Informational Referential Integrity Constraints Support in Apache Spark

Ioana Delaney, Suresh Thalamati
Spark Technology Center, IBM

About the Speakers

- Ioana Delaney
  - Spark Technology Center, IBM
  - DB2 Optimizer developer working in the areas of query semantics, rewrite, and optimizer.
  - Worked on various releases of DB2 LUW and DB2 with BLU Acceleration
  - Apache Spark SQL Contributor
- Suresh Thalamati
  - Spark Technology Center, IBM
  - Apache Derby Committer and PMC Member, Apache Spark Contributor
  - Worked on various releases of IBM BigInsights, Apache Derby, Informix Database.

IBM Spark Technology Center

- Founded in 2015
- Location: 505 Howard St., San Francisco
- Web: http://spark.tc
- Twitter: @apachespark_tc
- Mission:
  - Contribute intellectual and technical capital to the Apache Spark community.
  - Make the core technology enterprise and cloud-ready.
  - Build data science skills to drive intelligence into business applications http://bigdatauniversity.com

SPARK SUMMIT 2017

Motivation

- Open up an area of query optimization techniques that rely on *referential integrity* (RI) constraints semantics
- Support for *informational primary key* and *foreign key (referential integrity)* constraints
- Not enforced by the Spark SQL engine; rather used by Catalyst to optimize the query processing
- Targeted to applications that load and analyze data that originated from a *Data Warehouse* for which the conditions for a given constraint are known to be true
- Improvement of up to 8x for some of the TPC-DS queries

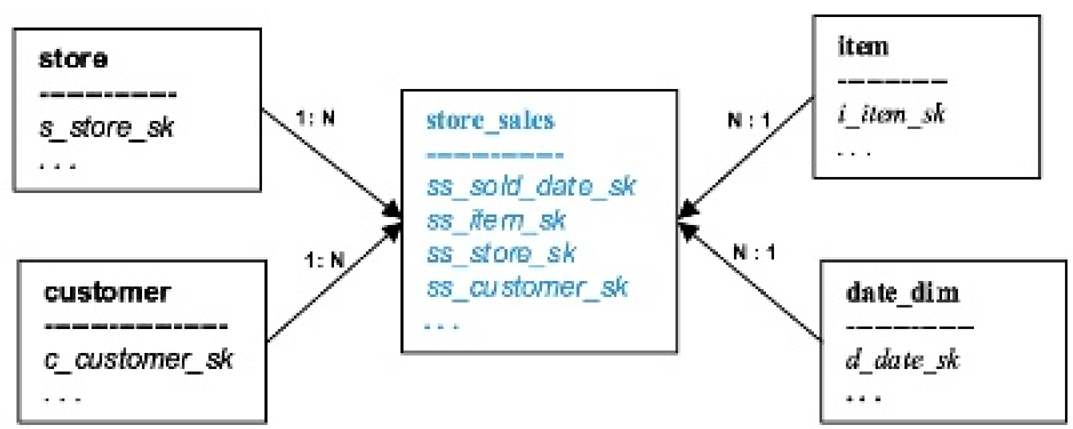
What is Data Warehouse?

- Relational database that integrates data from multiple heterogeneous sources e.g. transactional data, files, other sources
- Designed for data modelling and analysis
- Provides information around a subject of a business e.g. product, customers, suppliers, etc.
- The most important requirements are query performance and data simplicity
- Based on a dimensional, or Star Schema model
  - Consists of a *fact table* referencing a number of *dimension tables*
  - *Fact table* contains the main data, or measurements, of a business
  - *Dimension tables*, usually smaller tables, describe the different characteristics, or dimensions, of a business

TPC-DS Benchmark

- Proxy of a real organization data warehouse
- De-facto industry standard benchmark for measuring the performance of decision support solutions such as RDBMS and Hadoop/Spark based systems
- The underlying business model is a retail product supplier e.g. retail sales, web, catalog data, inventory, demographics, etc
- Examines large volumes of data e.g. 1TB to 100TB
- Executes SQL queries of various operational requirements and complexities e.g. ad-hoc, reporting, data mining

Excerpt from *store_sales fact table* diagram:



Integrity Constraints in Data Warehouse

- Typical constraints:
  - Unique
  - Not Null
  - Primary key and foreign key (referential integrity)
- Used for:
  - Data cleanliness
  - Query optimizations
- Constraint states:
  - Enforced
  - Validated
  - Informational

How do Optimizers use RI Constraints?

- Implement powerful optimizations based on RI semantics e.g. **Join Elimination**
- Example using a typical user scenario: *queries against views*

User view:

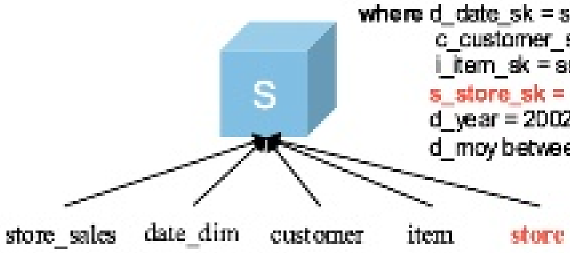
```
create view customer_purchases_2002 (id, last, first, product, store_id, month, quantity) as
select c_customer_id, c_last_name, c_first_name, i_product_name, s_store_id, d_moy, ss_quantity
from store_sales, date_dim, customer, item, store
where d_date_sk = ss_sold_date_sk and
        c_customer_sk = ss_customer_sk and
        i_item_sk = ss_item_sk and
        s_store_sk = ss_store_sk and
        d_year = 2002
```

User query:

Selects only a subset of columns from view

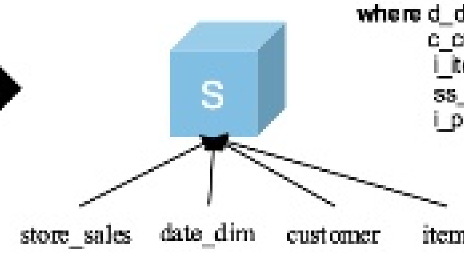
```
select id, first, last, product, quantity
from customer_purchases_2002
where product like 'bicycle%' and
month between 1 and 2
```

Internal optimizer query processing:




```
select c_customer_id as id, c_last_name as last, c_first_name as first,
        i_product_name as product, ss_quantity as quantity
from store_sales, date_dim, customer, item, store
where d_date_sk = ss_sold_date_sk and
        c_customer_sk = ss_customer_sk and
        i_item_sk = ss_item_sk and
        s_store_sk = ss_store_sk and
        d_year = 2002 and i_product like 'bicycle%' and
        d_moy between 1 and 2
```

Join between *store* and *store_sales* removed based on RI analysis




```
select c_customer_id as id, c_last_name as last, c_first_name as first,
        i_product_name as product, ss_quantity as quantity
from store_sales, date_dim, customer, item
where d_date_sk = ss_sold_date_sk and
        c_customer_sk = ss_customer_sk and
        i_item_sk = ss_item_sk and
        ss_store_sk is not null and d_year = 2002 and
        i_product like 'bicycle%' and d_moy between 1 and 2
```

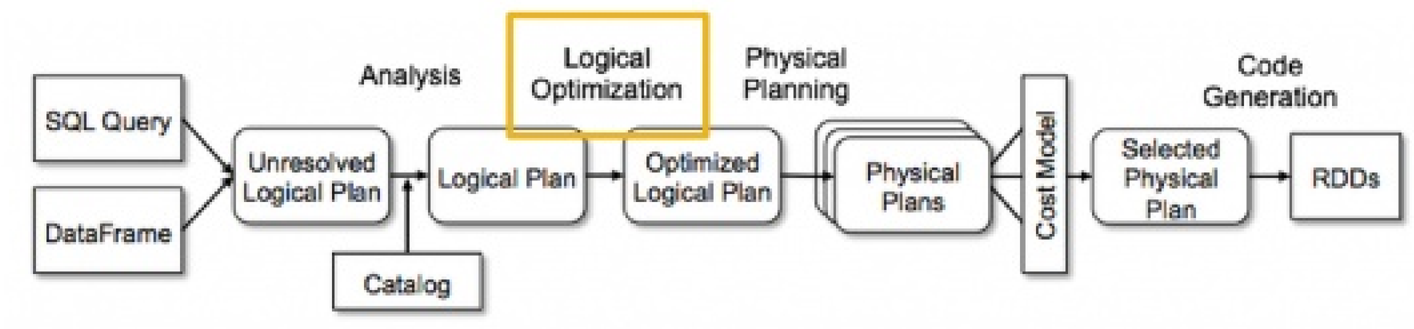
SPARK SUMMIT 2017

Let's make Spark do this too!

- Introduce query optimization techniques in Spark that rely on RI semantics
  - Star schema detection/Star-join optimizations
  - Existential subquery to Inner join transformation
  - Group by push down through joins
  - Many others

About Catalyst

- Apache Spark's Optimizer



- Queries are represented internally by *trees* of operators e.g. *logical trees* and *physical trees*
- Trees are manipulated by *rules*
- Each compiler phase applies a different set of rules
- For example, in the **Logical Plan Optimization** phase:
  - Rules rewrite logical plans into semantically equivalent ones for better performance
  - Rules perform natural heuristics:
    - e.g. merge query blocks, push down predicates, remove unreferenced columns, etc.
  - Earlier rules enable later rules
    - e.g. merge query blocks → global join reorder

Star Schema Optimizations

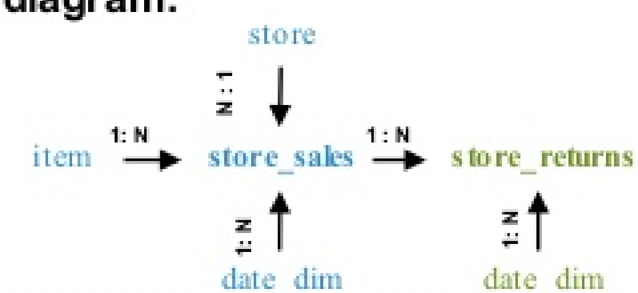
- Queries against star schema are expected to run fast based on RI relationships among the tables
- In a query, *star schema detection* algorithm:
  - Observes RI relationships based on the join predicates
  - Finds the tables connected in a star-join
  - Lets the Optimizer plan the star-join tables in an optimal way
- SPARK-17791 implements star schema detection based on heuristics
- Instead, use RI information to make the algorithm more robust

Star Schema Optimizations

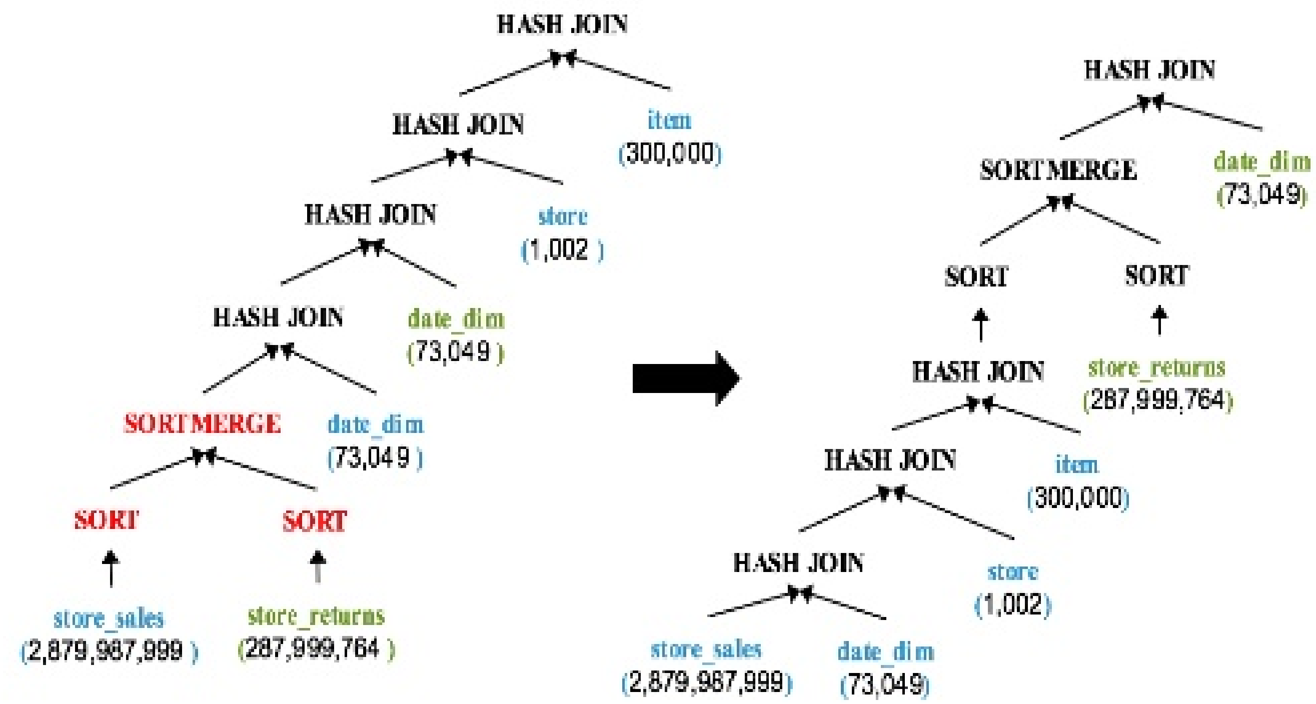
Simplified TPC-DS Query 25

```
select i_item_id, s_store_id, avg(ss_net_profit) as store_sales_profit,
       avg(sr_net_loss) as store_returns_loss
from
  store_sales, store_returns, date_dim d1, date_dim d2, store, item
where
  i_item_sk = ss_item_sk and
  s_store_sk = ss_store_sk and
  ss_customer_sk = sr_customer_sk and
  ss_item_sk = sr_item_sk and
  ss_ticket_number = sr_ticket_number and
  sr_returned_date_sk = d2.d_date_sk and
  d1.d_moy = 4 and  d1.d_year = 1998 and . . .
group by i_item_id, s_store_id
order by i_item_id, s_store_id
```

Star schema diagram:



Execution plan transformation:



- Query execution drops from 421 secs to 147 secs (1TB TPC-DS setup), ~ 3x improvement

Star Schema Optimizations

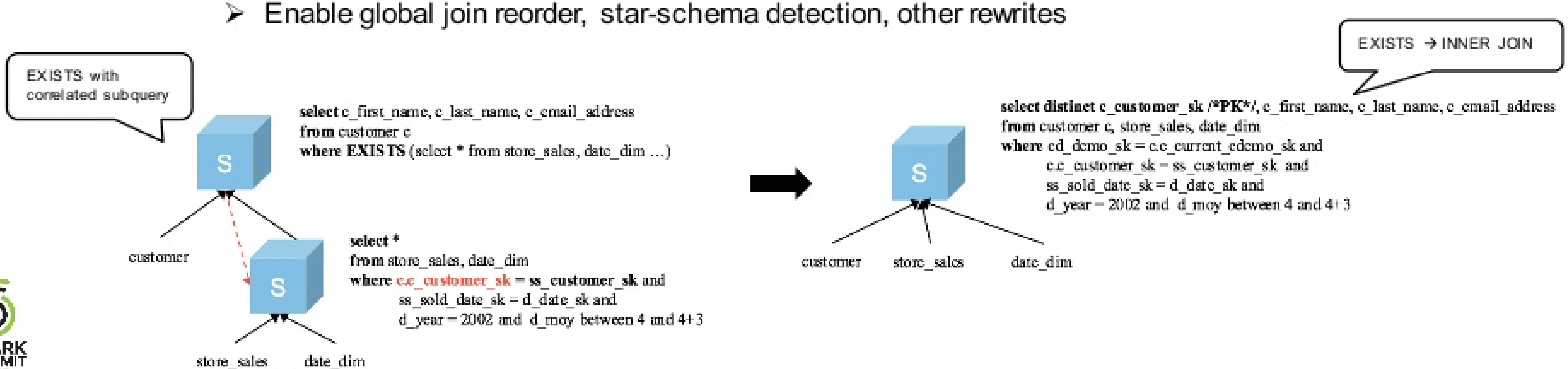
- TPC-DS query speedup: 2x – 8x
- By observing RI relationships among the tables, Optimizer makes better planning decisions
- Reduce the data early in the execution plan
- Reduce, or eliminate Sort Merge joins in favor of more efficient Broadcast Hash joins

TPC-DS 1TB performance results with star schema detection:

TPC-DS Query	spark-2.2 (secs)	spark-2.2 w/ starschema (secs)	Query Speedup
Q06	106	19	5x
Q13	296	98	3x
Q15	147	17	8x
Q17	398	146	2x
Q24	485	249	2x
Q25	421	147	2x
Q29	380	126	3x
Q45	93	17	5x
Q74	237	119	2x
Q85	104	42	2x

Existential Subquery to Inner Join

- Applied to certain types of EXISTS/IN subqueries
- Spark uses **Left Semi-join**
  - Returns a row of the outer if there is at least one match in the inner
  - Imposes a certain order of join execution
- Instead, use more flexible **Inner joins**
  - If the subquery produces at most one row, or by introducing a *Distinct* on the outer table's *primary key* to remove the duplicate rows after the join
  - Allows subquery tables to be merged into the outer query block
    - ➢ Enable global join reorder, star-schema detection, other rewrites



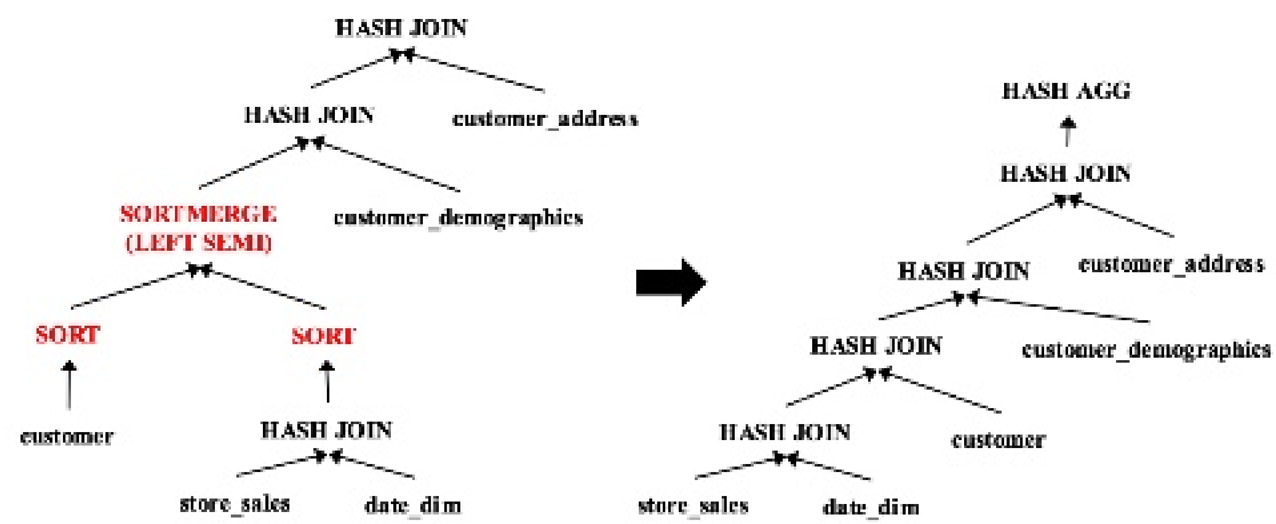
SPARK SUMMIT 2017

Existential Subquery to Inner Join

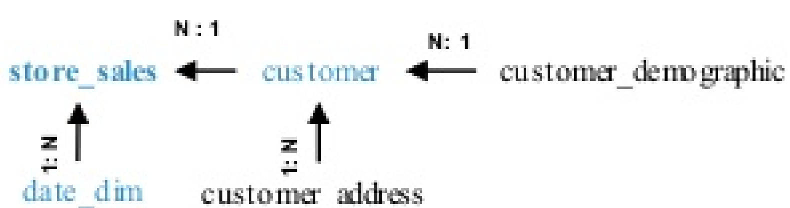
Simplified TPC-DS query 10

```
select cd_gender, cd_marital_status, cd_education_status, count(*) cnt1, ...
from  customer c, customer_address ca, customer_demographics
where
  c.c_current_addr_sk = ca.ca_address_sk and
  ca_county in ('Woods County','Madison County') and
  cd_demo_sk = c.c_current_cdemo_sk and
  EXISTS (select *
            from store_sales, date_dim
            where c.c_customer_sk = ss_customer_sk and
                  ss_sold_date_sk = d_date_sk and
                  d_year = 2002 and d_moy between 4 and 4+3)
group by cd_gender, cd_marital_status, cd_education_status, ...
order by cd_gender, cd_marital_status, cd_education_status, ...
limit 100
```

Execution plan transformation:



Star schema diagram:



- Query execution drops from 167 secs to 22 secs (1TB TPC-DS setup), 7x improvement

Existential Subquery to Inner Join

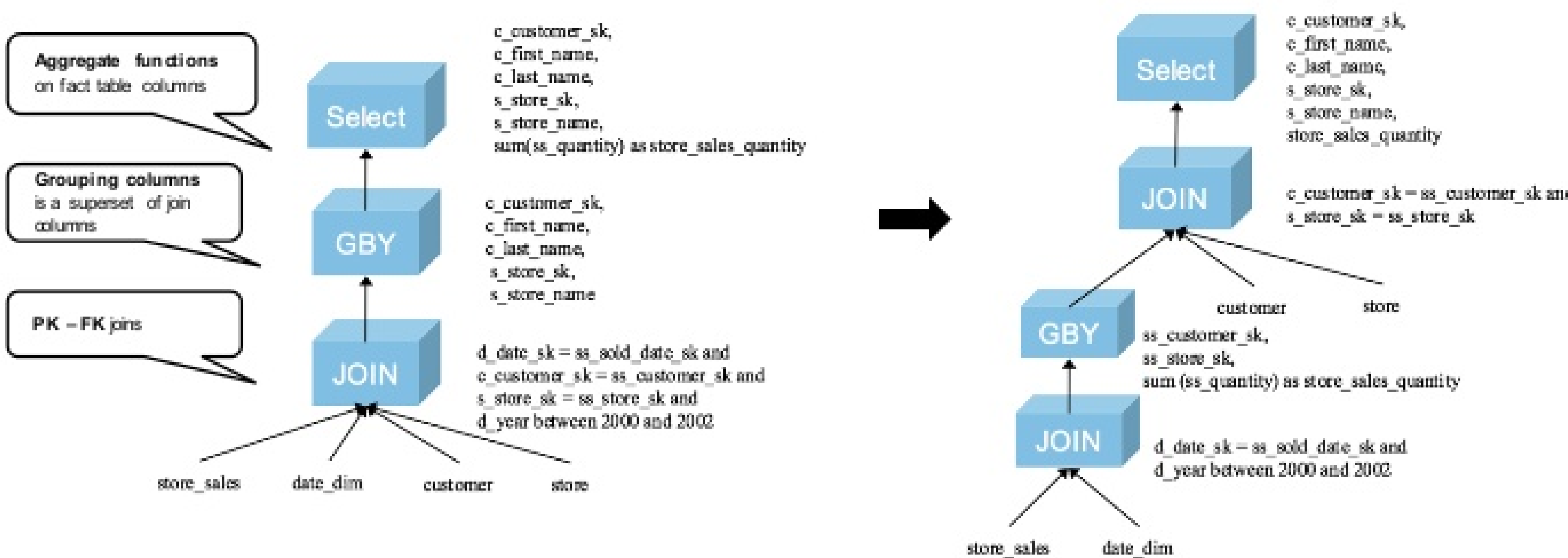
- Inner joins provide better alternative plan choices for the Optimizer
  - Avoids joins with large inners, and thus favors Broadcast Hash Joins
- Tables in the subquery are merged into the outer query block
  - Enables other rewrites
  - Benefits from star schema detection
- May introduce a Distinct/HashAggregate

TPC-DS 1TB performance results with subquery to join:

TPC-DS Query	spark-2.2 (secs)	spark-2.2 w/ sub2join (secs)	Query speedup
Q10	355	190	2x
Q16	1394	706	2x
Q35	462	285	1.5x
Q69	327	173	1.8x
Q94	603	307	2x

Group By Push Down Through Join

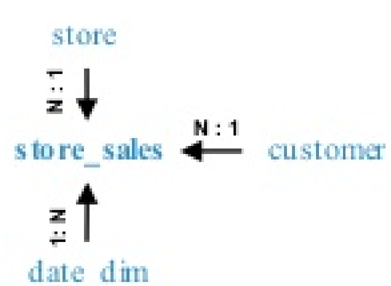
- In general, applied based on the cost/selectivity estimates
- If the join is an *RI join*, heuristically push down Group By to the fact table
  - The input to the Group By remains the same before and after the join
  - The input to the join is reduced
  - Overall reduction of the execution time



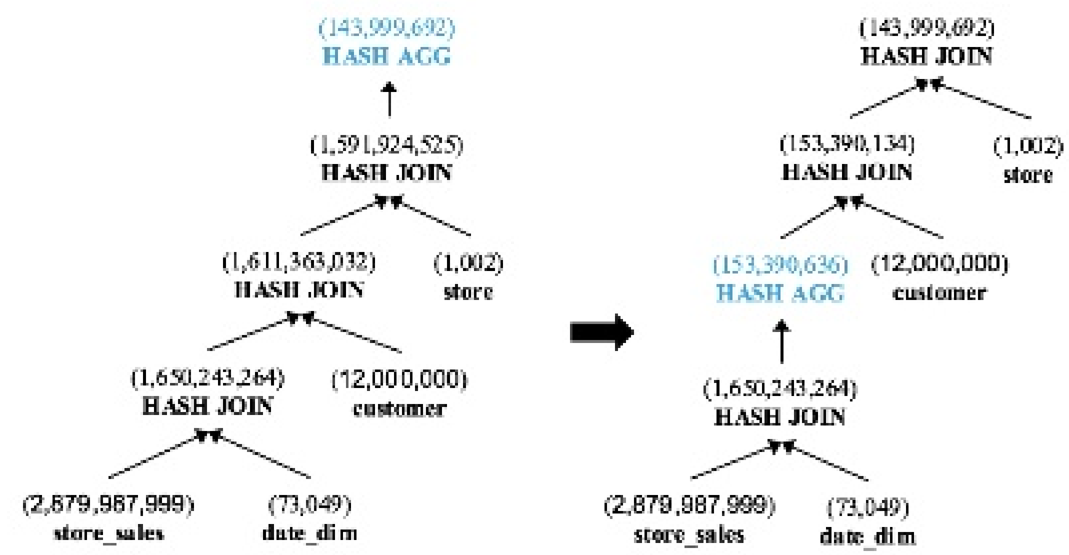
Group By Push Down Through Join

```
select c_customer_sk, c_first_name, c_last_name, s_store_sk, s_store_name,
       sum(ss.ss_quantity) as store_sales_quantity
from store_sales ss, date_dim, customer, store
where d_date_sk = ss_sold_date_sk and
       c_customer_sk = ss_customer_sk and
       s_store_sk = ss_store_sk and
      d_year between 2000 and 2002
group by c_customer_sk, c_first_name, c_last_name, s_store_sk, s_store_name
order by c_customer_sk, c_first_name, c_last_name, s_store_sk, s_store_name
limit 100;
```

Star schema:



Execution plan transformation:



- Query execution drops from 70 secs to 30 secs (1TB TPC-DS setup), 2x improvement

More Optimizations

- RI join elimination
  - Eliminates dimension tables if none of their columns, other than the PK columns, are referenced in the query
- Redundant join elimination
  - Eliminates self-joins on unique keys; self-joins may be introduced after view expansion
- Distinct elimination
  - Distinct can be removed if it is proved that the operation produces unique output
- Proving *maximal* cardinality
  - Used by query rewrite

Informational RI Implementation in Spark

- DDL Support for constraint lifecycle
- Metadata Storage
- Constraint Maintenance

DDL Support

- Support Informational Primary Key and Foreign Key constraint
- New DDLs for create, alter, drop constraint
- Create Constraint:
  - Constraints can be added as part of CREATE TABLE and ALTER TABLE
  - Create table DDL supports both inline and out_of_line syntax

CREATE TABLE *customer* (id int **CONSTRAINT** *pk1* **PRIMARY KEY RELY**,
name string, address string, demo int)

CREATE TABLE *customer_demographics* (id int, gender string, credit_rating string,
CONSTRAINT *pk1* **PRIMARY KEY** (*id*) **RELY**);

ALTER TABLE *customer*
ADD CONSTRAINT *fk1* **FOREIGN KEY** (*demo*) **REFERENCES** *customer_demographics* (*id*) **VALIDATE RELY**;

- Syntax is similar to Hive 2.1.0 DDL

Constraint States and Options

- Two states: VALIDATE and NOVALIDATE
  - Specify if the existing data conforms to the constraint
- Two options: RELY and NORELY
  - Specify whether a constraint is to be taken into account for query rewrite
- Used by Catalyst as follows:
  - NORELY: constraint cannot be used by Catalyst
  - NOVALIDATE RELY: constraint used for join order optimizations
  - VALIDATE RELY: constraint used for query rewrite and join order optimizations
- Constraints are not enforced, they are informational only.
- Namespace for a constraint is at the table level

Metadata Storage

- Constraint definitions are stored in the table properties
- Constraint information is stored in JSON format

```
spark.sql.constraint={"id":"fk1","type":"fk","keyCols"["demo"],"referenceTable":"customer_demographics",
                     "referenceCols":["id"],"rely":true,"validate":true}
```

- Describe formatted includes constraint information

```
# Constraint Information
Primary Key:
  Constraint Name:    pk1
  Key Columns:        [id]

Foreign Keys:
  Constraint Name:    fk1
  Key Columns:        [demo]
  Reference Table:    customer_demographics
  Reference Columns:[id]
```

Constraint Maintenance

- Dropping a constraint

  ALTER TABLE *customer_demographics* **DROP** CONSTRAINT *pk1* **CASCADE**

  DROP TABLE IF EXISTS *customer_demographics* **CASCADE CONSTRAINT**

- Validating Constraints

  ALTER TABLE *customer* **VALIDATE CONSTRAINT** *pk1*

  – Internally Spark SQL queries are run to perform the validation.

- Data cache invalidation

  – Entries in the cache that reference the altered table are invalidated

1TB TPC-DS Performance Results

Cluster: 4-node cluster, each node having:
12 2 TB disks,
Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz, 128 GB RAM, 10 Gigabit Ethernet
Number of cores: 48
Apache Hadoop 2.7.3, Apache Spark 2.2 main (Mar 31, 2017)
Database info:
Schema: TPCDS
Scale factor: 1TB total space
Storage format: Parquet with Snappy compression

TPC-DS Query	spark-2.2 (secs)	spark-2.2-ri (secs)	Speedup
Q06	106	19	5x
Q10	355	190	2x
Q13	296	98	3x
Q15	147	17	8x
Q16	1394	706	2x
Q17	398	146	2x
Q24	485	249	2x
Q25	421	147	2x
Q29	380	126	3x
Q35	462	285	1.5x
Q45	93	17	5x
Q69	327	173	1.8x
Q74	237	119	2x
Q85	104	42	2x
Q94	603	307	2x

Call to Action

- Read the Spec in SPARK-19842
  https://issues.apache.org/jira/browse/SPARK-19842

- Prototype code is under internal review

- Watch out for the upcoming PRs

- Meet us at the IBM demo booth #407

Thank You.

Ioana Delaney ursu@us.ibm.com

Suresh Thalamati tsuresh@us.ibm.com

Visit http://spark.tc