Extending Spark SQL Data Sources APIs with Join Push Down

Ioana Delaney, Jia Li

Spark Technology Center, IBM

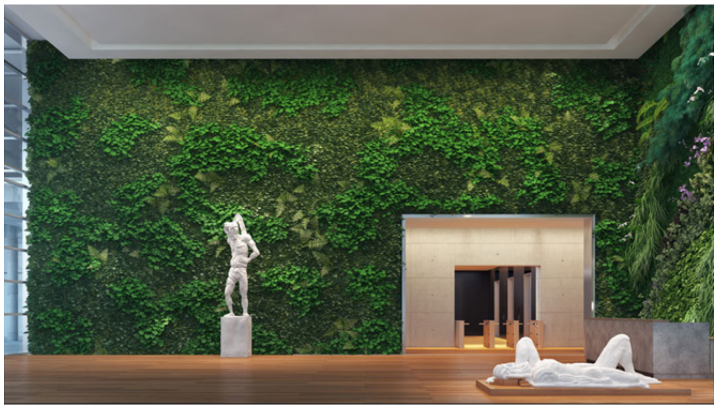
#EUdev7

SPARK SUMMIT EUROPE 2017

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
- Jia Li
  - Spark Technology Center, IBM
  - Apache Spark SQL Contributor
  - Worked on various releases of IBM BigInsights and IBM Optim Query Workload Tuner

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

Motivation

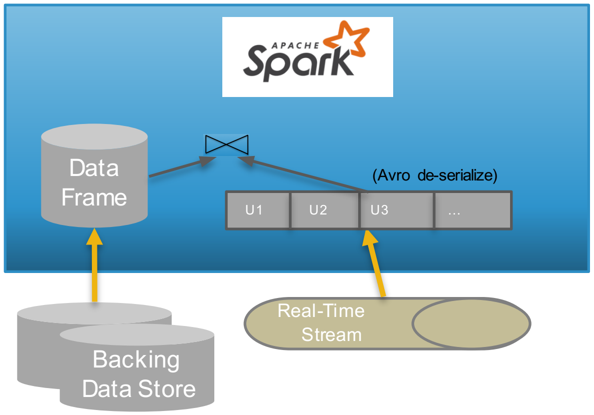
- Spark applications often directly query external data sources such as relational databases or files
- Spark provides *Data Sources APIs* for accessing structured data through Spark SQL
- Support optimizations such as *Filter push down* and *Column pruning* - subset of the functionality that can be pushed down to some data sources
- This work extends Data Sources APIs with *join push down*
- *Join push down* is nothing more than *Selection* and *Projection push down*
- Significantly improves query performance by reducing the amount of data transfer and exploiting the capabilities of the data sources such as index access

Data Sources APIs

- Mechanism for accessing external data sources
- Built-in libraries (Hive, Parquet, Json, JDBC, etc.) and external, third-party libraries through its *spark-packages*
- Users can read/save data through generic *load()/save()* functions, or *relational SQL tables* i.e. DataFrames with persistent metadata and names
- Developers can build new libraries for various data sources i.e. to create a new data source one needs to specify the schema and how to get the data
- Tightly integrated with Spark's Catalyst Optimizer by allowing optimizations to be pushed down to the data source

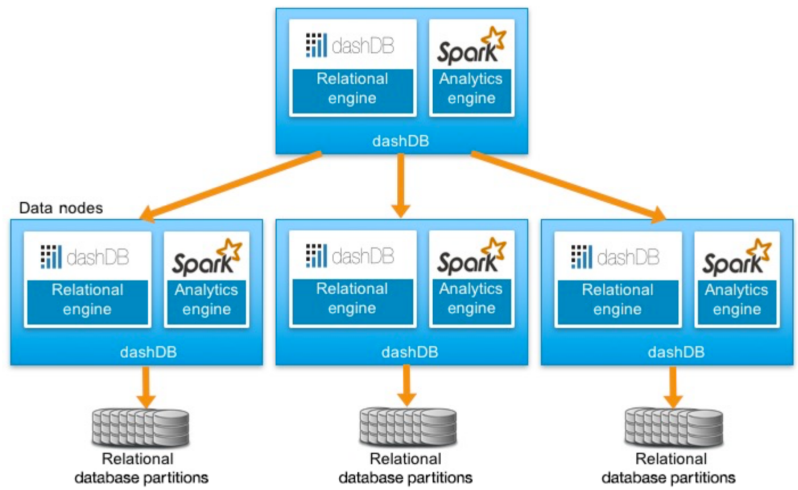
Use Case 1: Enterprise-wide data pipeline using Spark

- As data gets distributed from different data sources throughout an organization, there is a need to create a pipeline to connect these sources.
- Spark framework extracts the records from the backing store, caches the DataFrame results, and then performs joins with the updates coming from the stream
- e.g. [Spark at Bloomberg – Summit 2016]



Use Case 2: SQL Acceleration with Spark

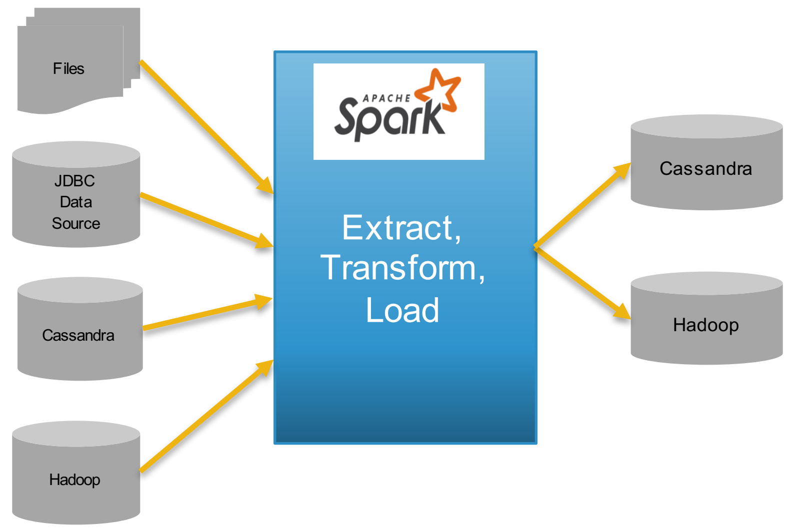
- e.g. RDBMS MPP cluster nodes are overlaid with local Spark executor processes
- The data frames in Spark are derived from the existing data partitions of the database cluster.
- The co-location of executors with the database engine processes minimizes the latency of accessing the data
- IBM's dashDB and other vendors provide SQL acceleration through co-location or other Spark integrated architectures



dashDB VERTICA SNAPPYDATA MAPD SAP HANA Pivotal Greenplum intel

Use Case 3: ETL from legacy data sources

- Common pattern in Big Data space is offloading relational databases and files into Hadoop and other data sources

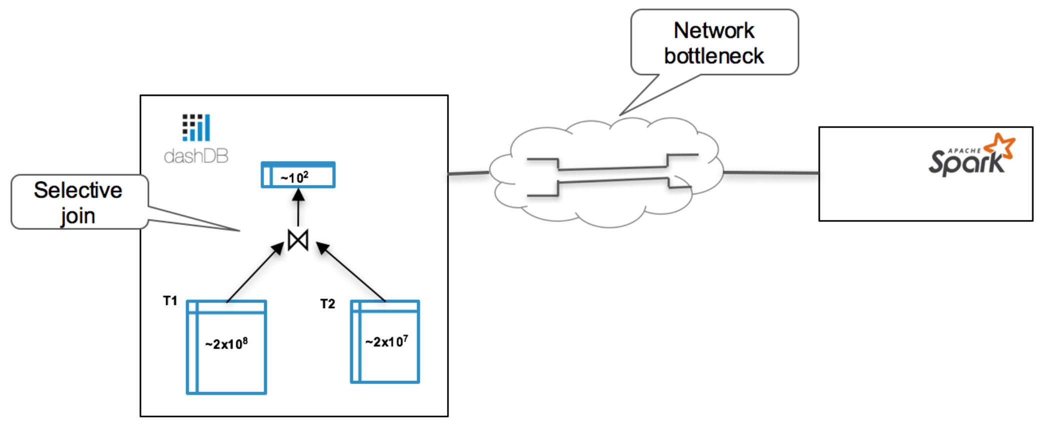


Spark as a Unified Analytics Platform for Data Federation

- Ingests data from disparate data sources and perform fast in-memory analytics on their combined data
- Supports a wide variety of structured and unstructured data sources
- Applies analytics everywhere without the need of data centralization

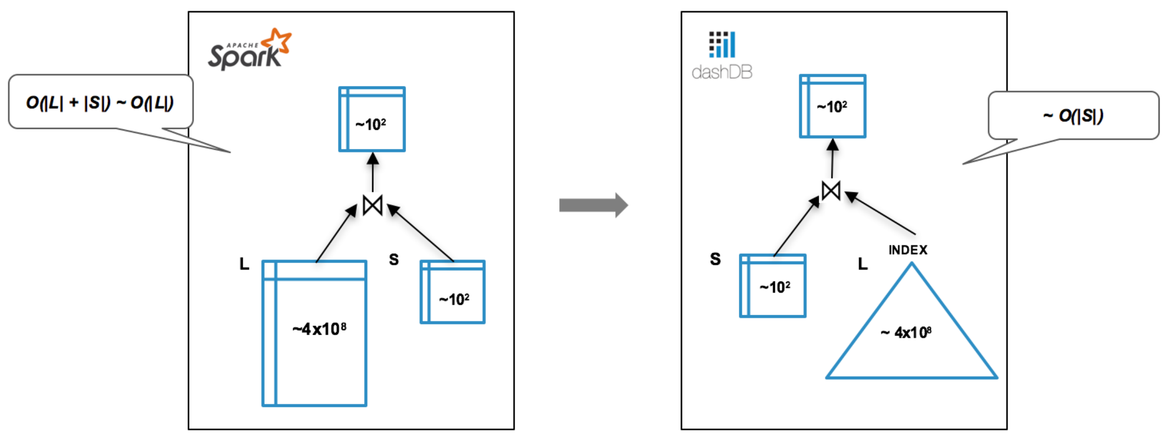
Challenges: Network speed

- e.g. large tables in DBMS and selective join in Spark
- Spark reads the entire table, applies some predicates at the data source, and executes the join locally
- Instead, push the join execution to the data source
  - ➢ Reduce the amount of data transfer
  - ➢ The query runs **40x** faster!



Challenges: Exploit data source capabilities

- e.g. DBMS has indexes
- Join execution in Spark ~ *O(|L|)*
- Join execution in DBMS ~ *O(|S|)*
- Instead, push the join execution to the data source
  - ➢ Efficient join execution using index access
  - ➢ Reduce the data transfer
  - ➢ The query runs **100x** faster!



Join push down to the data source

- Alternative execution strategy in Spark
- Nothing more than Selection and Projection push down
- May reduce the amount of data transfer
- Provides Spark with the functions of the underlying data source
- Small API changes with significant performance improvement

Push down based on cost vs. heuristics

- *Acceptable performance* is the most significant concern about data federation
- *Query Optimizer* determines the best execution of a SQL query e.g. implementation of relational operators, order of execution, etc.
- In a federated environment, the optimizer must also decide whether the different operations should be done by the federated server, e.g. Spark, or by the data source
  - Needs knowledge of what each data source can do (e.g. file system vs. RDBMS), and how much it costs (e.g. statistics from data source, network speed, etc.)
- Spark's Catalyst Optimizer uses a combination of heuristics and cost model
- Cost model is an evolving feature in Spark
- Until federated cost model is fully implemented, use safe heuristics

Minimize data transfer

Filtering joins (e.g. Star-joins)

```
select   s_store_id  ,s_store_name  ,sum(ss_net_profit)   as store_sales_profit
from  store_sales,   date_dim,  store
where  d_moy  = 4 and d_year  = 2001  and
          d_date_sk  = ss_sold_date_sk   and
          s_store_sk = ss_store_sk
group  by s_store_id  ,s_store_name
order  by s_store_id  ,s_store_name
limit  100
```

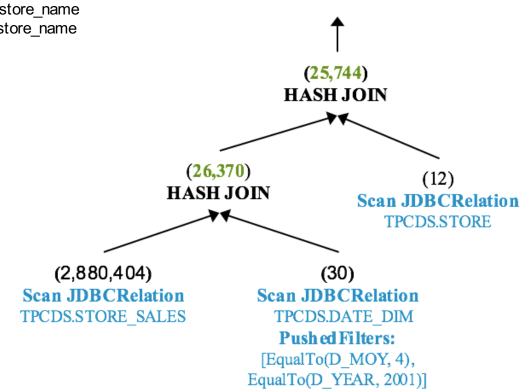
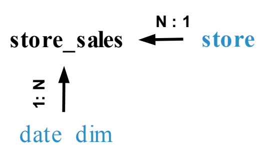


Table diagram:



Expanding joins

```
select   s_store_id  ,s_store_name,   sum(ss_net_profit)   as store_sales_profit,   sum(ss_net_loss)   as store_loss
from  store_sales,   date_dim,  store,  store_returns
where  d_moy  = 4 and d_year  = 2001  and
          d_date_sk  = ss_sold_date_sk   and
          s_store_sk = ss_store_sk   and
          ss_ticket_number  = sr_ticket_number   and
          ss_item_sk = sr_item_sk
group  by s_store_id  ,s_store_name
order  by s_store_id  ,s_store_name
limit  100
```

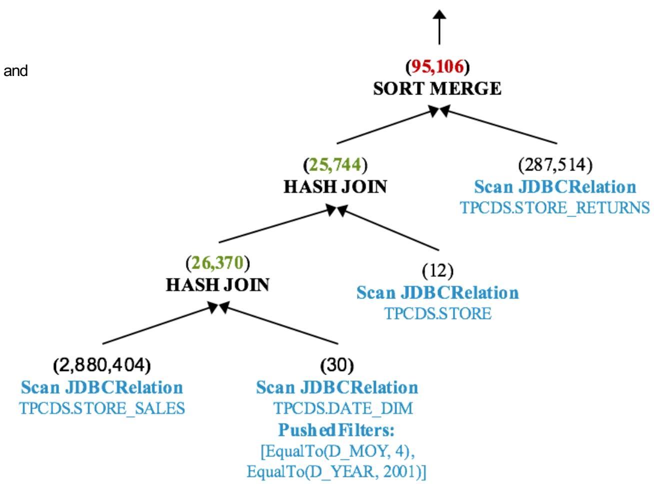
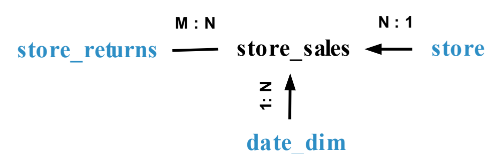
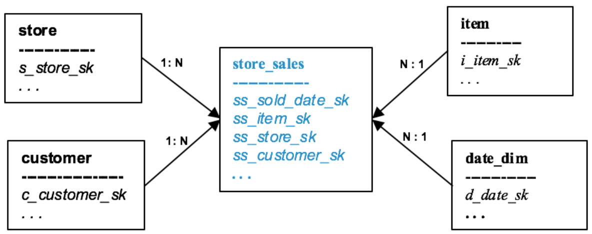


Table diagram:



Star-schema joins

- Joins among tables in a *star-schema* relationship
- *Star-schema* is the simplest form of a data warehouse schema
- *Star-schema* model consists of a *fact table* referencing a number of *dimension tables*
- Fact and dimension tables are in a *primary key – foreign key* relationship.

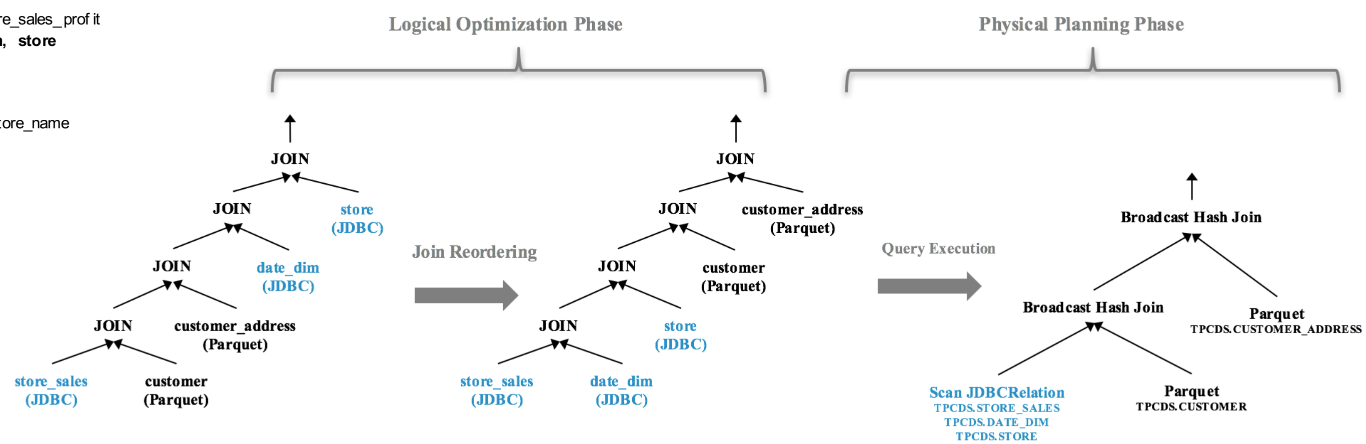


- Star joins are filters applied to the fact table
- Catalyst recognizes star-schema joins
- We use this information to detect and push down star joins to the data source

Join re-ordering based on data source

- Maximize the amount of functionality that can be pushed down to the data source
- Extends Catalyst's join enumeration rules to re-order joins based on data source
- Important alternative execution plan for global federated optimization

```
select   s_store_id  ,s_store_name  ,sum(ss_net_profit)  as store_sales_profit
from store_sales,  customer,  customer_address,  date_dim,  store
where  d_moy  = 4 and d_year  = 2001  and
         d_date_sk = ss_sold_date_sk   and
         sr_store_sk = ss_store_sk
group  by s_store_id  ,s_store_name   order  by s_store_id   ,s_store_name
limit 100
```

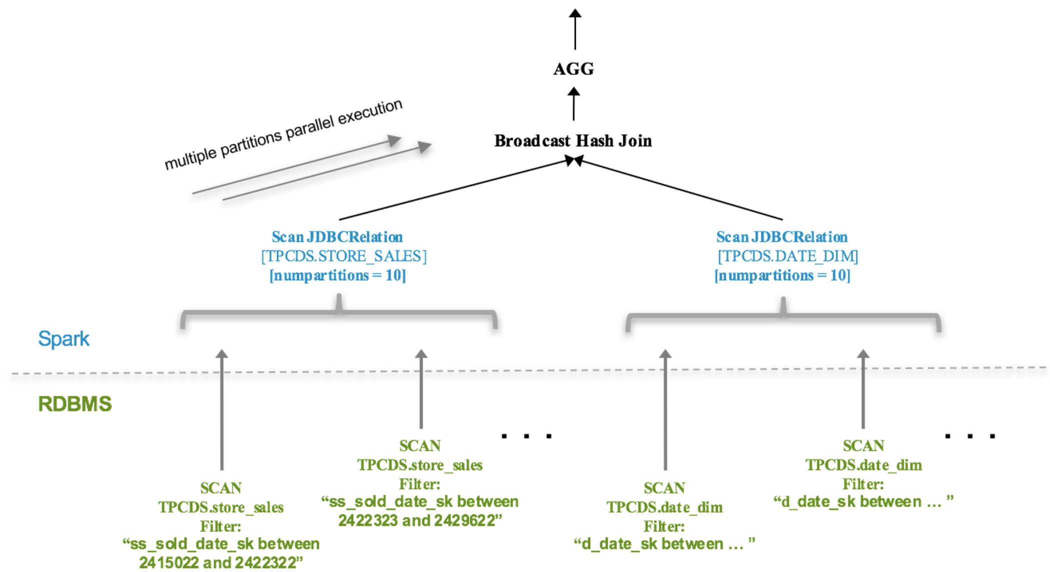


Managing parallelism: Single vs. multiple partitions join

- *Partitions* are the basic unit of parallelism in Spark
- JDBC options to specify data partitioning: *partitionColumn*, *lowerbound*, *upperbound*, and *numPartitions*
- Spark splits the table read across *numPartitions* tasks with a stride specified by *lowerbound* and *upperbound*

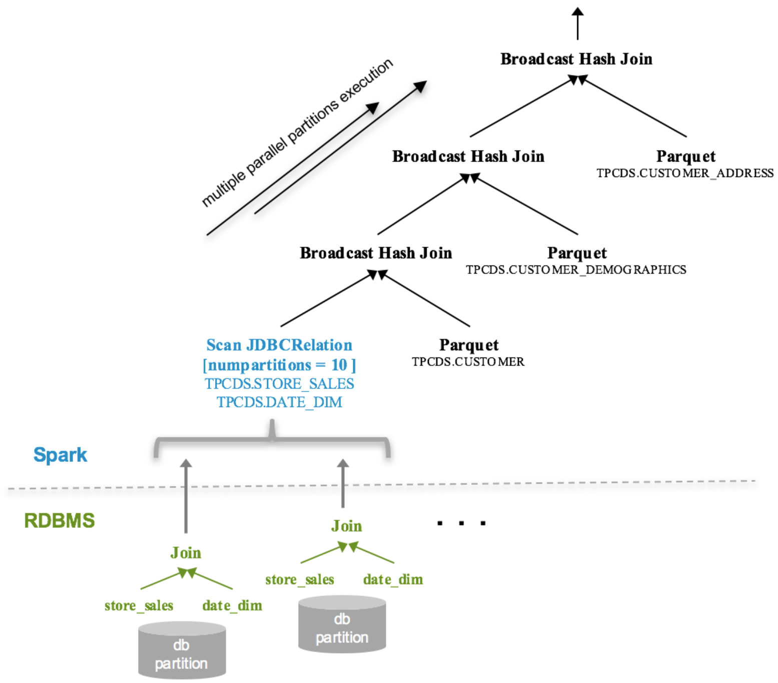
```
CREATE TABLE STORE_SALES USING org.apache.spark.sql.jdbc OPTIONS
        (. . .
         dbtable 'TPCDS.STORE_SALES',
         columnName="ss_sold_date_sk"
         lowerBound=2415022,
         upperBound=2488020
         numpartitions = 10
         . . .
```

```
select d_moy ,sum(ss_net_profit) as store_sales_profit
from store_sales, date_dim
where d_year = 2001 and
      d_date_sk = ss_sold_date_sk
group by d_moy
order by d_moy
```



How to partition when joins are pushed down?

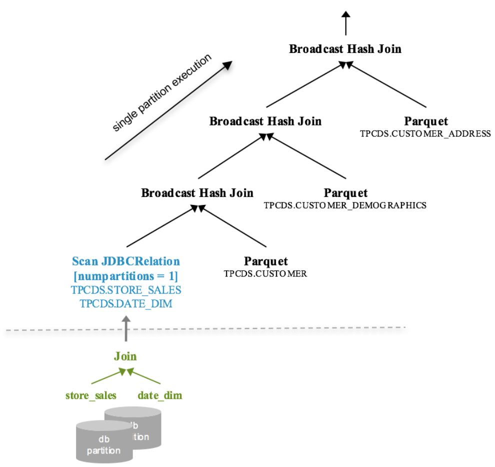
1) Push down partitioned, co-located joins to data source
   - A *co-located join* occurs locally on the database partition on which the data resides.
   - Partitioning in Spark aligns with the data source partitioning
   - Hard to achieve for multi-way joins
   - Cost based decision



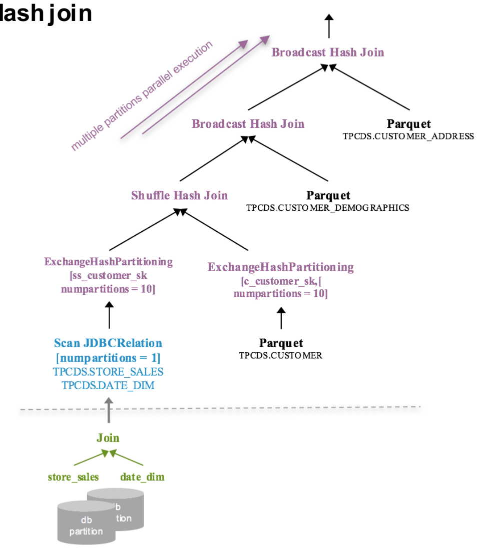
How to partition when joins are pushed down?

2) Perform partitioning in Spark i.e. choose a join method that favors parallelism
   - *Broadcast Hash Join* is the preferred method when one of the tables is small
   - If the large table comes from a single partition JDBC connection, the execution is serialized on that single partition
   - In such cases, *Shuffle Hash Join* and *Sort Merge Join* may outperform *Broadcast Hash Join*

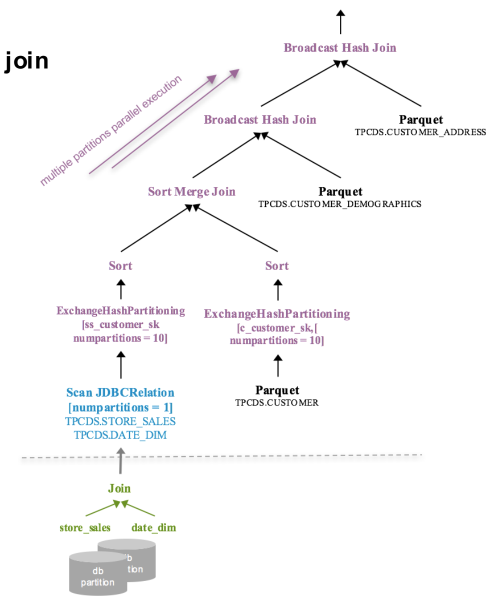
Broadcast Hash join to single partition



Shuffle Hash join



Sort Merge join



1 TB TPC-DS Performance Results

- Proxy of a real data warehouse
- Retail product supplier e.g. retail sales, web, catalog, etc.
- Ad-hoc, reporting, and data mining type of queries
- Mix of two data sources: IBM DB2/JDBC and Parquet

Cluster: 4-node cluster, each node having:
12 2 TB disks,
Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz, 128 GB RAM
Number of cores: 48
Apache Hadoop 2.7.3, Apache Spark 2.2 main (August, 2017)
Database info:
Schema: TPCDS
Scale factor: 1TB total space
Mix of Parquet and DB2/JDBC data sources
DB2 DPF info: 4-node cluster, each node having:
10 2 TB disks,
Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz, 128 GB RAM
Number of cores: 48

TPC-DS Query	spark-2.2 (mins)	spark-2.2-jpd (mins)	Speedup
Q8	32	4	8x
Q13	121	5	25x
Q15	4	2	2x
Q17	77	7	11x
Q19	42	4	11x
Q25	153	7	21x
Q29	81	7	11x
Q42	31	3	10x
Q45	14	3	4x
Q46	61	5	12x
Q48	155	5	31x
Q52	31	5	6x
Q55	31	4	8x
Q68	69	4	17x
Q74	47	23	2x
Q79	63	4	15x
Q85	22	2	11x
Q89	55	4	14x

Data Sources APIs for reading the data

- *BaseRelation:* The abstraction of a collection of tuples read from the data source. It provides the *schema* of the data.
- *TableScan:* Reads all the data in the data source.
- *PrunedScan:* Eliminates unneeded columns at the data source.
- *PrunedFilteredScan:* Applies predicates at the data source.
- ***PushDownScan:*** New API that applies complex operations such as joins at the data source.

PushDownScan APIs

- Trait for a *BaseRelation*
- Used with data sources that support complex functionality such as joins
- Extends *PrunedFilteredScan* with *DataSourceCapabilities*
- Methods needed to be overriden:
  - def *buildScan*(columns: Array[String],
                   filters: Array[Filter],
                   tables: Seq[BaseRelation]):RDD[Row]
  - def *getDataSource()*: String
- *DataSourceCapabilities* trait to model data source characteristics e.g. type of joins

Future work: Cost Model for Data Source APIs

- Transform Catalyst into a *global optimizer*
- *Global optimizer* generates an optimal execution plan across all data sources
- Determines where an operation should be evaluated based on:
  1. The cost to execute the operation.
  2. The cost to transfer data between Spark and the data sources
- Key factors that affect global optimization:
  – Remote table statistics (e.g. number of rows, number of distinct values in each column, etc)
  – Data source characteristics (e.g. CPU speed, I/O rate, network speed, etc.)
- Extend Data Source APIs with *data source characteristics*
- Retrieve/compute data source *table statistics*
- Integrate data source cost model into Catalyst

Thank You.

Ioana Delaney ursu@us.ibm.com
Jia Li jiali@us.ibm.com

Visit http://spark.tc

SPARK SUMMIT EUROPE 2017