the Right SQL Engine

Know Your Audience, Know Your Use Case

HIVE

MapReduce

OR

Spark

Batch Processing

Impala

BI and SQL Analytics

Spark SQL

Procedural Development

Apache Impala (incubating): Open Source & Open Standard

1 > 1 MM downloads since GA

2 Majority adoption across Cloudera customers

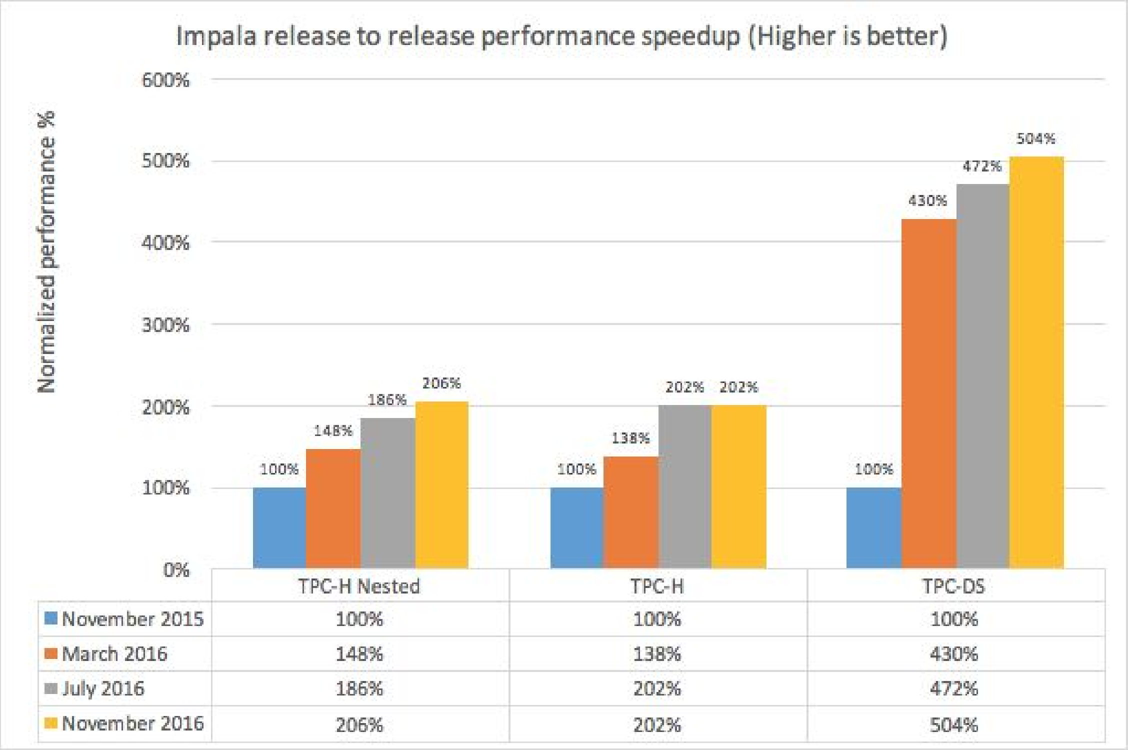
3 Certification across key application partners:

and others

4 De facto standard with multi-vendor support:

Impala Performance trend

- Track record in improving release to release performance
- 5x speedup in TPC-DS over the last 12 months
- Continued to add new features without introducing regressions



SQL on Hadoop benchmark : Definition

20x node cluster each with Hardware

- 384GB memory, 2x sockets, 12x total cores, Intel Xeon CPU E5-2630L 0 at 2.00GHz
- 24 disk drives at 932GB each

Workload

- TPC-DS 3TB stored in Parquet file format (default of 256MB block size)
- Ran 66 out of the 99 TPC-DS queries without any modifications
- Multi user test consisting of 8x concurrent streams
- Queries in streams 1 through 8 use different parameters (No query is ever repeated)
- Each stream executes queries in a randomized order

Comparative Set

- Impala 2.7
- Spark SQL 2.0
- Presto 0.148-t.1.2

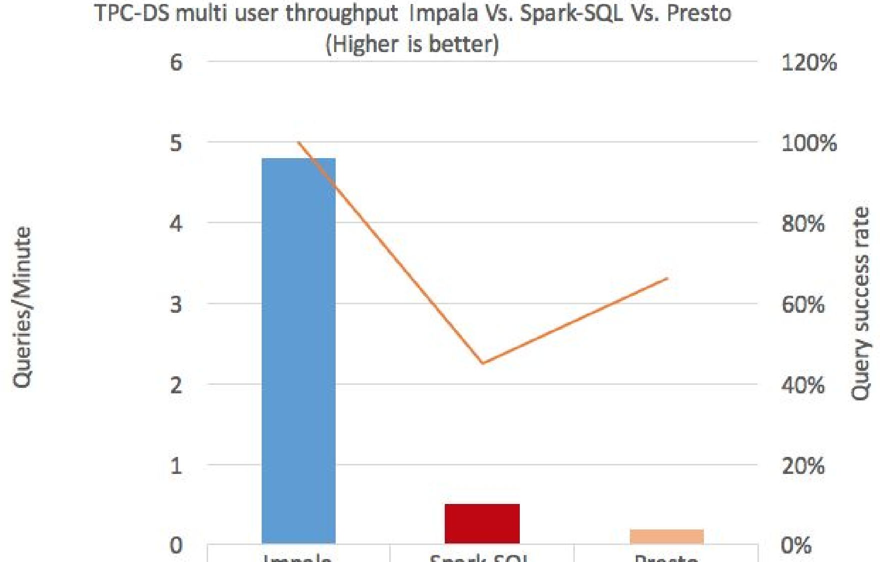
Engines were configured to use all available cores and 256GB of memory per node

Presto used a dedicated coordinator node

cloudera

SQL on Hadoop benchmark : Results

- Impala outperforms Spark-SQL 2.0 by 9x & Presto by 23x
- 45% of the queries succeed with Spark-SQL, the remaining queries error out
- Only 66% of the queries succeed with Presto, the remaining queries error out



	Impala	Spark-SQL	Presto
Queries/Minute	4.8	0.5	0.2
Success %	100%	45%	66%

cloudera

Impala
The Leader in Analytic SQL for Hadoop

Impala delivers the best of both worlds

Multi-User Performance & Usability	✔	• 10x vs alternatives with latest benchmarks • Cost-based optimization allows for more users and tools to run a broader range of queries
Compatibility	✔	• Provides both ANSI SQL and vendor-specific extensions • Compatibility with the leading BI partners
Flexibility	✔	• Supports the common native Hadoop file formats • Parquet provides best-of-breed columnar performance across Hadoop frameworks
Native Integration	✔	• Unified with Hadoop resource management, metadata, security, and management

cloudera

Impala
The Leader in Analytic SQL for Hadoop

InfoWorld Technology of the Year

Impala delivers the best of both worlds

Multi-User Performance & Usability	✔	• 10x vs alternatives with latest benchmarks • Cost-based optimization all run a broader range of quer
Compatibility	✔	• Provides both ANSI SQL and • Compatibility with the lead
Flexibility	✔	• Supports the common native Hadoop file formats • Parquet provides best-of-breed columnar performance across Hadoop frameworks
Native Integration	✔	• Unified with Hadoop resource management, metadata, security, and management

"If you want to do BI on Hadoop, Impala is the most accessible, and easily implementable, of the performant solutions."

cloudera

Performance Tuning: Agenda

Physical data modeling and schema design:
Choosing the best physical representation for your logical data model

- Partitioning
- 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)

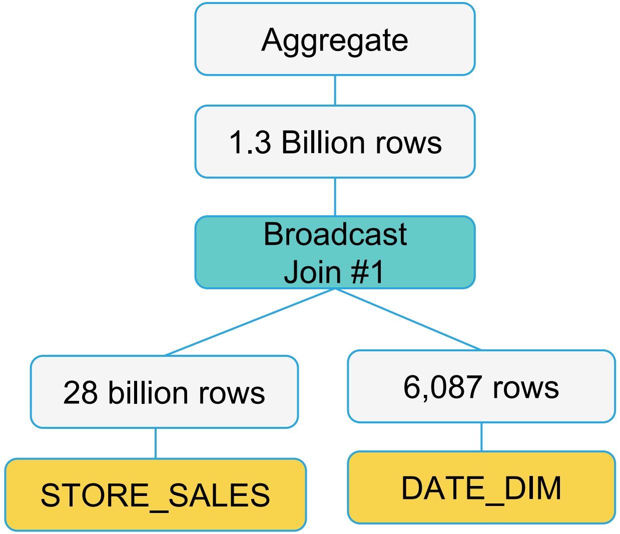
cloudera

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
```



cloudera

Runtime Filters and Dynamic Partition Pruning

Business question: *How much was sold in June*

```
SELECT d_year
      ,sum(ss_ext_sales_price) sum
FROM DATE_DIM
      ,STORE_SALES
WHERE
DATE_DIM.d_date_sk = STORE_SAL
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.

s

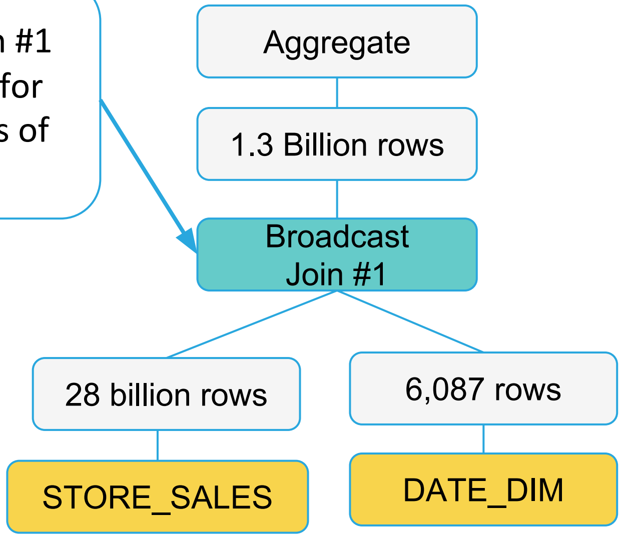
M

cloudera

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



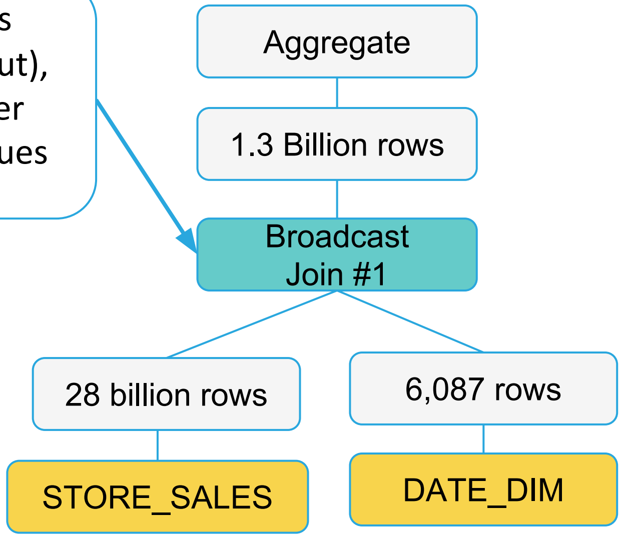
Bloom filter: compact, probabilistic representation of a data set

Essentially a sophisticated bitmap

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

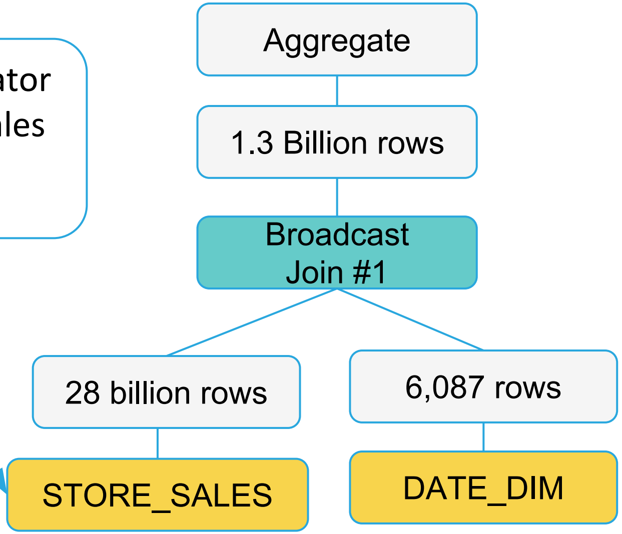


cloudera

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.



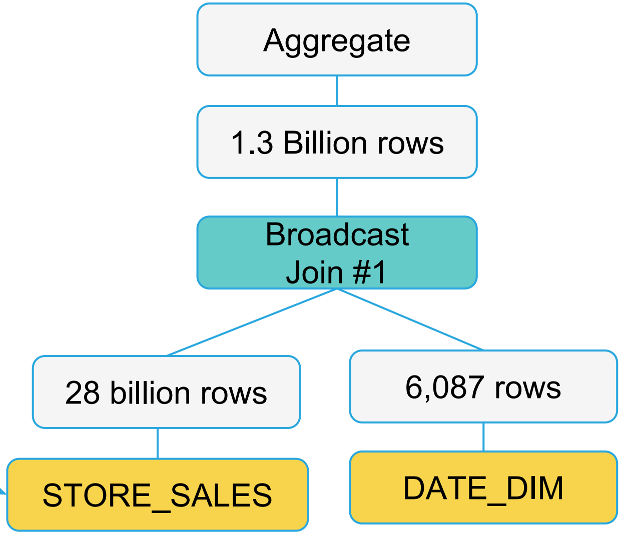
cloudera

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.

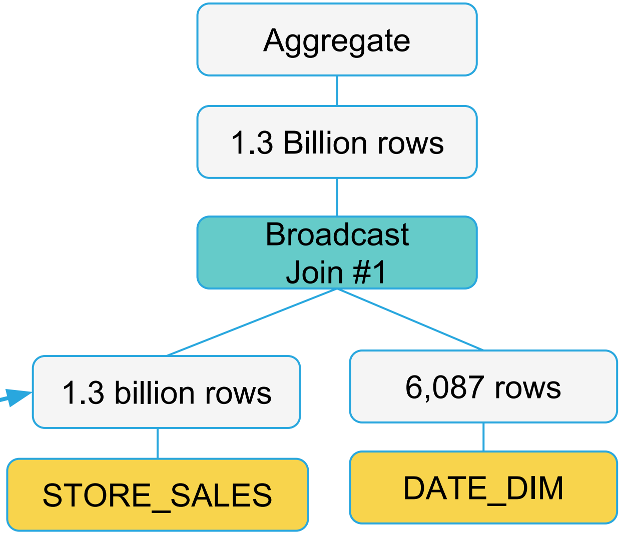
Only 150 out of the 1824 partitions are read from disk



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 5: Rows coming out of the scan is reduced from 28 Billion to 1.3 Billion



cloudera

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

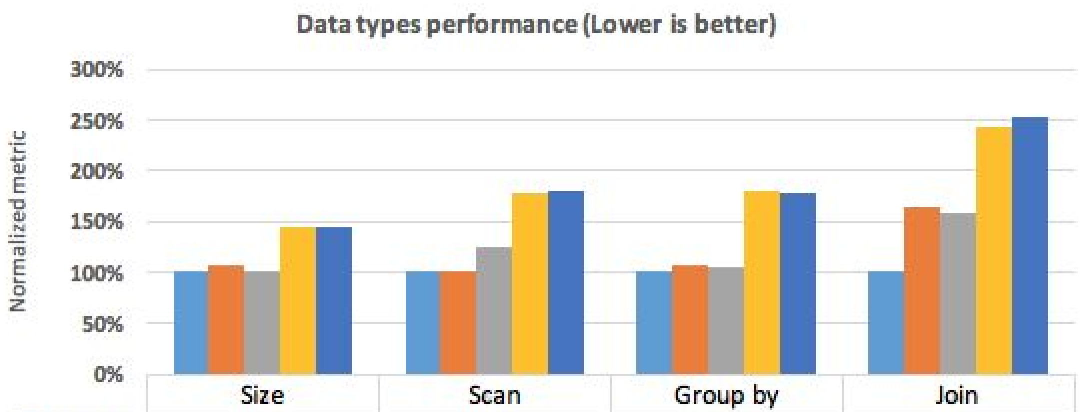
cloudera

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



	Size	Scan	Group by	Join
Bigint	100%	100%	100%	100%
Double	107%	100%	106%	164%
Decimal	100%	124%	104%	159%
String	144%	178%	179%	242%
Varchar	144%	180%	177%	252%

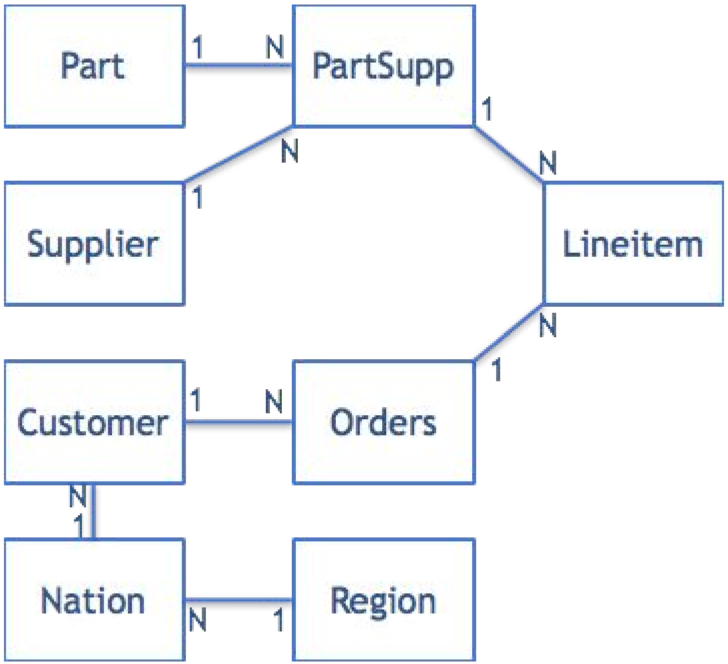
cloudera

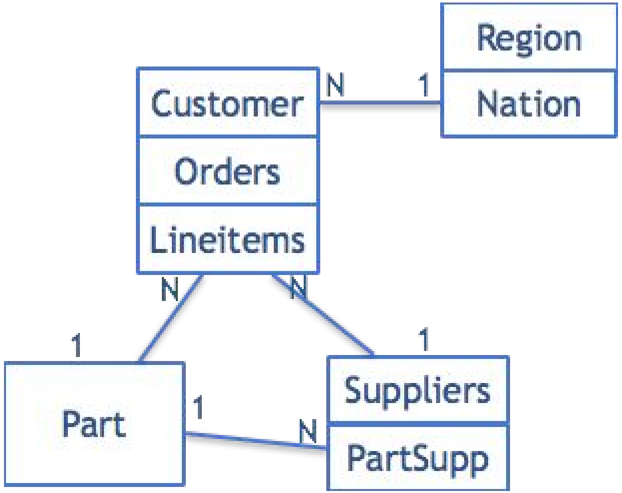
Performance Tuning Basics: Complex Schemas

- 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

cloudera

Example: TPC-H, Flat and Nested





cloudera

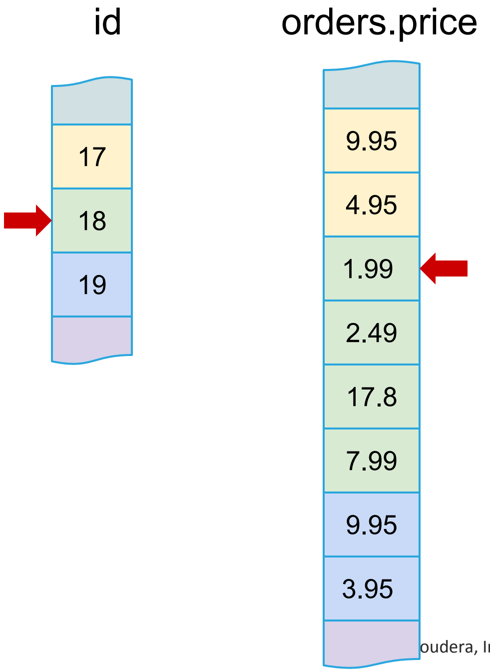
Columnar Storage for Complex Schemas

- 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

Query Execution with Complex Schemas

- 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
```



cloudera

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)
```

17	9.95
19	9.95
18	7.99
18	1.99
19	3.95
17	4.95

```
SELECT c.id, max_price
FROM customers c,
  (SELECT MAX(price)
    FROM c.orders)
```

17	9.95
17	4.95
18	1.99
18	2.49
18	17.8
18	7.99
19	9.95

cloudera

Performance Tuning Basics: Nested Queries

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

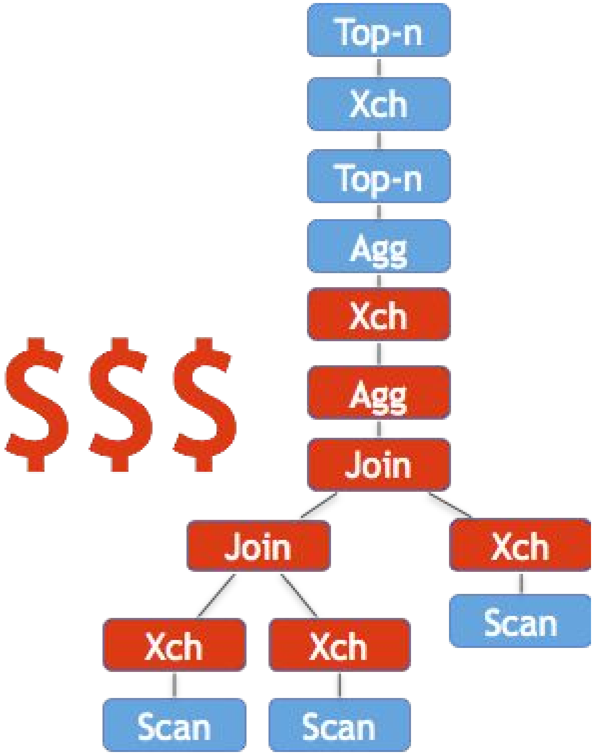
Flat

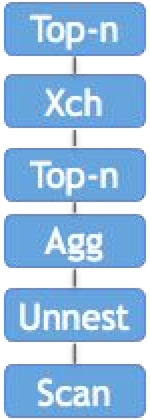
```
SELECT c.id,  AVG(i.price)
FROM customer c, order o, item i
WHERE c.id = o.cid and o.id =
i.oid
GROUP BY c.id
ORDER BY avg_price DESC LIMIT 10
```

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

```
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
```

cloudera

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

```
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

```
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
```

cloudera

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

```
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

```
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
```


Admission Control: Basics

Purpose

- Create a predictable multi-user environment by avoiding resource usage spikes and out-of-memory conditions

Approach

- Impose limits on concurrent SQL queries and memory usage
- Each incoming query is assigned to a resource pool
- The parameters of the resource pool, and current state of the system, determine whether a query gets to run, gets queued, or is rejected
- Pool parameters: max total memory, max running queries, max queued queries, queue timeout, ...

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.

Utilize admission control features to shape your workload.

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.

cloudera