cloudera

Performance optimizations in Apache Impala

Mostafa Mokhtar (mmokhtar@cloudera.com)
@mostafamokhtar7

Outline

- History and motivation
- Impala architecture
- Performance focused overview on
  - Front-end
    - Query planning overview
    - Query optimization
    - Metadata and statistics
  - Back-end
    - Partitioning and sorting for Selective scans
    - Code-generation using LLVM
    - Streaming Aggregation
    - Runtime filters
    - Handling cache misses for Joins and Aggs

cloudera

“Big data” revolution

SQL on Apache Hadoop

- SQL
- Run on top of HDFS
- Supported various file formats
- Converted query operators to map-reduce jobs
- Run at scale
- Fault-tolerant
- High startup-costs/materialization overhead (slow...)

cloudera

Impala: An open source SQL engine for Apache Hadoop

- Fast
  - C++ instead of Java
  - Run time code generation
  - Support interactive BI and analytical workloads
  - Supports queries that take from milliseconds to hours
- Scalable
  - Run directly on “big” hadoop clusters (100+ nodes)
- Flexible
  - Support multiple storage engines (HDFS, S3, ADLS, Apache Kudu, ...)
  - Support multiple file formats (Parquet, Text, Sequence, Avro, ..)
  - Support both structured and semi-structured data
- Enterprise-grade
  - Authorization/authentication/lineage/audit

Impala’s history

- First commit in May 2011
- Public beta in October 2012
  - Over a million downloads since then
  - Most recent released version is 2.10, associated with CDH 5.13
- November 2017
  - Apache® Impala™ has graduated from the Apache Incubator to become a Top-Level Project (TLP), signifying that the project's community and products have been well-governed under the ASF's meritocratic process and principles.

SQL on “big data”. The race is on..

HAWQ

Amazon Redshift

Spark SQL

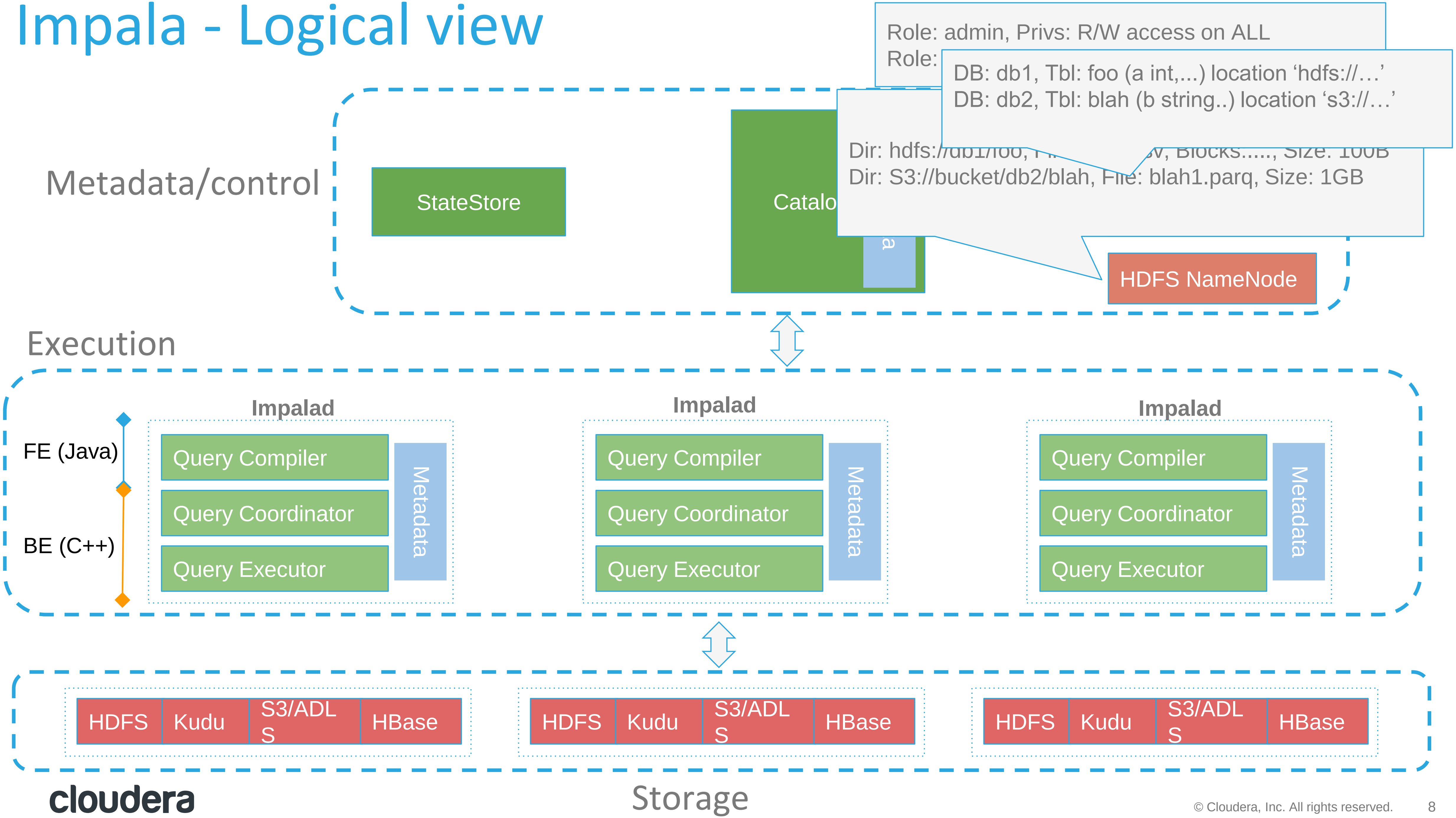
Hive on Tez

presto

APACHE DRILL

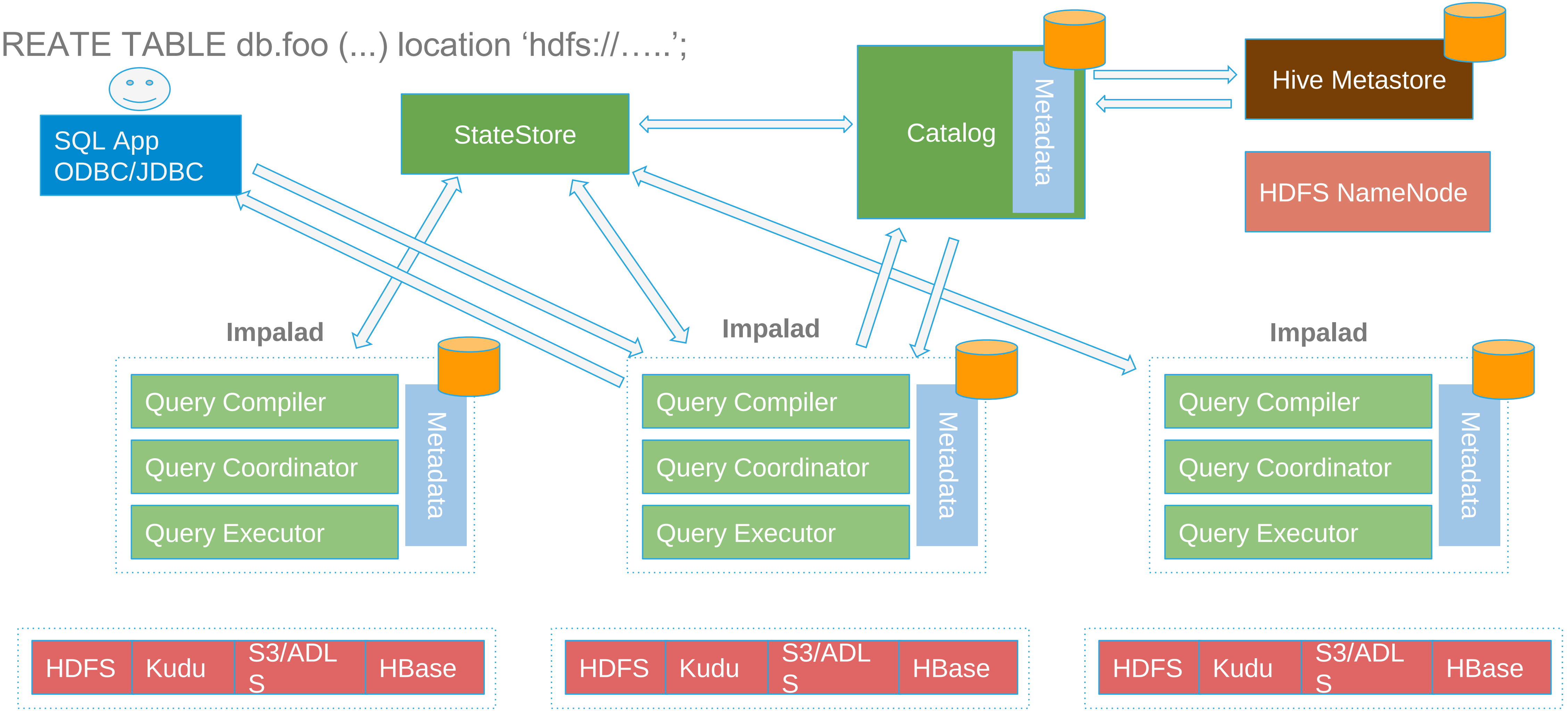
Hive on Tez + LLAP

Impala - Logical view



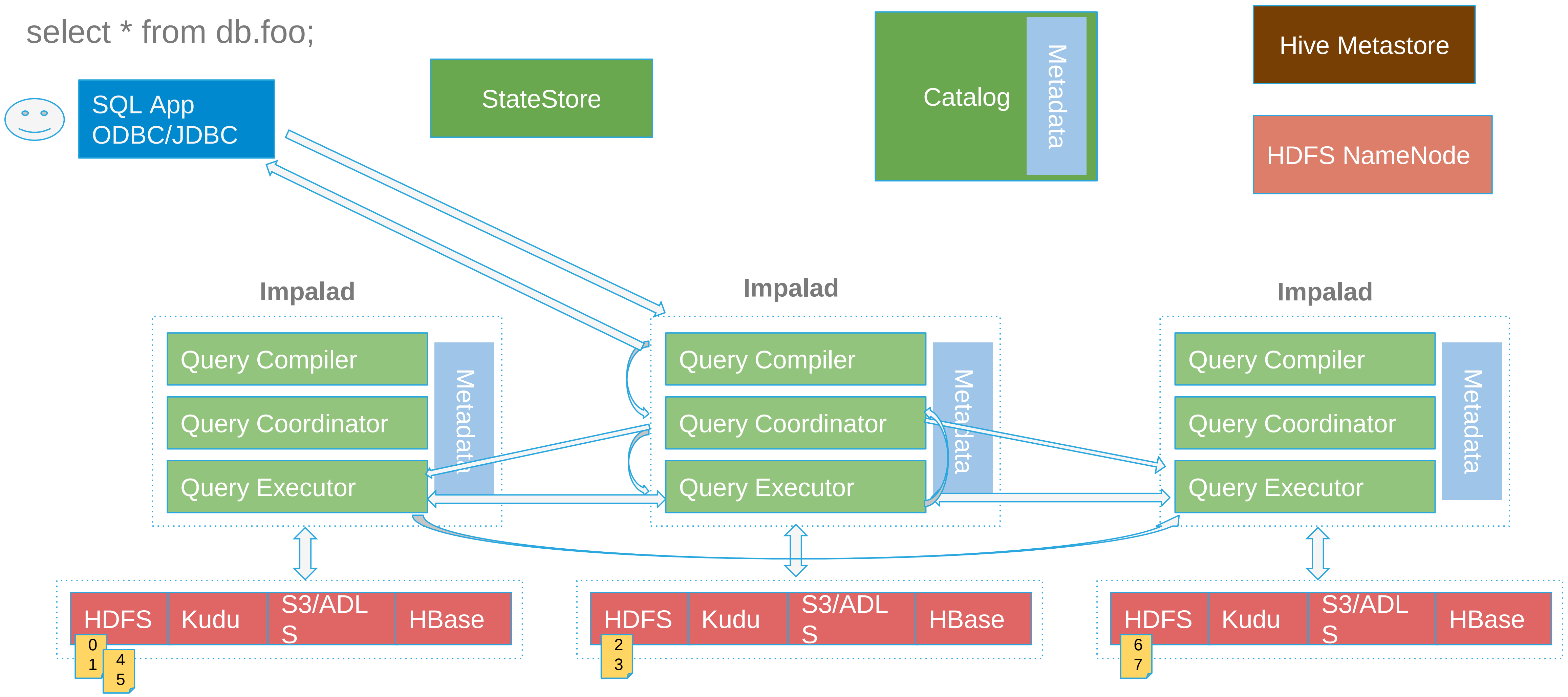
Impala in action - DDL

CREATE TABLE db.foo (...) location ‘hdfs://…..’;



cloudera

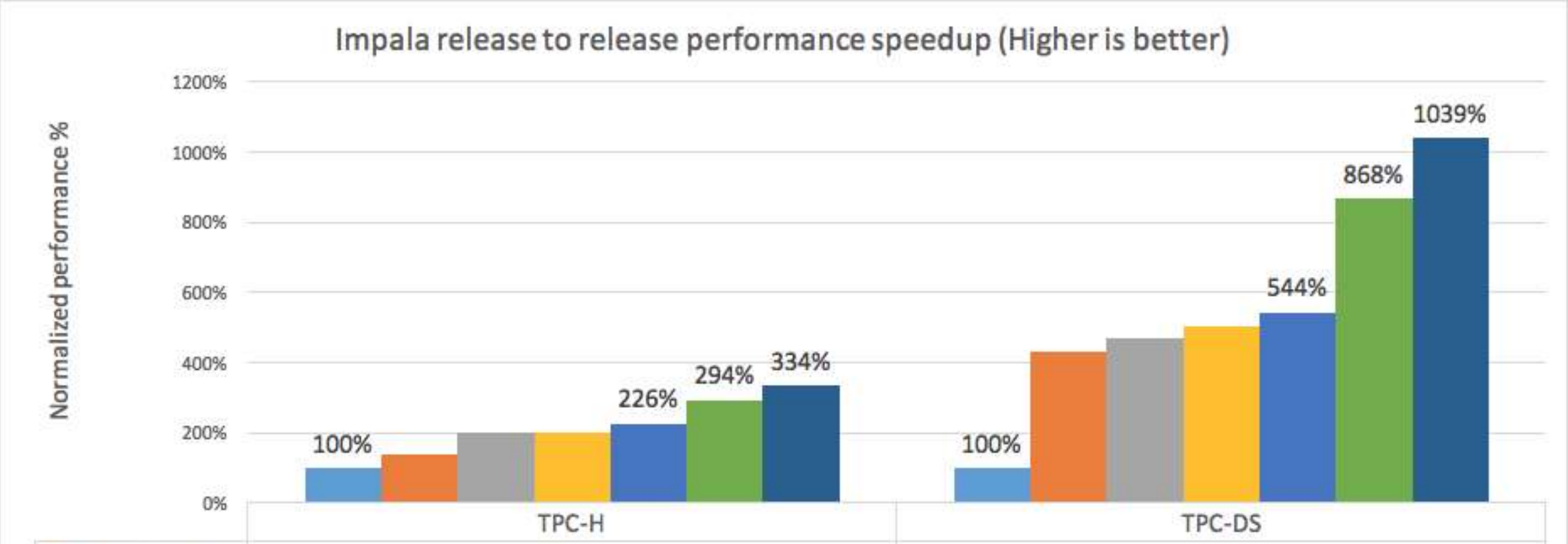
Impala in action - Select query



cloudera

Impala release to release performance trend

- Track record in improving release to release performance
- 10x speedup in standard benchmarks over the last 24 months
- Continued to add functionality without introducing regressions



	TPC-H	TPC-DS
November 2015	100%	100%
February 2016	138%	430%
July 2016	202%	472%
November 2016	202%	504%
February 2017	226%	544%
July 2017	294%	868%
November 2017	334%	1039%

cloudera

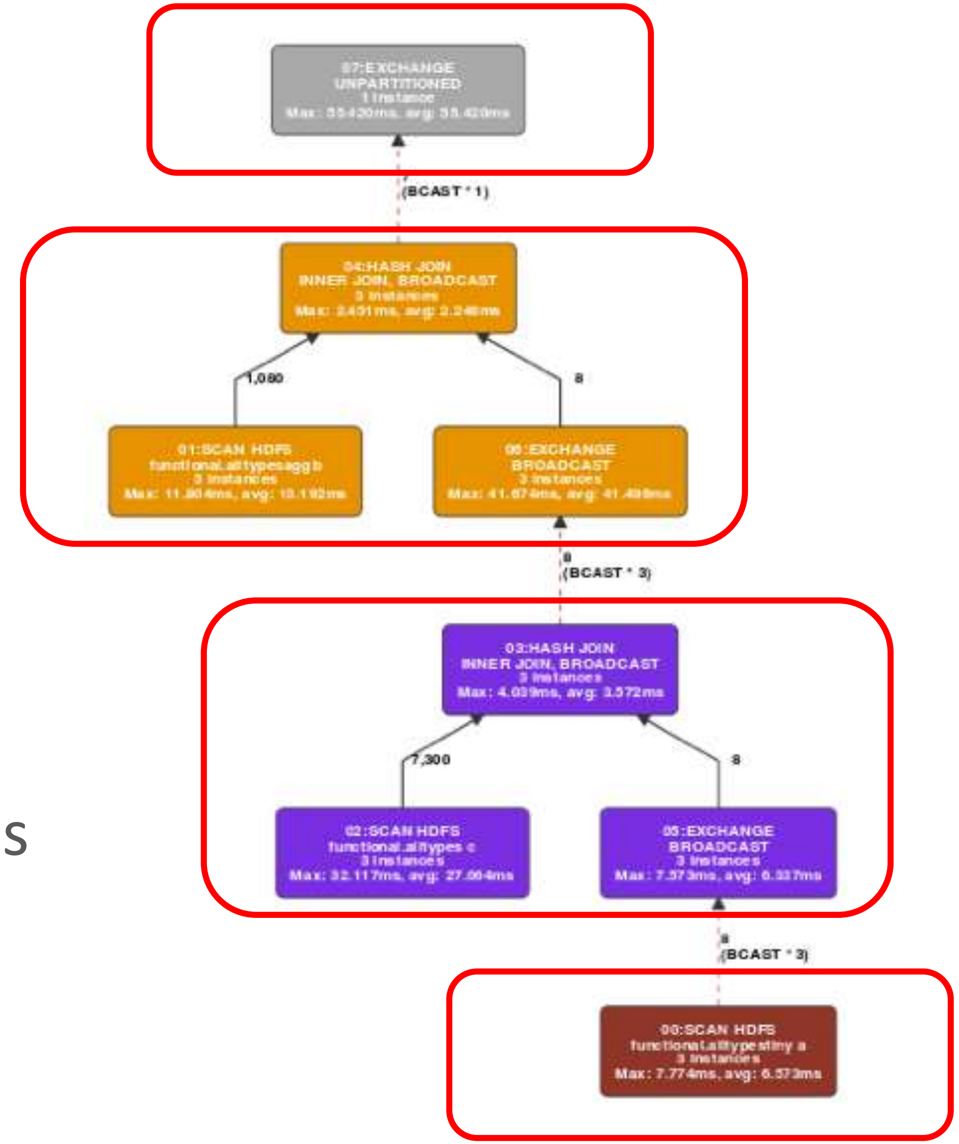
Query planning

2-phase cost-based optimizer

- Phase 1: Generate a single node plan (transformations, join ordering, static partition pruning, runtime filters)
- Phase 2: Convert the single node plan into a distributed query execution plan (add exchange nodes, decide join strategy)

Query fragments (units of work):

- Parts of query execution tree
- Each fragment is executed in one or more impalads



cloudera

Metadata & Statistics

Metadata

- Table metadata (HMS) and block level (HDFS) information are cached to speed-up query time
- Cached data is stored in FlatBuffers to save space and avoid excessive GC
- Metadata loading from HDFS/S3/ADLS uses a thread pool to speedup the operation when needed

Statistics

- Impala uses an HLL to compute Number of distinct values (NDV)
- HLL is much faster than the combination of COUNT and DISTINCT, and uses a constant amount of memory and thus is less memory-intensive for columns with high cardinality.
- HLL size is 1KB per column
- A Novel implementation of sampled stats is coming soon

Query optimizations based on statistics

- Order scan predicates by selectivity and cost, mitigate correlated predicates (exponential backoff)
- Detection of common join pattern of Primary key/Foreign key joins
- 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
- Determine optimal join order

```
SELECT l_shipmode,
       Sum(l_extendedprice)
FROM   orders,
       lineitem
WHERE  o_orderkey =
l_orderkey
       AND l_comment LIKE
'%long string%'
       AND l_receiptdate
>= '1994-01-01'
       AND l_partkey < 100
       AND o_orderdate <
'1993-01-01'
GROUP  BY l_shipmode
ORDER  BY l_shipmode
```

```
02:HASH JOIN [INNER JOIN, BROADCAST]
|  hash predicates: l_partkey = o_orderkey
|  fk/pk conjuncts: 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_partkey < 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

Query optimizations based on statistics

- Order scan predicates by selectivity and cost, mitigate correlated predicates (exponential backoff)
- Detection of common join pattern of Primary key/Foreign key joins
- 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
- Determine optimal join order

```
02:HASH JOIN [INNER JOIN, BROADCAST]
|  hash predicates: l_partkey = o_orderkey
|  fk/pk conjuncts: 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_partkey < 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
```

```sql
SELECT l_shipmode,
       Sum(l_extendedprice)
FROM   orders,
       lineitem
WHERE  o_orderkey =
l_orderkey
       AND l_comment LIKE
'%long string%'
       AND l_receiptdate
>= '1994-01-01'
       AND l_partkey < 100
       AND o_orderdate <
'1993-01-01'
GROUP  BY l_shipmode
ORDER  BY l_shipmode
```

cloudera

Query optimizations based on statistics

- Order scan predicates by selectivity and cost, mitigate correlated predicates (exponential backoff)
- Detection of common join pattern of Primary key/Foreign key joins
- 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
- Determine optimal join order

```
02:HASH JOIN [INNER JOIN, BROADCAST]
|  hash predicates: l_partkey = o_orderkey
|  fk/pk conjuncts: 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_partkey < 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
```

```sql
SELECT l_shipmode,
       Sum(l_extendedprice)
FROM   orders,
       lineitem
WHERE  o_orderkey =
l_orderkey
       AND l_comment LIKE
'%long string%'
       AND l_receiptdate
>= '1994-01-01'
       AND l_partkey < 100
       AND o_orderdate <
'1993-01-01'
GROUP  BY l_shipmode
ORDER  BY l_shipmode
```

Query optimizations based on statistics

- Order scan predicates by selectivity and cost, mitigate correlated predicates (exponential backoff)
- Detection of common join pattern of Primary key/Foreign key joins
- 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
- Determine optimal join order

```
SELECT l_shipmode,
       Sum(l_extendedprice)
FROM   orders,
       lineitem
WHERE  o_orderkey =
l_orderkey
       AND l_comment LIKE
'%long string%'
       AND l_receiptdate
>= '1994-01-01'
       AND l_partkey < 100
       AND o_orderdate <
'1993-01-01'
GROUP  BY l_shipmode
ORDER  BY l_shipmode
```

```
02:HASH JOIN [INNER JOIN, BROADCAST]
|  hash predicates: l_partkey = o_orderkey
|  fk/pk conjuncts: 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_partkey < 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

Query optimizations based on statistics

- Order scan predicates by selectivity and cost, mitigate correlated predicates (exponential backoff)
- Detection of common join pattern of Primary key/Foreign key joins
- 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
- Determine optimal join order

```
SELECT l_shipmode,
       Sum(l_extendedprice)
FROM   orders,
       lineitem
WHERE  o_orderkey =
l_orderkey
       AND l_comment LIKE
'%long string%'
       AND l_receiptdate
>= '1994-01-01'
       AND l_partkey < 100
       AND o_orderdate <
'1993-01-01'
GROUP  BY l_shipmode
ORDER  BY l_shipmode
```

```
02:HASH JOIN [INNER JOIN, BROADCAST]
|  hash predicates: l_partkey = o_orderkey
|  fk/pk conjuncts: 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_partkey < 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

Query optimizations based on statistics

- Order scan predicates by selectivity and cost, mitigate correlated predicates (exponential backoff)
- Detection of common join pattern of Primary key/Foreign key joins
- 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
- Determine optimal join order

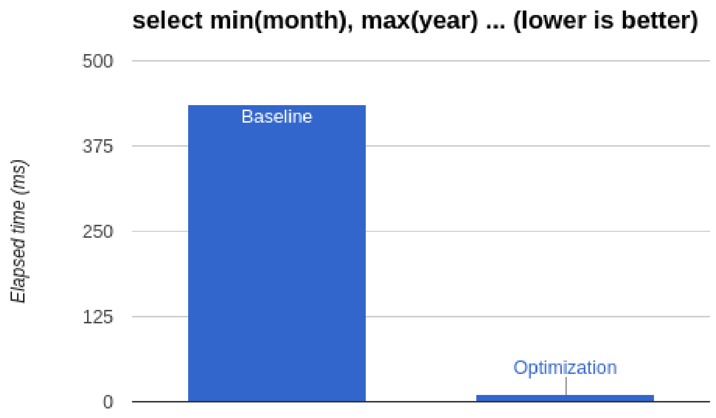
```
02:HASH JOIN [INNER JOIN, BROADCAST]
|  hash predicates: l_partkey = o_orderkey
|  fk/pk conjuncts: l_orderkey = o_orderkey
|  runtime filters: RF000 <- o_orderkey
|  tuple-ids=1,0 row-size=113B cardinality=27,381,196
|
|--05:EXCHANGE [BROADCAST]
|  |  hosts=20 per-host-mem=0B
|  |  tuple-ids=0 row-size=8B 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_partkey < 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
```

```
SELECT l_shipmode,
       Sum(l_extendedprice)
FROM   orders,
       lineitem
WHERE  o_orderkey =
l_orderkey
       AND l_comment LIKE
'%long string%'
       AND l_receiptdate
>= '1994-01-01'
       AND l_partkey < 100
       AND o_orderdate <
'1993-01-01'
GROUP  BY l_shipmode
ORDER  BY l_shipmode
```

cloudera

Optimization for metadata only queries

- Use metadata to avoid table accesses for partition key scans:
  - select min(month), max(year) from functional.alltypes;
  - month, year are partition keys of the table
- Enabled by query option OPTIMIZE_PARTITION_KEY_SCANS
- Applicable:
  - min(), max(), ndv() and aggregate functions with **distinct** keyword
  - partition keys only



Plan without optimization

```
03:AGGREGATE [FINALIZE]
|   output: min:merge(month),
max:merge(year)
|
02:EXCHANGE [UNPARTITIONED]
|
01:AGGREGATE
|   output: min(month), max(year)
|
00:SCAN HDFS [functional.alltypes]
    partitions=24/24 files=24 size=478.45KB
```

Plan with optimization

```
01:AGGREGATE [FINALIZE]
|   output: min(month),max(year)
|
00:UNION
    constant-operands=24
```

Scanner : Extract common conjuncts from disjunctions

- Extract common conjuncts from disjunctions.
  (a AND b) OR (a AND b) ==> a AND b
  (a AND b AND c) OR (c) ==> c

- >100x speedup for TPC-DS Q13, Q48 & TPC-H Q19

```
00:SCAN HDFS [tpch_300_parquet.lineitem, RANDOM]
   partitions=1/1 files=259 size=63.71GB
   predicates: l_shipmode IN ('AIR', 'AIR REG'), l_shipinstruct = 'DELIVER IN PERSON'
   runtime filters: RF000 -> l_partkey
   table stats: 1799989091 rows total
   hosts=7 per-host-mem=528.00MB
   tuple-ids=0 row-size=80B cardinality=240533660
```

```
Select sum(l_extendedprice* (1 - l_discount)) as revenue
from
       Lineitem, part
Where      (          p_partkey = l_partkey
            and p_brand = 'Brand#32'
            and p_container in ('SM CASE', 'SM BOX',
            'SM PACK', 'SM PKG')
            and l_quantity >= 7 and l_quantity <= 7 +
            10
            and p_size between 1 and 5
            and l_shipmode in ('AIR', 'AIR REG')
            and l_shipinstruct = 'DELIVER IN PERSON')
       or( p_partkey = l_partkey
            and p_brand = 'Brand#35'
            and p_container in ('MED BAG', 'MED BOX',
            'MED PKG', 'MED PACK')
            and l_quantity >= 15 and l_quantity <= 15
            + 10
            and p_size between 1 and 10
            and l_shipmode in ('AIR', 'AIR REG')
            and l_shipinstruct = 'DELIVER IN PERSON')
       Or( p_partkey = l_partkey
            and p_brand = 'Brand#24'
            and p_container in ('LG CASE', 'LG BOX',
            'LG PACK', 'LG PKG')
            and l_quantity >= 26 and l_quantity <= 26
            + 10
            and p_size between 1 and 15
            and l_shipmode in ('AIR', 'AIR REG')
            and l_shipinstruct = 'DELIVER IN PERSON')
```

cloudera

Query execution

- **Not based on Map-Reduce**
- Single-threaded, row-based, Volcano-style (iterator-based with batches) query execution engine

- Multi-threaded scans
- Utilizes HDFS short-circuit local reads & multi-threaded I/O subsystem
- Supports multiple file formats (Parquet, Avro, Text, Sequence, …)
- Supports nested types (only in Parquet)
- Code generation using LLVM

- No transactions
- No indices

Partitioning

- Impala does not have native indexes (e.g. B+-tree), but it does allow a type of indexing by partitions
- What it is: physically dividing your data so that queries only need to access a subset
- Partitioning schema is expressed through DDL
- Pruning 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)
```

cloudera

Optimizations for selective scans: Sorting

- Sorting data files improves the effectiveness of file statistics (min/max) and compression (e.g., delta encoding).
- Predicates are evaluated against Min/Max statistics as well as Dictionary encoded columns, this approximates lazy materialization
- 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.
- 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;
```

cloudera

Optimizations for selective scans: Augment Partitioning

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

SORT BY 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 = 2017   AND Month=12
  AND 12 AND hour BETWEEN 1 AND 4
GROUP BY customer_name
ORDER BY revenue DESC LIMIT 10;
```

Metric	Partition + Sorting Speedup
Elapsed time	3x
CPU time (seconds)	5x
HDFS MBytes read	17x

```
CREATE TABLE Sales (...)
PARTITIONED BY (year INT, month INT)
SORT BY (day, hour)
Stored as Parquet;
```

cloudera

Optimizations for selective scans:Complement Partitioning

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

SORT BY helps meet query SLAs without over partitioning the table

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

Metric	Partition + Sorting Speedup
Elapsed time	18x
HDFS MBytes read	22x

```
CREATE TABLE Sales (...)
PARTITIONED BY (year INT, month INT)
SORT BY (customer_id)
Stored as Parquet;
```

cloudera

LLVM Codegen in Impala

- Impala uses runtime code generation to produce query specific versions of functions that are critical to performance.
- In particular, code generation is applied to “inner loop” functions
- Code generation (codegen) lets Impala use query-specific information to do less work
  - Remove conditionals
  - Propagate constant offsets, pointers, etc.
  - Inline virtual functions calls

cloudera

LLVM Codegen in Impala

Operations:

- Hash join
- Aggregation
- Scans: Parquet, Text, Sequence, Avro
- Expressions in all operators
- Union
- Sort
- Top-N
- Runtime filters

Data Types:

- TINYINT, SMALLINT, INT, BIGINT
- FLOAT, DOUBLE
- BOOLEAN
- STRING, VARCHAR
- DECIMAL

Further optimizations:

- Don't codegen short running queries (Relies on cardinality estimates)
- Limit amount of inlining for long and complex expressions (Group by 1K columns)

cloudera

Codegen for Order by & Top-N

```
select
    l_extendedprice, l_orderkey
from
    lineitem
order by l_orderkey
limit 100
```

l_orderkey	l_extendedprice	l_shipdate	l_shipmode
26039617	13515	12/8/1997	FOB
30525093	16218	12/16/1997	REG AIR
28809990	7208	10/19/1997	AIR

cloudera

Codegen for Order by & Top-N

```
int Compare(TupleRow* lhs, TupleRow* rhs) const {
  for (int i = 0; i < sort_cols_lhs_.size(); ++i) {
    void* lhs_value = sort_cols_lhs_[i]->GetValue(lhs);
    void* rhs_value = sort_cols_rhs_[i]->GetValue(rhs);

    if (lhs_value == NULL && rhs_value != NULL) return nulls_first_[i];
    if (lhs_value != NULL && rhs_value == NULL) return -nulls_first_[i];

    int result = RawValue::Compare(lhs_value, rhs_value,
                                   sort_cols_lhs_[i]->root()->type());
    if (!is_asc_[i]) result = -result;
    if (result != 0) return result;
    // Otherwise, try the next Expr
  }
  return 0; // fully equivalent key
}
```

```
select
   l_extendedprice, l_orderkey
from
   lineitem
order by l_orderkey
limit 100
```

l_orderkey	l_extendedprice	l_shipdate	l_shipmode
26039617	13515	12/8/1997	FOB
30525093	16218	12/16/1997	REG AIR
28809990	7208	10/19/1997	AIR

cloudera

Codegen for Order by & Top-N

```
int Compare(TupleRow* lhs, TupleRow* rhs) const {
  for (int i = 0; i < sort_cols_lhs_.size(); ++i) {
    void* lhs_value = sort_cols_lhs_[i]->GetValue(lhs);
    void* rhs_value = sort_cols_rhs_[i]->GetValue(rhs);

    if (lhs_value == NULL && rhs_value != NULL) return nulls_first_[i];
    if (lhs_value != NULL && rhs_value == NULL) return -nulls_first_[i];

    int result = RawValue::Compare(lhs_value, rhs_value,
                                   sort_cols_lhs_[i]->root()->type());
    if (!is_asc_[i]) result = -result;
    if (result != 0) return result;
    // Otherwise, try the next Expr
  }
  return 0; // fully equivalent key
}
```

```
void* ExprContext::GetValue(Expr* e, TupleRow* row) {
  switch (e->type_.type) {
    case TYPE_BOOLEAN: {
      ..
      ..
    }
    case TYPE_TINYINT: {
      ..
      ..
    }
    case TYPE_INT: {
      ..
      .
```

cloudera

Codegen for Order by & Top-N

```
int Compare(TupleRow* lhs, TupleRow* rhs) const {
   for (int i = 0; i < sort_cols_lhs_.size(); ++i) {
     void* lhs_value = sort_cols_lhs_[i]->GetValue(lhs);
     void* rhs_value = sort_cols_rhs_[i]->GetValue(rhs);

     if (lhs_value == NULL && rhs_value != NULL) return nulls_first_[i];
     if (lhs_value != NULL && rhs_value == NULL) return -nulls_first_[i];

     int result = RawValue::Compare(lhs_value, rhs_value,
                                    sort_cols_lhs_[i]->root()->type());
     if (!is_asc_[i]) result = -result;
     if (result != 0) return result;
     // Otherwise, try the next Expr
   }
   return 0; // fully equivalent key
  }
```

```
void* ExprContext::GetValue(Expr* e, TupleRow* row) {
  switch (e->type_.type) {
    case TYPE_BOOLEAN: {
      ..
      ..
    }
    case TYPE_TINYINT: {
      ..
      ..
    }
    case TYPE_INT: {
      ..
      .
```

Codegen for Order by & Top-N

```
int Compare(TupleRow* lhs, TupleRow* rhs) const {
  for (int i = 0; i < sort_cols_lhs_.size(); ++i) {
    void* lhs_value = sort_cols_lhs_[i]->GetValue(lhs);
    void* rhs_value = sort_cols_rhs_[i]->GetValue(rhs);

    if (lhs_value == NULL && rhs_value != NULL) return nulls_first_[i];
    if (lhs_value != NULL && rhs_value == NULL) return -nulls_first_[i];

    int result = RawValue::Compare(lhs_value, rhs_value,
                                   sort_cols_lhs_[i]->root()->type());
    if (!is_asc_[i]) result = -result;
    if (result != 0) return result;
    // Otherwise, try the next Expr
  }
  return 0; // fully equivalent key
}
```

```
int RawValue::Compare(const void* v1, const void* v2,
const ColumnType& type) {
  switch (type.type) {
    case TYPE_INT:
      i1 = *reinterpret_cast<const int32_t*>(v1);
      i2 = *reinterpret_cast<const int32_t*>(v2);
      return i1 > i2 ? 1 : (i1 < i2 ? -1 : 0);
    case TYPE_BIGINT:
      b1 = *reinterpret_cast<const int64_t*>(v1);
      b2 = *reinterpret_cast<const int64_t*>(v2);
      return b1 > b2 ? 1 : (b1 < b2 ? -1 : 0);
```

cloudera

Codegen for Order by & Top-N

Original code

```
int Compare(TupleRow* lhs, TupleRow* rhs) const {
  for (int i = 0; i < sort_cols_lhs_.size(); ++i) {
    void* lhs_value = sort_cols_lhs_[i]->GetValue(lhs);
    void* rhs_value = sort_cols_rhs_[i]->GetValue(rhs);

    if (lhs_value == NULL && rhs_value != NULL) return nulls_first_[i];
    if (lhs_value != NULL && rhs_value == NULL) return -nulls_first_[i];

    int result = RawValue::Compare(lhs_value, rhs_value,
                                   sort_cols_lhs_[i]->root()->type());
    if (!is_asc_[i]) result = -result;
    if (result != 0) return result;
    // Otherwise, try the next Expr
  }
  return 0; // fully equivalent key
}
```

Codegen code

```
int CompareCodgened(TupleRow* lhs, TupleRow* rhs) const {
    int64_t lhs_value = sort_columns[i]->GetBigIntVal(lhs);
    int64_t rhs_value = sort_columns[i]->GetBigIntVal(rhs);

    int result = lhs_value > rhs_value ? 1 :
                 (lhs_value < rhs_value ? -1 : 0);
    if (result != 0) return result;
    // Otherwise, try the next Expr
  return 0; // fully equivalent key
```

cloudera

Codegen for Order by & Top-N

Original code

```
int Compare(TupleRow* lhs, TupleRow* rhs) const {
  for (int i = 0; i < sort_cols_lhs_.size(); ++i) {
    void* lhs_value = sort_cols_lhs_[i]->GetValue(lhs);
    void* rhs_value = sort_cols_rhs_[i]->GetValue(rhs);

    if (lhs_value == NULL && rhs_value != NULL) return nulls_first_[i];
    if (lhs_value != NULL && rhs_value == NULL) return -nulls_first_[i];

    int result = RawValue::Compare(lhs_value, rhs_value,
                                   sort_cols_lhs_[i]->root()->type());
    if (!is_asc_[i]) result = -result;
    if (result != 0) return result;
    // Otherwise, try the next Expr
  }
  return 0; // fully equivalent key
}
```

Codegen code

```
int CompareCodgened(TupleRow* lhs, TupleRow* rhs) const {
    int64_t lhs_value = sort_columns[i]->GetBigIntVal(lhs); // i = 0
    int64_t rhs_value = sort_columns[i]->GetBigIntVal(rhs); // i = 1

    int result = lhs_value > rhs_value ? 1 :
                 (lhs_value < rhs_value ? -1 : 0);
    if (result != 0) return result;
    // Otherwise, try the next Expr
   return 0; // fully equivalent key
}
```

- Perfectly unrolls “for each grouping column” loop
- No switching on input type(s)
- Removes branching on ASCENDING/DESCENDING, NULLS FIRST/LAST

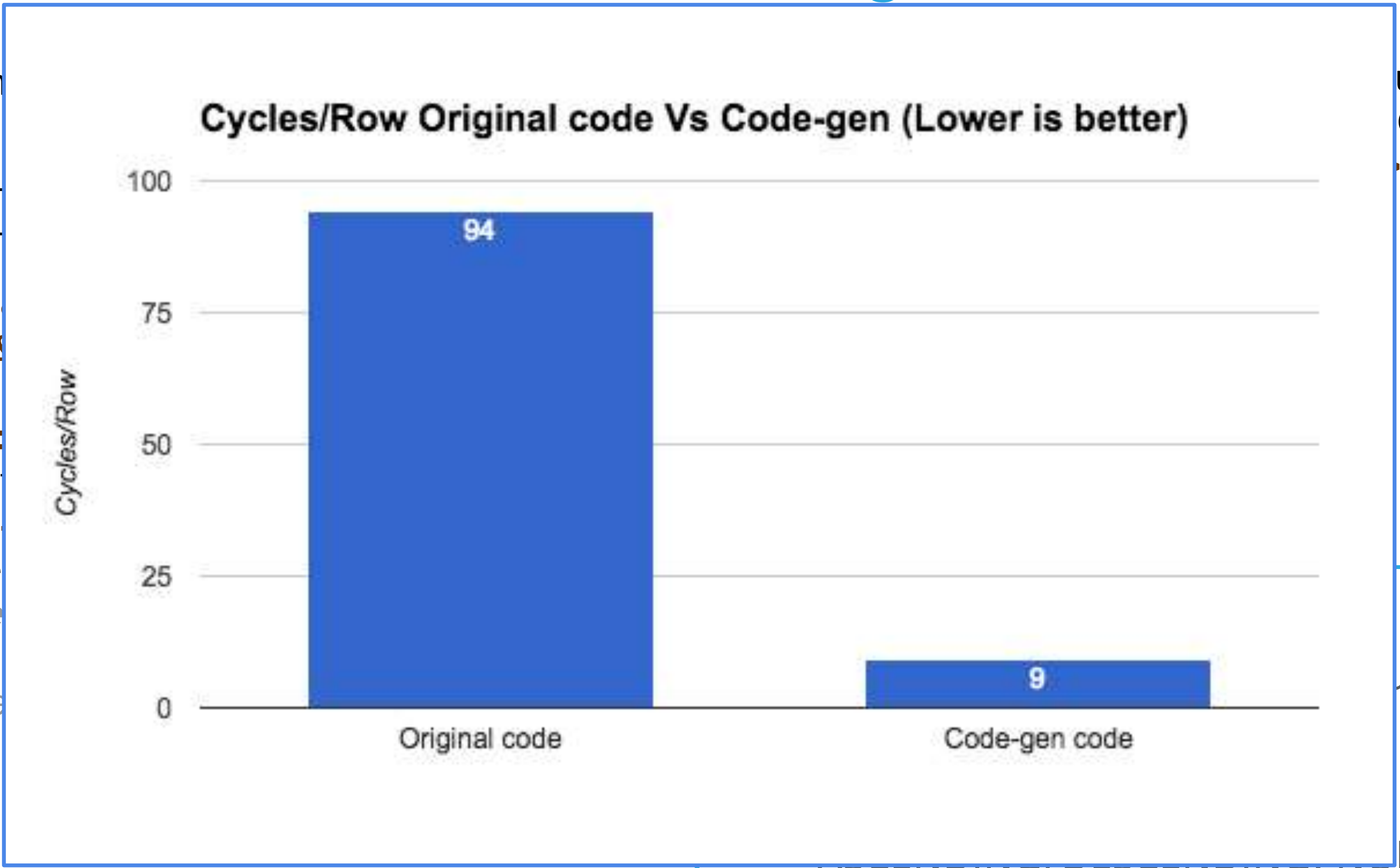
cloudera

Codegen for Order by & Top-N

Original code

```
int Compare(TupleRow* lh
  for (int i = 0; i < sort_col
    void* lhs_value = sort_
    void* rhs_value = sort_

    if (lhs_value == NULL &
    if (lhs_value != NULL &

    int result = RawValue::
                      sor
    if (!is_asc_[i]) result = 
    if (result != 0) return re
    // Otherwise, try the ne
  }
  return 0; // fully equivale
}
```

Codegen code




```
upleRow* rhs) const {
GetBigIntVal(lhs);
GetBigIntVal(rhs);

:
: 0);
```

uping column”

LLS FIRST/LAST

cloudera

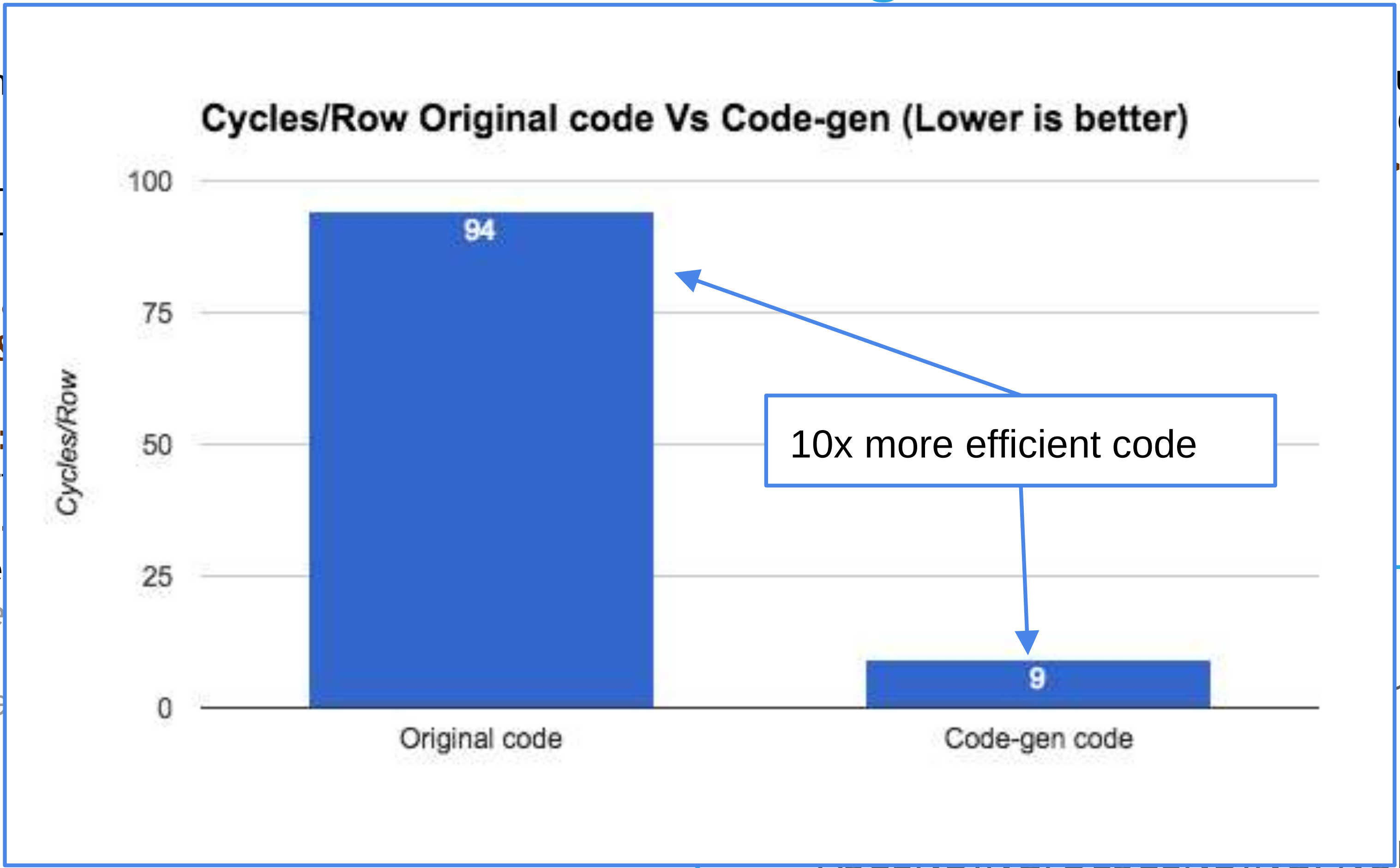
Codegen for Order by & Top-N

Original code

```
int Compare(TupleRow* lh
  for (int i = 0; i < sort_col
    void* lhs_value = sort_
    void* rhs_value = sort_

    if (lhs_value == NULL &
    if (lhs_value != NULL &

    int result = RawValue::
                      sor
    if (!is_asc_[i]) result =
    if (result != 0) return re
    // Otherwise, try the ne
  }
  return 0; // fully equivale
}
```

Codegen code

```
upleRow* rhs) const {
GetBigIntVal(lhs);
GetBigIntVal(rhs);

:
: 0);
```

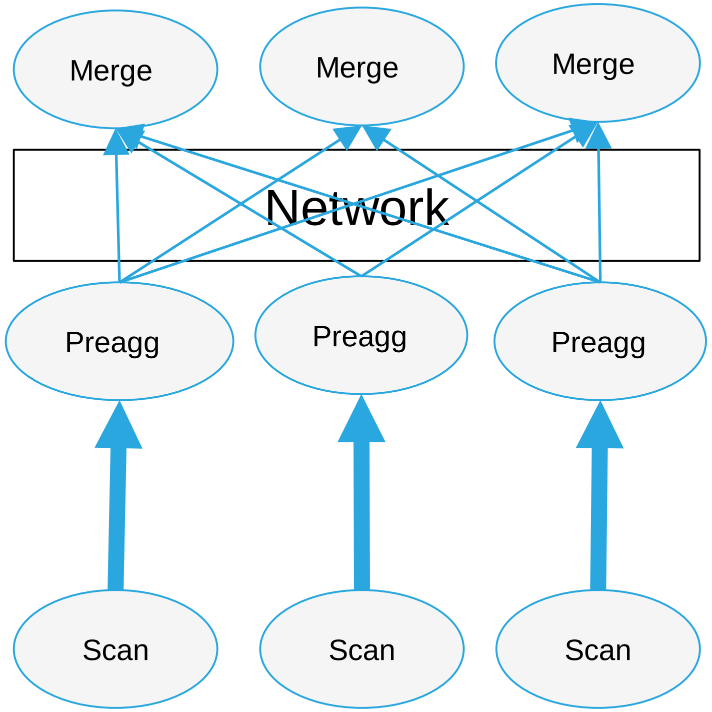


uping column”

LLS FIRST/LAST

Distributed Aggregation in MPP

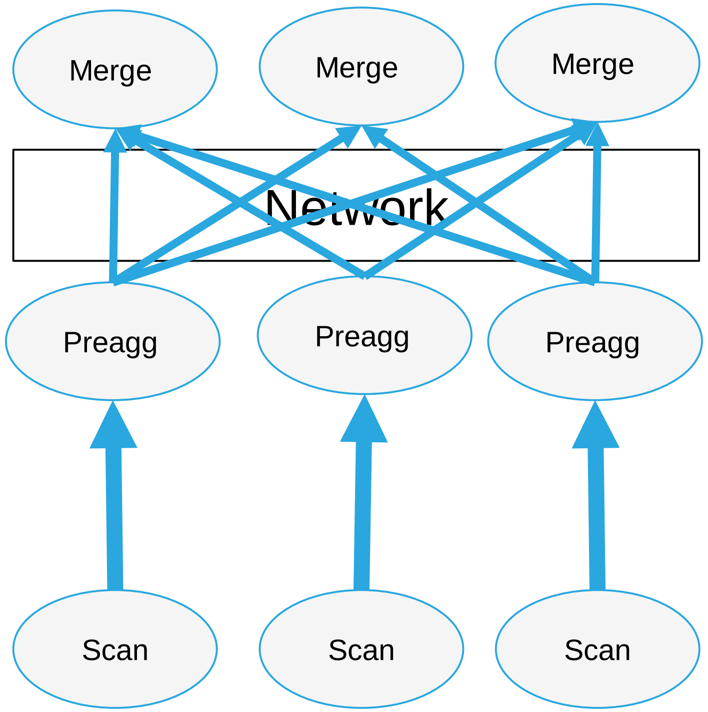
```
select cust_id, sum(dollars)
from sales group by cust_id;
```



- Aggregations have two phases:
  - Pre-aggregation phase
  - Merge phase
- The pre-aggregation phase greatly reduces network traffic ***if*** there are many input rows per grouping value.
  - E.g. many sales per customer.

Downside of Pre-aggregations

```
select distinct * from sales;
```

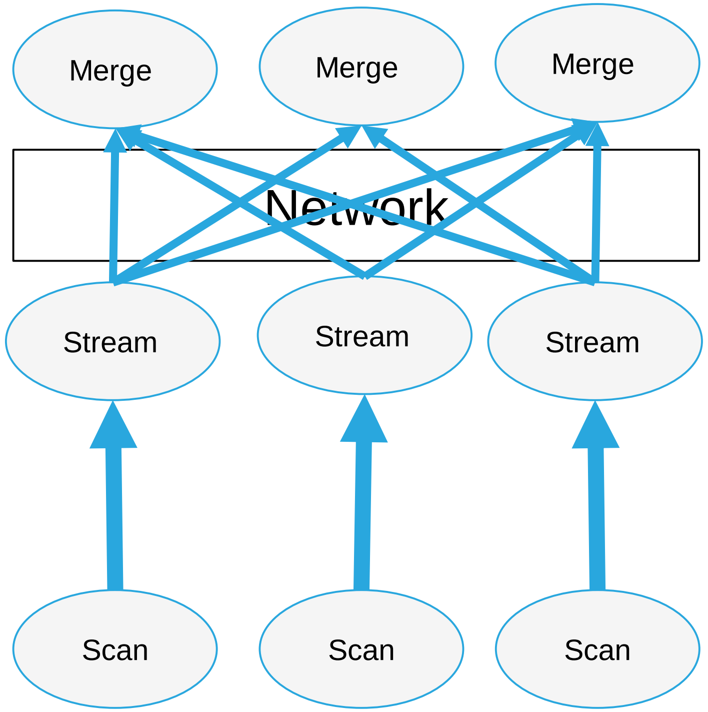


- Pre-aggregations consume:
  - Memory
  - CPU cycles
- Pre-aggregations are not always effective at reducing network traffic
  - E.g. select distinct for nearly-distinct rows
- Pre-aggregations can spill to disk under memory pressure
  - Disk I/O is bad - better to send to merge agg rather than disk

cloudera

Streaming Pre-aggregations in Impala

```
select distinct * from sales;
```



- *Reduction factor* is dynamically estimated based on the actual data processed
- Pre-aggregation expands memory usage only if reduction factor is good
- Benefits:
  - Certain aggregations with low reduction factor see speedups of up to 40%
  - Memory consumption can be reduced by 50% or more
  - **Streaming pre-aggregations don't spill to disk**

cloudera

Optimizations for selective joins : Runtime filters

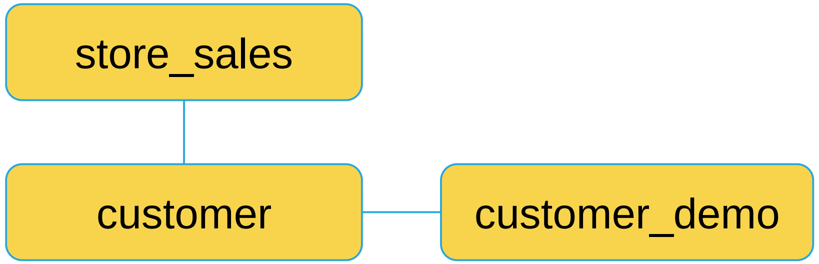
- When are Runtime filters useful
  - To Optimize selective equi-joins against large partitioned or unpartitioned tables

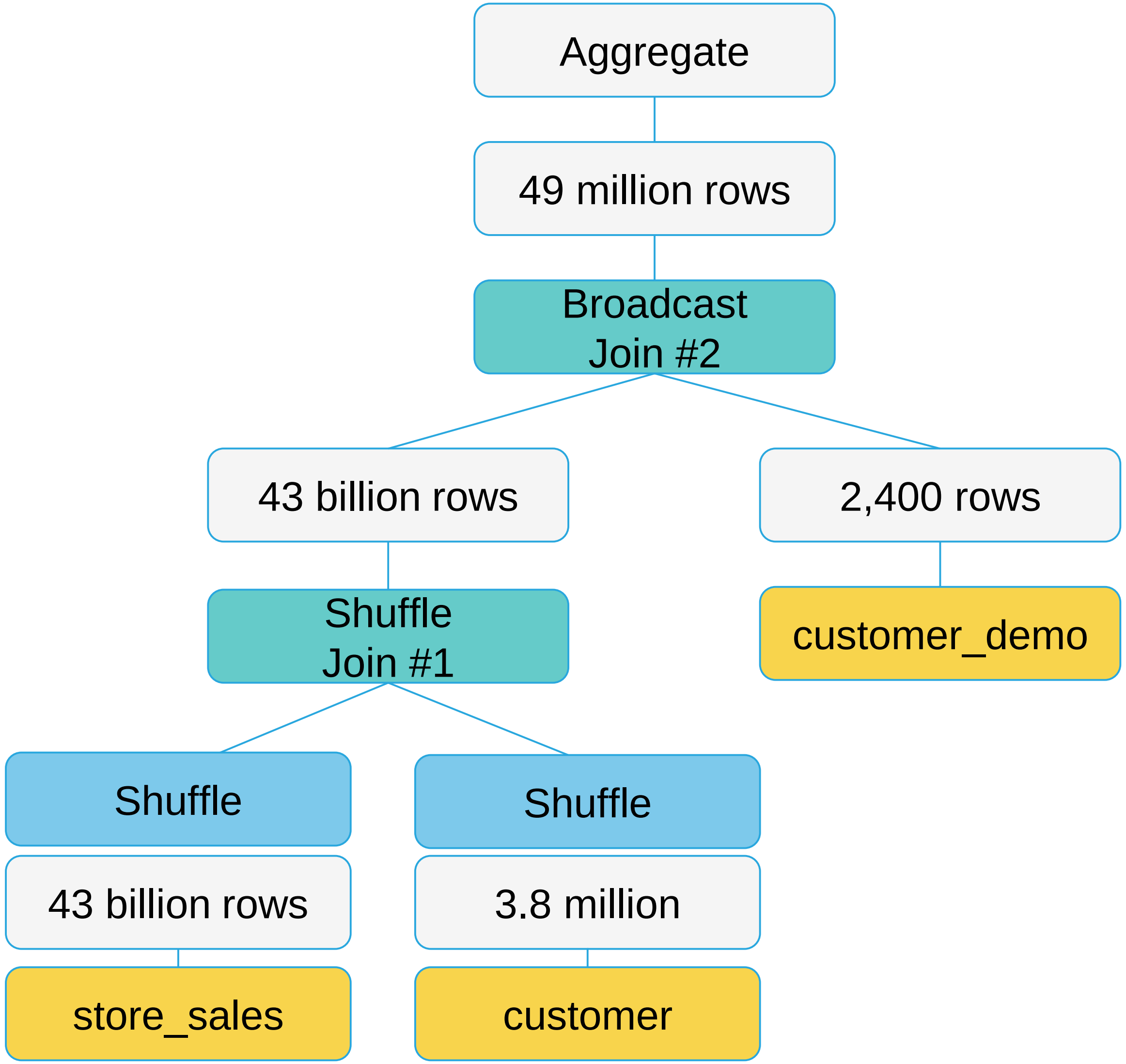
- General idea
  - Avoid unnecessary I/O to read partition data, and avoid unnecessary network transmission by sending only the subset of rows that match the join keys across the network
  - Some predicates can only be computed at *runtime*
  - Use a Bloom filter, which uses a probability-based algorithm to store and query values for joins column(s)

cloudera

Runtime filters in action

```
  SELECT c_email_address
             ,sum(ss_ext_sales_price) sum_agg
FROM store_sales
             ,customer
             ,customer_demographics
WHERE ss_customer_sk = c_customer_sk
             AND cd_demo_sk = c_current_cdemo_sk
             AND cd_gender = 'M'
             AND cd_purchase_estimate = 10000
             AND cd_credit_reting = 'Low Risk'
GROUP BY c_email_address
ORDER BY sum_agg DESC
```

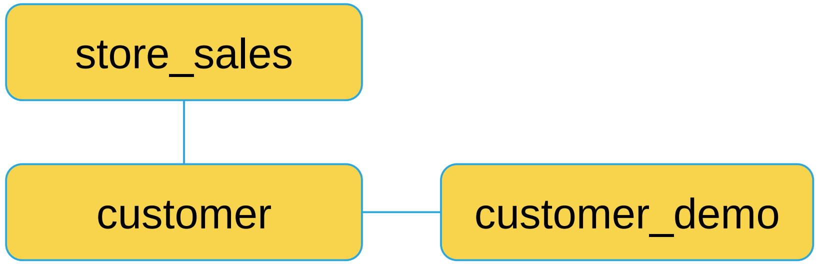


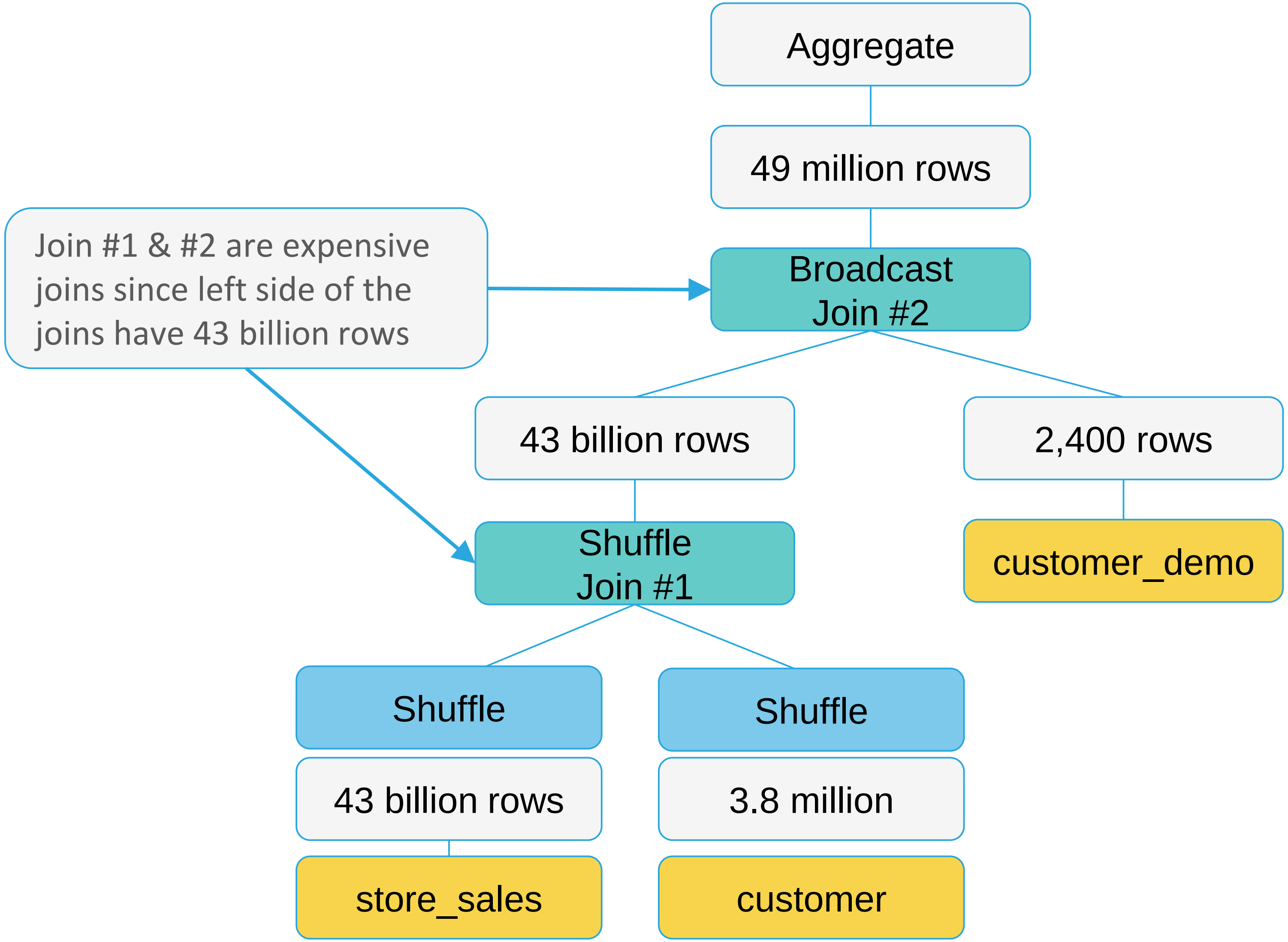


cloudera

Runtime filters in action

```
  SELECT c_email_address
             ,sum(ss_ext_sales_price) sum_agg
FROM store_sales
             ,customer
             ,customer_demographics
WHERE ss_customer_sk = c_customer_sk
             AND cd_demo_sk = c_current_cdemo_sk
             AND cd_gender = 'M'
             AND cd_purchase_estimate = 10000
             AND cd_credit_reting = 'Low Risk'
GROUP BY c_email_address
ORDER BY sum_agg DESC
```

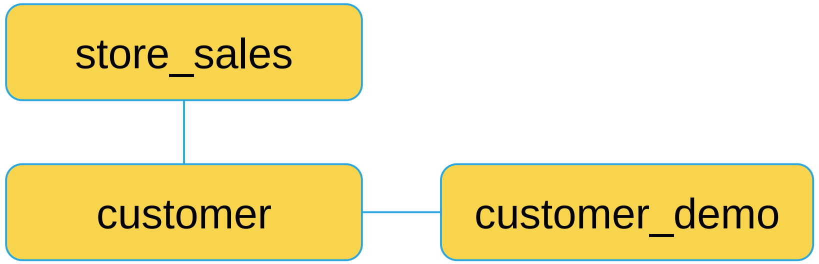


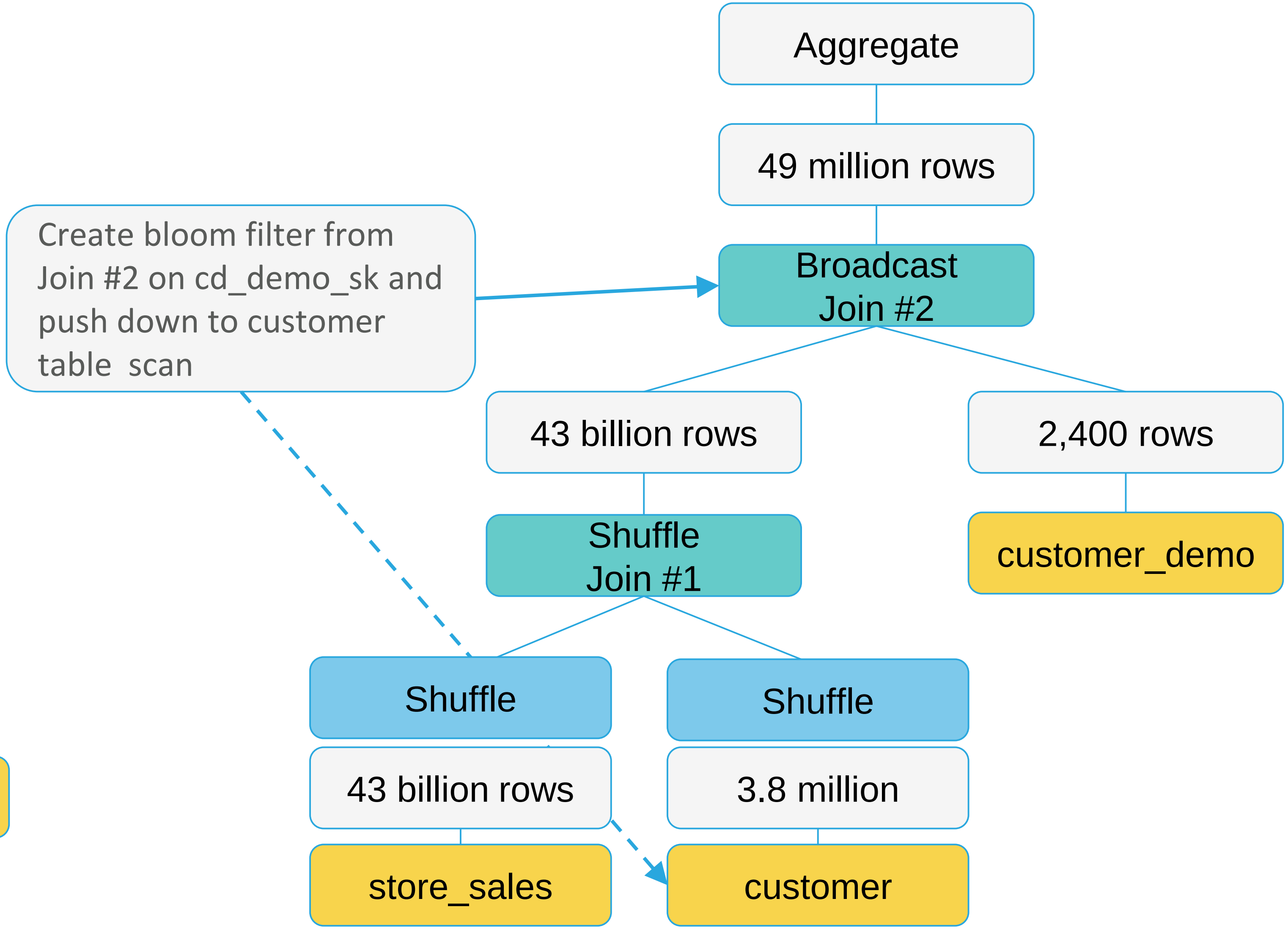


cloudera

Runtime filters in action

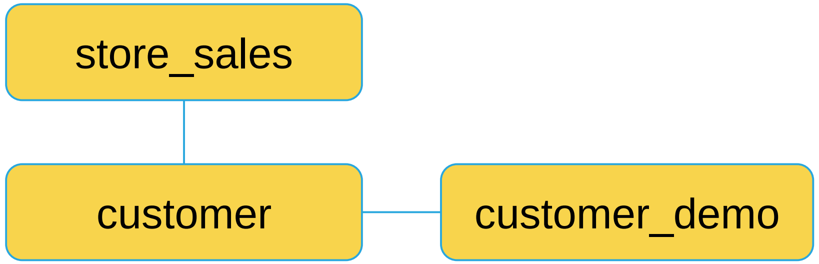
```
  SELECT c_email_address
              ,sum(ss_ext_sales_price) sum_agg
FROM store_sales
              ,customer
              ,customer_demographics
WHERE ss_customer_sk = c_customer_sk
              AND cd_demo_sk = c_current_cdemo_sk
              AND cd_gender = ‘M’
              AND cd_purchase_estimate = 10000
              AND cd_credit_reting = ‘Low Risk’
GROUP BY c_email_address
ORDER BY sum_agg DESC
```

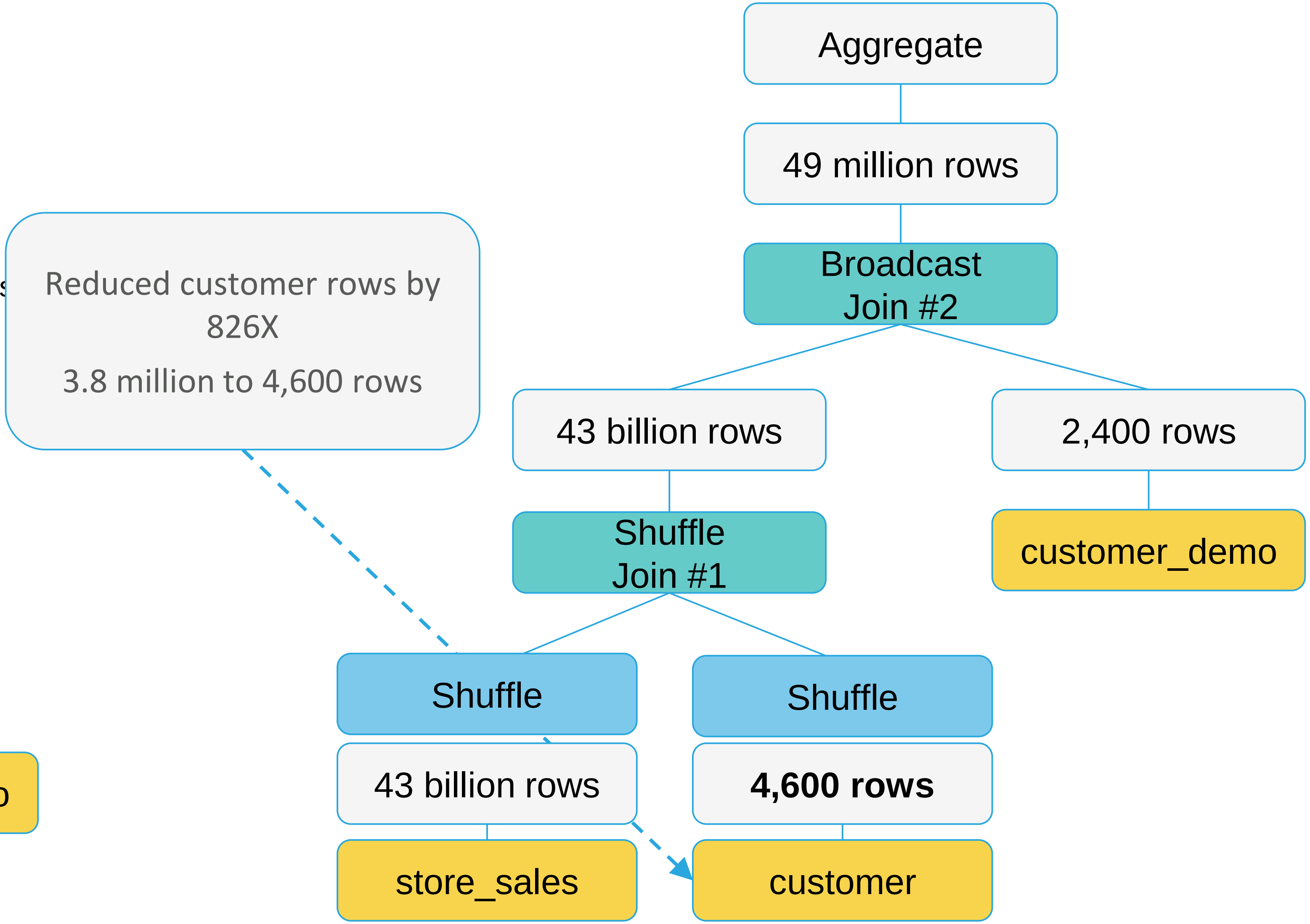




Runtime filters in action

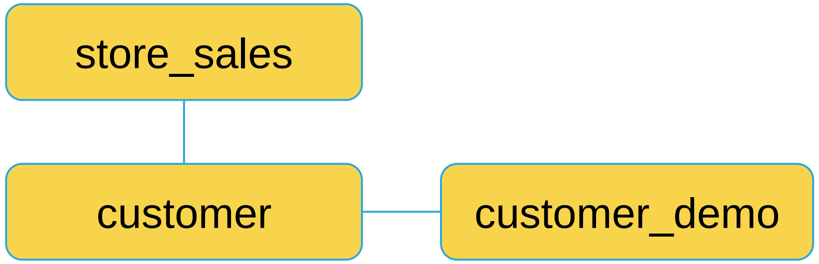
```
SELECT c_email_address
        ,sum(ss_ext_sales_price) sum_agg
FROM store_sales
        ,customer
        ,customer_demographics
WHERE ss_customer_sk = c_customer_sk
        AND cd_demo_sk = c_current_cdemo_s
        AND cd_gender = 'M'
        AND cd_purchase_estimate = 10000
        AND cd_credit_reting = 'Low Risk'
GROUP BY c_email_address
ORDER BY sum_agg DESC
```

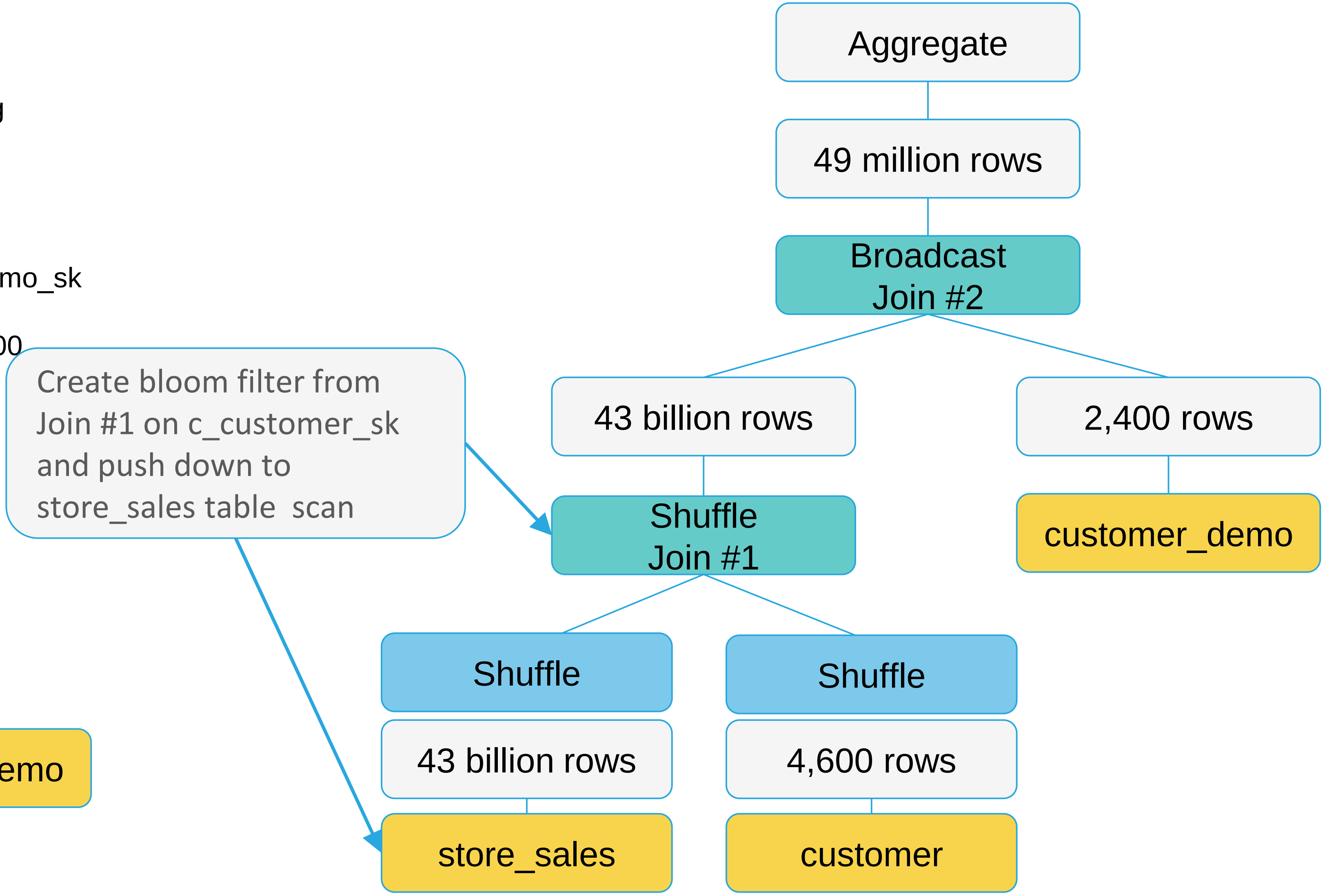




Runtime filters in action

```sql
  SELECT c_email_address
              ,sum(ss_ext_sales_price) sum_agg
FROM store_sales
              ,customer
              ,customer_demographics
WHERE ss_customer_sk = c_customer_sk
              AND cd_demo_sk = c_current_cdemo_sk
              AND cd_gender = 'M'
              AND cd_purchase_estimate = 10000
              AND cd_credit_reting = 'Low Risk'
GROUP BY c_email_address
ORDER BY sum_agg DESC
```



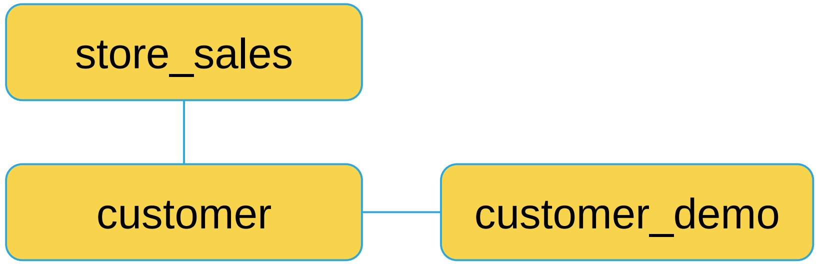


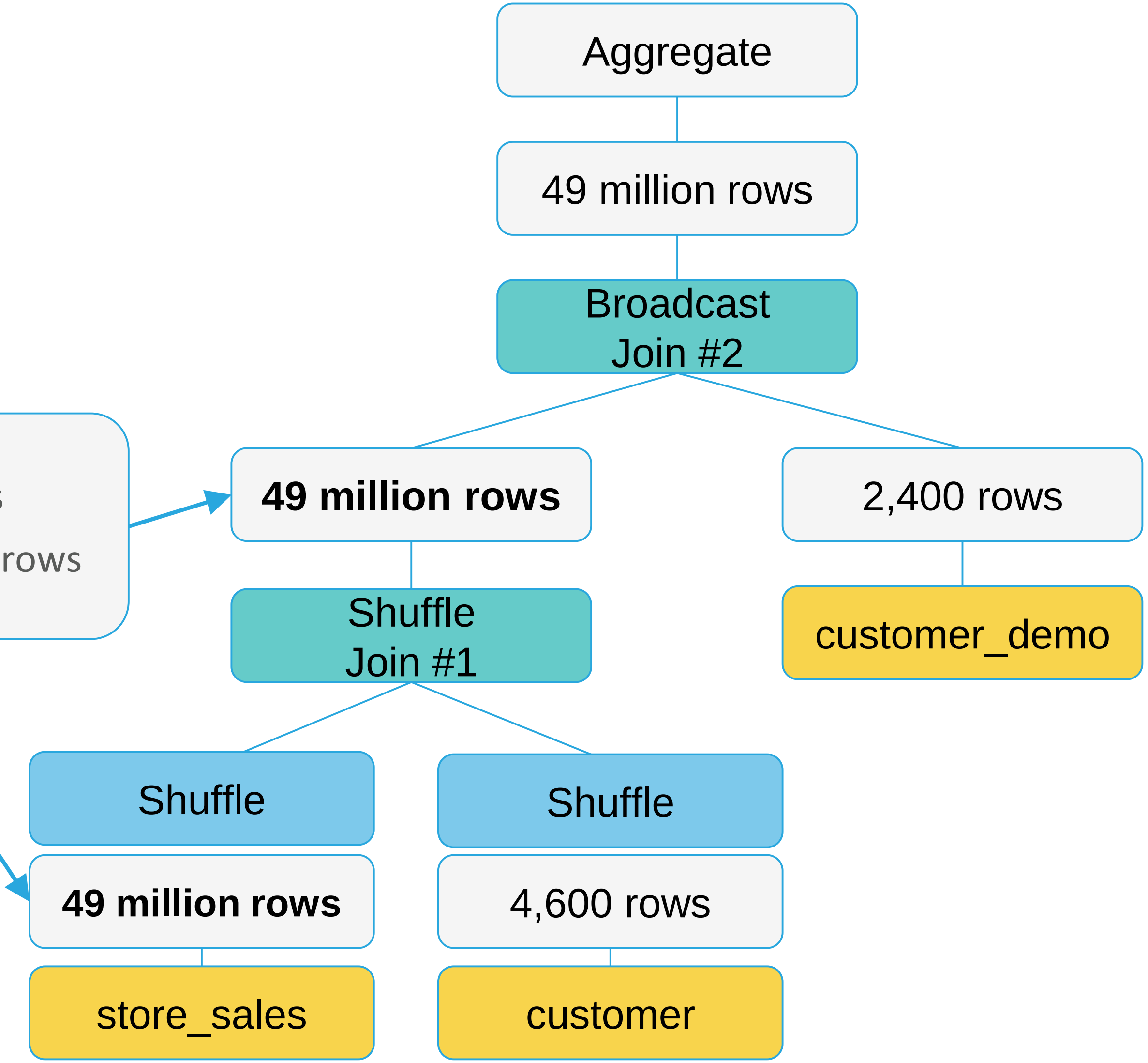
Runtime filters in action

```
  SELECT c_email_address
            ,sum(ss_ext_sales_price) sum_agg
FROM store_sales
            ,customer
            ,customer_demographics
WHERE ss_customer_sk = c_customer_sk
            AND cd_demo_sk = c_current_cdemo_sk
            AND cd_gender = ‘M’
            AND cd_purchase_estimate = 10000
            AND cd_credit_reting = ‘Low Risk’
GROUP BY c_email_address
ORDER BY sum_agg DESC
```

877x reduction in rows

43 billion -> 49 million rows



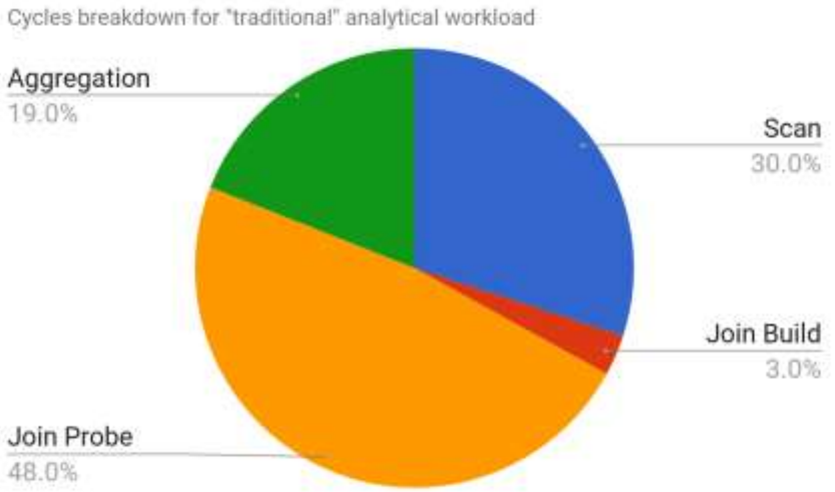


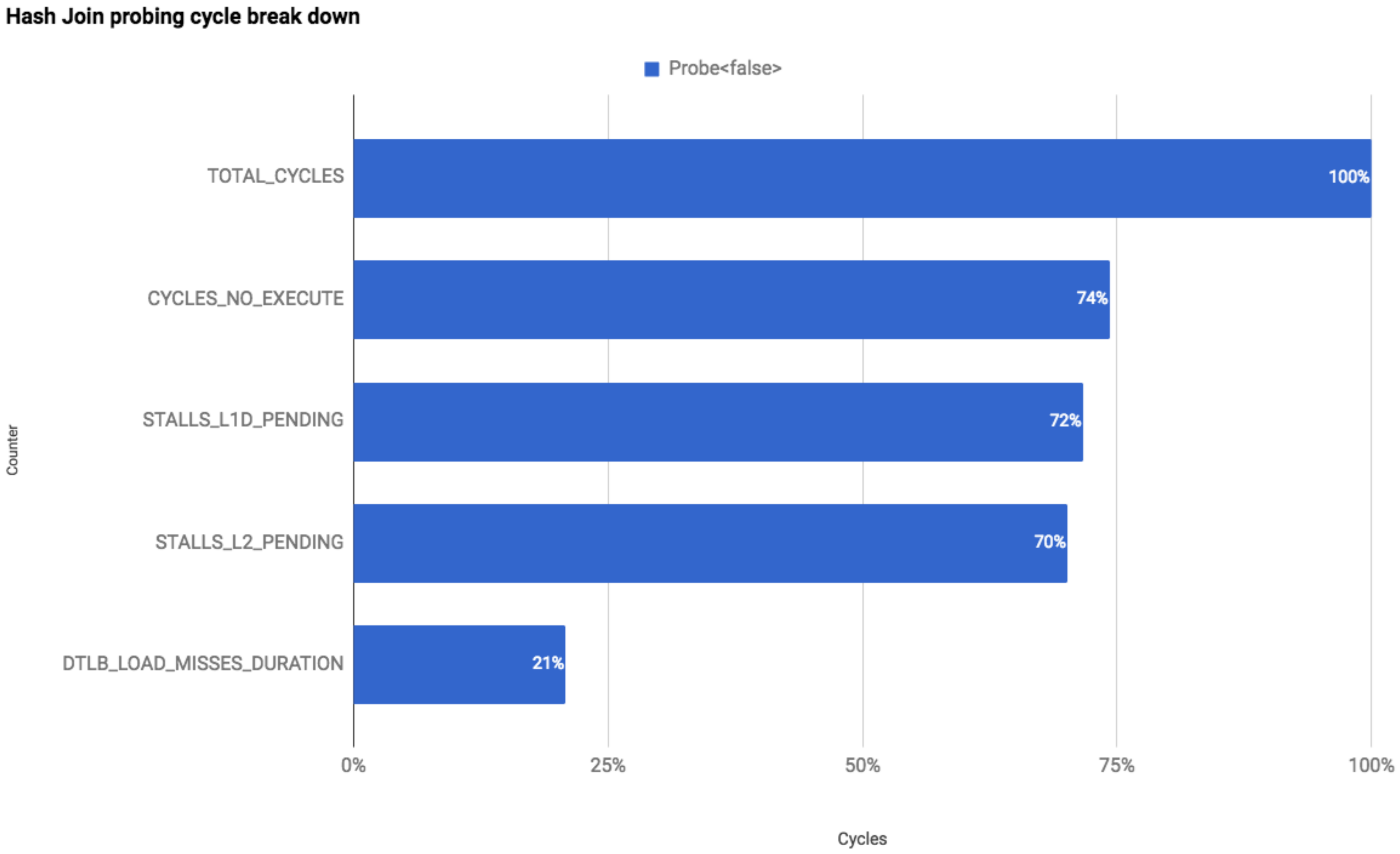
cloudera

Optimizations for selective joins : Runtime filters

- Cache friendly, hash each key to a Bloom filter the size of a cache line or smaller
- Use AVX2 instruction set for Insert and Find operations
- Planner uses a cost model to decide which joins benefit from Runtime Filters
- Bloom filters are sized based on estimated number of distinct values for the build side (1MB-16MB by default)
- Ineffective runtime filters are disabled dynamically when
  - Bloom filters has a higher false positive rate
  - Filter selectivity (Reduce CPU overhead)
- Track Min and Max values from build side and push new implied predicates to the scan of the probe side (Supported for Kudu in CDH5.14, Parquet support coming soon)
- Runtime filters are codegened, avoids virtual functions calls and branching in inner loop

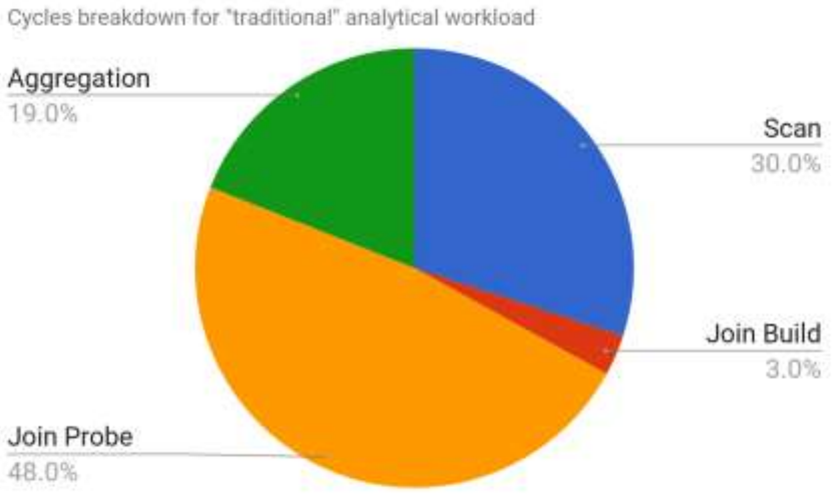
Joins and aggregations under the hood

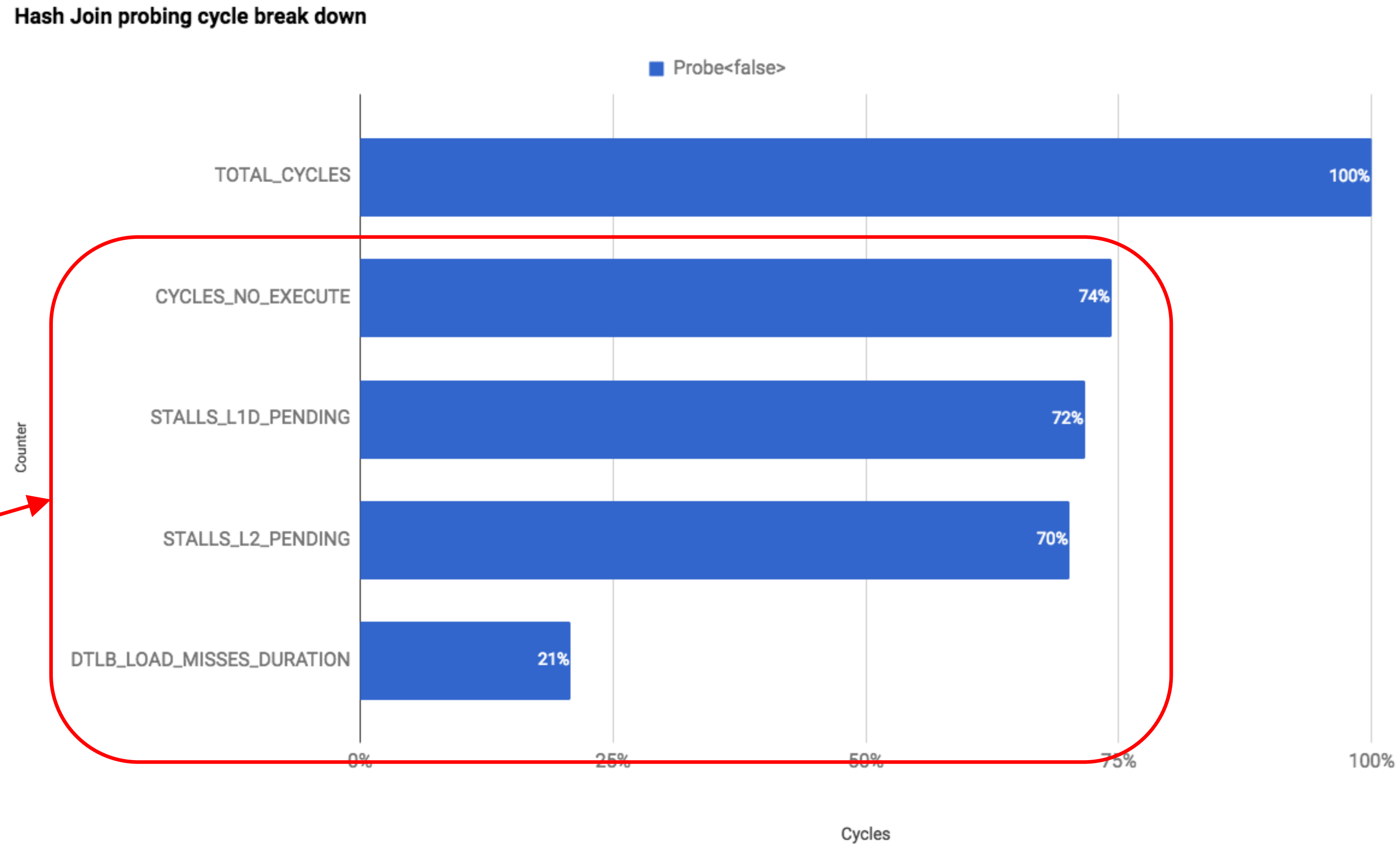




cloudera

Joins and aggregations under the hood





Most of the time is spent in L1 & L2 cache misses!

Joins and aggregations under the hood

Old hash table probe pseudo code

```
int hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
   // Read probe key from input batch
   probe_value =probe_batch_->GetRow(i);

   // Compute hash value
   hash_value =Hash(probe_value);

   // Find the matching partition in the hash table
   partition_id = hash_value >> (32 - NUM_PARTITIONING_BITS);

   iterator = hash_tbls_[partition_id]->FindProbeRow(
                                                  ht_ctx, hash_value,
probe_value);
}
```

Read probe join key from input batch

Joins and aggregations under the hood

Old hash table probe pseudo code

```
int hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
   // Read probe key from input batch
   probe_value =probe_batch_->GetRow(i);

   // Compute hash value
   hash_value =Hash(probe_value);

   // Find the matching partition in the hash table
   partition_id = hash_value >> (32 - NUM_PARTITIONING_BITS);

   iterator = hash_tbls_[partition_id]->FindProbeRow(
                                              ht_ctx, hash_value,
probe_value);
}
```

Compute hash value

cloudera

Joins and aggregations under the hood

Old hash table probe pseudo code

```
int hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
   // Read probe key from input batch
   probe_value =probe_batch_->GetRow(i);

   // Compute hash value
   hash_value =Hash(probe_value);

   // Find the matching partition in the hash table
   partition_id = hash_value >> (32 - NUM_PARTITIONING_BITS);

   iterator = hash_tbls_[partition_id]->FindProbeRow(
                                                  ht_ctx, hash_value,
probe_value);
}
```

Find the partition ID

Joins and aggregations under the hood

Old hash table probe pseudo code

```
int hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
   // Read probe key from input batch
   probe_value =probe_batch_->GetRow(i);

   // Compute hash value
   hash_value =Hash(probe_value);

   // Find the matching partition in the hash table
   partition_id = hash_value >> (32 - NUM_PARTITIONING_BITS);

   iterator = hash_tbls_[partition_id]->FindProbeRow(
                                             ht_ctx, hash_value,
probe_value);
}
```

Probe hash table

Joins and aggregations under the hood

Old hash table probe pseudo code

```
int hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
  // Read probe key from input batch
  probe_value =probe_batch_->GetRow(i);

  // Compute hash value
  hash_value =Hash(probe_value);

  // Find the matching partition in the hash table
  partition_id = hash_value >> (32 - NUM_PARTITIONING_BITS);

  iterator = hash_tbls_[partition_id]->FindProbeRow(
                              ht_ctx, hash_value,
probe_value);
}
```

Data dependency resulting in poor pipelining

Probe hash table

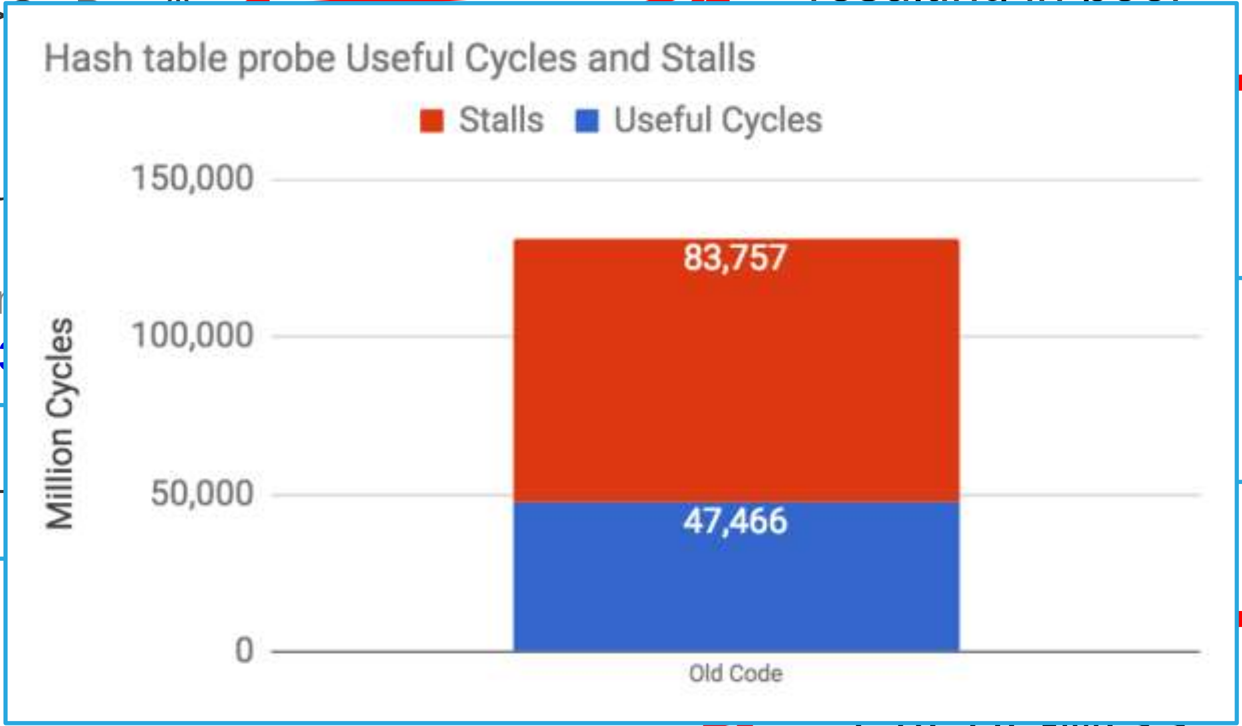
CACHE-MISS

Joins and aggregations under the hood

Old hash table probe pseudo code

```
int hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
   // Read probe key from input batch
   probe_value =probe_batch_->

   // Compute hash value
   hash_value =Hash(probe_valu

   // Find the matching partition i
   partition_id = hash_value >> (

   iterator = hash_tbls_[partition_
probe_value);
}
```

Data dependency
resulting in poor



Joins and aggregations in Impala : Current state

Old hash table probe pseudo code

```
int hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
   // Read probe key from input batch
   probe_value =probe_batch_->GetRow(i);

   // Compute hash value
   hash_value =Hash(probe_value);

   // Find the matching partition in the hash table
   partition_id = hash_value >> (32 - NUM_PARTITIONING_BITS);
   iterator = hash_tbls_[partition_id]->FindProbeRow(
                                                 ht_ctx, hash_value,
probe_value);
}
```

Current hash tableJoins probe pseudo code

```
int[] hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
   // Read probe key from input batch
   probe_values[i] =probe_batch_->GetRow(i);

   // Compute hash value
   hash_value[i] =Hash(pro

   // Find the matching partition in the hash table
   partition_id[i] = hash_value >> (32 - NUM_PARTITIONING_BITS);

   // Prefetch the hash value into LLC
   hash_tbls_[partition_ids[i]]->Prefetch(hash_values[i]);
}

for (int i =0; i < probe_batch_size; i++) {
   // Probe the hash table
   iterator = hash_tbls_[partition_id]->FindProbeRow(
                                                 ht_ctx, hash_value,
probe_value);
```

Prefetch the hash value

cloudera

Joins and aggregations in Impala : Current state

Old hash table probe pseudo code

```
int hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
  // Rea
  prob

  // Co
  hash

  // Fin
  parti                                    ONING_BITS);
  iterat                                   w(
                                 ht_ctx, hash_value
probe_value);
}
```

Breakup the loop into two parts, to reduce data dependency

Batch size is 1024 rows, giving enough time to ensure prefetched data is in LLC

Current hash tableJoins probe pseudo code

```
int[] hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
  // Read probe key from input batch
  probe_values[i] =probe_batch_->GetRow(i);

  // Compute hash value
  hash_value[i] =Hash(pro

  // Find the matching partition in the hash table
  partition_id[i] = hash_value >> (32 - NUM_PARTITIONING_BITS);

  // Prefetch the hash value into LLC
  hash_tbls_[partition_ids[i]]->Prefetch(hash_values[i]);
}

for (int i =0; i < probe_batch_size; i++) {
  // Probe the hash table
  iterator = hash_tbls_[partition_id]->FindProbeRow(
                                                    ht_ctx, hash_value,
probe_value);
```

Prefetch the hash value

cloudera

Joins and aggregations in Impala : Current state

Old hash table probe pseudo code

```
int hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
  // Rea
  prob

  // Co
  hash

  // Fi
  parti                              ONING_BITS);
  iterat                             w(
                         ht_ctx, hash_value
probe_value);
}
```

Breakup the loop into two parts, to reduce data dependency

Batch size is 1024 rows, giving enough time to ensure prefetched data is in LLC

CACHE-HIT

Current hash tableJoins probe pseudo code

```
int[] hash_value, probe_value, partition_id;
for (int i =0; i < probe_batch_size; i++) {
  // Read probe key from input batch
  probe_values[i] =probe_batch_->GetRow(i);

  // Compute hash value
  hash_value[i] =Hash(pro

  // Find the matching partition in the hash table
  partition_id[i] = hash_value >> (32 - NUM_PARTITIONING_BITS);

  // Prefetch the hash value into LLC
  hash_tbls_[partition_ids[i]]->Prefetch(hash_values[i]);
}

for (int i =0; i < probe_batch_size; i++) {
  // Probe the hash table
  iterator = hash_tbls_[partition_id]->FindProbeRow(
                                    ht_ctx, hash_value,
probe_value);
```

Prefetch the hash value

Joins and aggregations in Impala : Current state

Old hash table probe pseudo code

int hash_value, pr

for (int i =0; i < pr

// Read probe k

probe_value =p

// Compute has

hash_value =Ha

// Find the matc

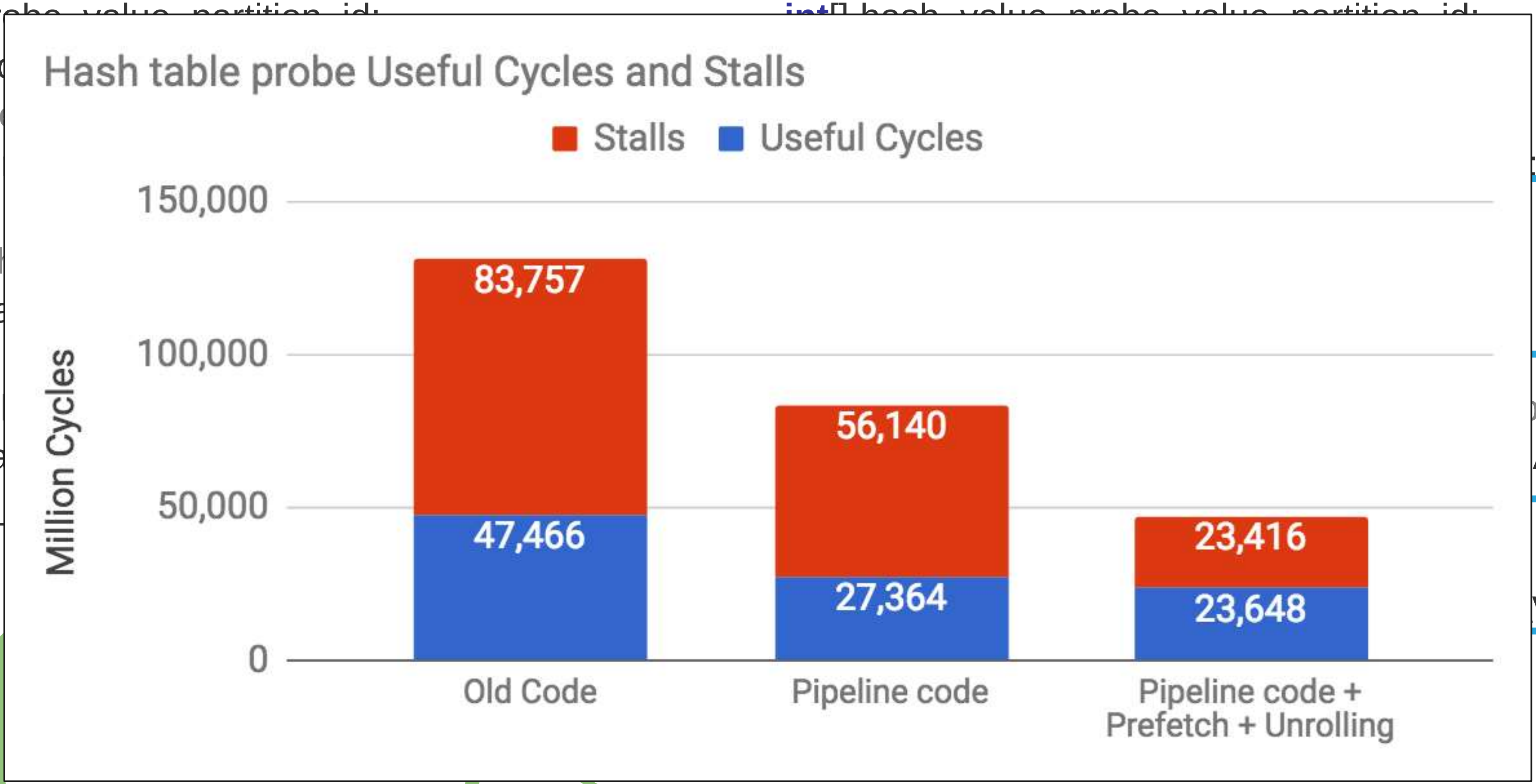
partition_id = ha

iterator = hash_

probe_value);

}

Current hash tableJoins probe pseudo code



hash

ble

ARTITIONING_BITS);

values[i]);

// Probe the hash table

iterator = hash_tbls_[partition_id]->FindProbeRow(
ht_ctx, hash_value,
probe_value);

cloudera

Joins and aggregations in Impala : Current state

Lessons learnt

- Don’t underestimate impact of cache-misses
- Prefetching is useful when done **<u>before</u>** data needs to be read from Memory
    - 30-40% speedup for Join and aggregation operations
- Removing data dependencies opens opportunities for improvement

Caveats

- TLB misses is still an issue
- Prefetching is as good as the hash function used, won’t handle chaining

cloudera

Impala Roadmap Focus

- Performance & Scalability
  - Parquet scanner performance
  - Metadata
  - RPC Layer
- Reliability
  - Node decommission
  - Better resource management
- Cloud

Thank you

https://github.com/apache/impala

http://impala.apache.org/

[Impala: A Modern, Open-Source SQL Engine for Hadoop]

cloudera