Design Specification of Spark Cost-Based Optimization

Authors:

Ron Hu, Ph.D.
Huawei Technologies Co., Ltd.
2330 Central Expressway, E2-153
Santa Clara, CA 95050, USA
Email: ron.hu@huawei.com

Zhenhua Wang， Ph.D.
Huawei Technologies Co., Ltd.
410 Jiang Hong Road, Binjiang District,
Hangzhou, Zhejiang 310052, China
Email: wangzhenhua@huawei.com

Contents

1) Rationale

Spark SQL's Catalyst optimizer is mainly a rule executor. It has many rule based optimization techniques implemented. Most of the rules are heuristics-based and/or empirical based. For example, predicate pushdown is a good rule to reduce the number of the qualified records before a join operation is performed. And project pruning is a good rule to reduce the number of the participating columns before further processing. However, without detailed column statistics information on data distribution, it is difficult to accurately estimate the filter factor and cardinality, and thus output size of a database operator. With the inaccurate and/or misleading statistics, it often leads the optimizer to choose suboptimal query execution plans.

In order to improve the quality of query execution plans, we decided to enhance Spark SQL optimizer with detailed statistics information [Hu2016]. From the detailed statistics information, we can better estimate the number of output records and output size for each database operator. This way can help optimizer choose a better query plan.

As Huawei's CBO work is built on top of Spark SQL, we assume that a reader is familiar with Spark SQL. In this document, we focus on the changes to Spark SQL, specifically on the major entry points into Spark SQL's data structure and work flow. We describe our design in 6 major areas:

- Statistics structure
- Statistics collection
- System Catalog (or meta store)
- Cost Functions
- Cardinality estimation of predicate expressions
- Cardinality estimation of execution operators

Our CBO work has been pretty much broken down to align with the above major areas. We plan to contribute our code in multiple decoupled pull requests. It should be noted that we took a non-intrusive approach when adding CBO to Spark SQL. First, we defined a configuration parameter "spark.sql.cbo" for users to enable/disable this feature. As a Boolean field, "spark.sql.cbo" parameter can be set to either true or false. Currently, the default value is false.

2) Statistics Class

Before we discuss the major components, we need to describe the common key data structure holding statistics information. This data structure is referenced when we execute statistics collection SQL statement to save information into system catalog. This data structure is also referenced when we fetch statistics information from system catalog to optimize a query plan.

The `Statistics` class is the main data structure holding CBO statistics information. We extend this class by adding the following fields:

- `estimatedSize`: Total uncompressed output data size from a `LogicalPlan` node.
- `rowCount`: Number of output rows from a `LogicalPlan` node.
- `basicStats`: Basic column statistics, i.e., max, min, number of nulls, number of distinct values, max column length, average column length.
- `Histograms`: Histograms of columns, i.e., equi-width histogram (for numeric and string types) and equi-height histogram (only for numeric types).

```
case class Statistics(
   sizeInBytes: BigInt,
   isBroadcastable: Boolean = false,
   var estimatedSize: Option[BigInt] = None,
   var rowCount: Option[BigInt] = None,
   var basicStats: AttributeMap[BasicStat] = AttributeMap(Seq()),
   var histograms: AttributeMap[Histogram] = AttributeMap(Seq())
)
```

The basic column level statistics , such as maximal value, minimal value, number of distinct values, number of null values, are saved in this data structure:

```
case class BasicStat(
   dataType: DataType,
   max: Any,
   min: Any,
   ndv: Long,
   numNull: Long,
   maxLength: Long,
   avgLength: Double
)
```

3) Statistics Collection

SQL Commands

CBO relies on detailed statistics to optimize a query execution plan. Users can use these commands to collect/display statistics.

- SQL statement: `ANALYZE TABLE table_name COMPUTE STATISTICS`

  The above SQL statement can collect table level statistics such as number of rows, number of files (or HDFS data blocks), and table size in terms of byes. Note that both `ANALYZE` and `COMPUTE` are reserved keywords.

- SQL statement: `ANALYZE TABLE table_name COMPUTE STATISTICS FOR COLUMNS column-name1, column-name2, ….`

  The above SQL statement can collect column level statistics for the specified columns. For a specified column, the information includes maximal column value, minimal column value, number of distinct values, number of null values, histogram for column data distribution.

- SQL statement: `DESCRIBE EXTENDED table_name`

  The above SQL statement can display the metadata, including table level statistics, of a table in an extended format.

- SQL statement: `DESCRIBE EXTENDED table_name column_name`

  The above SQL statement can display the metadata, including column level statistics, of a specified column in an extended format.

Statistics Collection Functions

In `SparkSQLParser` class, we made SQL parser recognize the reserved keywords `COMPUTE STATISTICS`. Based on the specified table name and column name, function `analyzeTable` or `analyzeColumns` are called respectively.

Function `analyzeTable` first gets row count, hdfs file size, and hdfs data block count of a table. And then it holds the information in a `PlanStatistics` structure. The row count is actually obtained by running a SQL statement like "`select count(1) from table_name`". Using SQL statement to get row count is a fast way as we piggy-back on top of Spark SQL's execution parallelism. Histogram information is handled separately and will be discussed in the next section

Similarly Function `analyzeColumns` gets the basic statistics information for a given column. The basic statistics such as max, min, number-of-distinct-values, etc are obtained by running

SQL statements to get them. And then it holds the information in a `PlanStatistics` structure.

Histogram and its Algorithm

We support two kinds of histograms:

- Equi-width histogram: We have a fixed width for each column interval in the histogram. The height of a histogram represents the frequency for those column values in a specific interval. For this kind of histogram, its height varies for different column intervals. We use the equi-width histogram when the number of distinct values is less than 254.
- Equi-height histogram: For this histogram, the width of column interval varies. The heights of all column intervals are the same. The equi-height histogram is effective in handling skewed data distribution. We use the equi- height histogram when the number of distinct values is greater than or equal to 254.

For both types of histograms, we assume column data is evenly distributed within a single column interval (*a.k.a.* histogram bucket) if there are multiple distinct values in a single column interval.

The Histogram information of all the columns referenced in a SQL query is saved in the following field inside `PlanStatistics` class.

```
var histograms: Map[AttributeReference, Seq[Bucket]]
```

The histogram information of a given column is saved as a sequence of buckets.

We implemented a generic, re-usable histogram class that supports partial aggregations. The algorithm is a heuristic adapted from the paper [BenH2010]. Although there are no approximation guarantees, it appears to work well with adequate data and a large number of histogram bins (>100). We highlight the algorithm as below:

Phase 1:

1. Run parallel processes to scan full table to collect data statistics for a column.
2. Each process will use a maximal number of buckets, for example 1000. Binary search is used to locate the proper bucket position.
3. When the maximum reaches, it adapts to consolidate with neighboring buckets.

Phase 2:

1. All processes send local bucket data to a central node
2. Consolidate all bucket data and form final 254 buckets at most.

4) System Catalog

To save the collected table/column statistics information into system catalog, we use polymorphism to take advantage of the externally visible interface `ExternalCatalog`. For `updateTableColumnStats` method defined in `ExternalCatalog`, we override this method `updateTableColumnStats` in child class `HiveExternalCatalog`. Doing so can alleviate our work from dealing with meta store directly. It is up to Hive (`HiveClient`) to save system catalog information into either a database like MySQL or just a file.

```
override def updateTableColumnStats(
   catalogTable: CatalogTable,
    attrStatsMap: Map[AttributeReference, Seq[Any]]):
    Map[String, String] = withHiveState
{……  …… }
```

Inside this method, we organize column statistics information as an array buffer of Hive `ColumnStatisticsObj` objects. Finally we call Hive method `updateTableColumnStatistics` to store the collected column statistics into system catalog.

To fetch the collected table/column statistics information from system catalog, we use same polymorphism mechanism. For `getTableColumnStats` method defined in `ExternalCatalog`, we override this method `getTableColumnStats` in child class `HiveExternalCatalog`.

```
override def getTableColumnStats(
   catalogTable: CatalogTable,
   attributes: Seq[AttributeReference]):
    Map[AttributeReference, Seq[Any]] = withHiveState
{……  …… }
```

Inside this method, we first call Hive method `getTableColumnStatistics` to retrieve the column statistics from system catalog. Then we decompose Hive `ColumnStatisticsObj` objects to get the column statistics information we want.

5) Cost Functions

The CBO can select good physical strategy for an execution operator. For example, CBO can choose the build side selection for a hash join operation. It can also decide whether or not a

broadcast join should be performed. Besides, the execution sequence of the database operators for a given query can be re-arranged. CBO can choose the best plan among multiple candidate plans for a given query. The goal is to select the candidate plan with the lowest cost.

Multi-way Join Ordering Optimization

Spark SQL optimizer's heuristics rules can transform a SELECT statement into a query plan with the following characteristics:

- The filter operator and project operator are pushed down below the join operator. That is both filter and project operators are executed before join operator.
- Without subquery block, the join operator is pushed down below the aggregate operator for a select statement. That is a join operator is usually executed before aggregate operator.

With this observation, the biggest benefit we can get with CBO is with multi-way join ordering optimization. Using dynamic programming technique, we try to get the globally optimal join order for a multi-way join query. We illustrate our algorithm as below.

Procedure MultiwayJoinOrdering(Set of *k* join input relations, set of join conditions)
// Suppose there are k join input relations R_1, R_2, …., R_k ; Ω = { R_1, R_2, …., R_k}
// || R_i || represents the number of elements in set R_i

// Variable K_j represents j-way join cost, where 1 <= j <= *k*

if (*k* == 1) **then** { // reduce to one join input relation
return the size of the join input relation $||R_1||$
}

for i = 1 **to** *k* **do** {
set cost K_{L-1}(i) by calling MultiwayJoinOrdering (Ω minus R_i);
}

Choose join input relation u and its relevant join condition subject to:
minimize { K_{L-1}(i) + <join cost when adding R_i >, where 1 <= i <= *k* }
End Procedure

In the above algorithm, we use the term "join input relation" to refer to the data set to be used as input to a join operator. A join input relation usually is the intermediate data set after applying filter operator, predicate operator, and/or other join operators.

Clearly the join cost is the dominant factor in choosing the best join order. We now discuss how we set the cost functions for various execution operators. We first discuss our phase 1 (or current) implementation. And then we discuss how we plan to enhance it in the future.

Phase 1 Cost Functions

In phase 1, we greatly simplified the cost of a join operator by setting it to the number of join output rows. For our multi-way join algorithm, we compute the cost of each candidate plan with this formula:

<cost-of-join-order-plan> =
    <summation-of-cardinalities-of-join-input-relations-at-leaf-node-level> +
    <summation-of-all-join-output-rows-for-a-given-join-ordering>

As each join input relation is read once, each candidate plan has same value for <summation-of-cardinalities-of-join-input-relations-at-leaf-node-level>. Hence, we can further simplify the cost of a join ordering plan as:

<cost-of-join-order-plan> = <summation-of-all-join-output-rows-for-a-given-join-ordering>

Hive 0.14 Cost Based Optimizer used a similar approach by focusing on join selectivity when it chooses the best plan for multi-way join queries [Mokh2015]. Note that the number of join output rows can be computed by multiplying join input relation size by the corresponding join selectivity.

Phase 2 Cost Functions

It is fair to say that setting join cost to the number of join output rows is a heuristics approach. In the future, we plan to derive cost formula by estimating the cost of each execution operator on modern hardware architecture [Baus2012].

As defined in [Pull2013], the following are the important cost parameters to use in the cost function formula:

- Hr = Cost of reading 1 byte from HDFS
- Hw = Cost of writing 1 byte to HDFS
- Lr = Cost of reading 1 byte from local FS
- Lw = Cost of writing 1 byte to local FS
- NEt = Average cost of transferring 1 byte over network in Spark cluster from any node to ant node
- T(R) = Number of tuples in the relation

- Tsz = Average size of the tuple in the relation
- V(R, a) = Number of distinct values for attribute a in relation R
- CPUc = CPU cost for comparison function

There are many physical database operators. As an example, we list the cost formula for table scan operator and hash join operator. We use Hive Optimizer's cost formula as a base and adapt them to Spark SQL. A detailed description of cost functions for other execution operators can be found in [Pull2013].

Table Scan Cost

In Phase I, we have such assumptions for Table Scan Cost,

- T(R) = the cardinality of relation R
- Tsz = average tuple size
- CPU usage is 0, all cost are I/O cost

And the Table Scan Cost is,

*Table Scan Cost = I/O Usage = Hr * T(R) * Tsz*

Hash Join

For Hash Join, normally we should build the hash table using small table, and probing using the large table. This way can consume less memory buffer for in-memory hash table. As we know, building hash table is more complex than just doing comparison. So here we should set the ratio between them = Times(Hash).

Times(Hash) = CPUhash / CPUprobe

Here we assume CPUprobe = CPUc, so CPUhash = CPUc * Times(Hash)

- T(R) = Join Cardinality Estimation
- Tsz = average tuple size based on join schema (i.e., tables used)

  // TODO: need to verify
- CPU Usage = Cost of Building hash table for the small side table + Cost of Probing with the large side table
  = T(Rsmall) * CPUc * Times(Hash) + T(Rlarge) * CPUc

  // TODO: need to verify

- IO Usage = Cost of reading Small Table + Cost of reading Large Table + Cost of writing to local FS when local memory is overflow.
  = T(Rsmall) * Lr * Tsz(small) + T(Rlarge) * Lr * Tsz(large) +
  (T(Rlarge) * Selectivity(this_query) – Local_Buffer_Size) * Lw * Tsz(large)

Here we have the assumption that all of hash join operations will be done in the local nodes. We don't consider the cost of data shuffling at this stage.

Clearly the cost formula is dependent on the implementation of Spark SQL execution engine. There will be lots of calibration work to validate all the cost parameters. While we use Hive's optimizer cost formula as a base here, there is still lots of work to validate the cost formula as Spark SQL's execution engine is quite different from Hive's execution engine.

Also Spark SQL supports operating on a variety of data sources and file formats. We need to factor in the performance characteristics of a specified file format such as Parquet. We are open-minded in supporting other file formats as needed. For example, if a file format is good at interactive analytics because of its superior index support, then we will cover the cost function for index support as well as table scan.

6) Cardinality Estimation of Predicate Expressions

A filter condition is the predicate expression specified in the WHERE clause of a SQL select statement. A predicate can be a compound logical expression with logical AND, OR, NOT operators combining multiple single conditions. A single condition usually has comparison operators such as =, <, <=, >, >=, 'like', etc. Hence, it can be quite complex when we evaluate the overall filter factor.

With 'predicate pushdown', a filter condition is usually executed at a pretty early stage of a query plan. Hence, the cardinality estimate of a predicate expression can greatly impact the precision of cardinality estimates of the subsequent execution operators.

We defined a recursive function `calculateConditions` so that we can calculate the filter factor of predicate expression in a top-down recursive (or depth first search) fashion. The output of this method is a filter factor between 0.0 and 1.0.

```
def calculateConditions(
    planStat: PlanStatistics,
    condition: Expression,
    update: Boolean = true): Double =
{ …..  ….. }
```

We now describe the various logical conditions forming a compound expression. In the following description, we focus on computing the filter factor. Actually the basic statistics such as minimal value, maximal value, and number of distinct values should also be adjusted accordingly.

Logical AND Operator

For logical AND expression, its filter factor is the filter factor of left condition multiplied by the filter factor of the right condition. Now we express filter factor *ff* of expression (a AND b) in a mathematical formula as below:
ff(a AND b) = *ff*(a) * *ff* (b)

Logical OR Operator

For logical OR expression, its filter factor is the filter factor of left condition, plus the filter factor of the right condition, and minus the filter factor of left condition logical-AND right condition. We can express the computation of filter factor for OR-expression as:
ff(a OR b) = *ff*(a) + *ff* (b) - *ff*(a AND b) = *ff*(a) + *ff* (b) – (*ff*(a) * *ff* (b))

Logical NOT Operator

For logical NOT expression, its filter factor is 1.0 minus the filter factor of the original expression. We can express the computation of filter factor for NOT-expression as:
ff(NOT a) = 1.0 - *ff*(a)

Single Logical Condition

In a single logical condition, we have comparison operators, such as =, <, <=, >, >=, etc. For string data type, we also need to deal with the LIKE operator.

- Equal To Condition: We need to check if the constant value of the condition falls within the qualified range between the current minimal column value and current maximal column value. This is necessary because the range may change due to a previous condition already applied earlier. If the constant value is outside the qualified range, then the filter factor is 0.0. Otherwise, it is the inverse of the number of distinct values. Note that, without histogram information, we assume a uniform distribution for the column

values. If there is histogram information, we need to locate the right histogram bucket to find out the number of distinct values in it.

- Less Than Condition: We need to check if the constant value of the condition is smaller than the current minimal column value. If so, then the filter factor is 0.0. Otherwise, we need to compute the filter factor based on the available information. If there is no histogram, the filter factor is set to (constant – minimum)/(maximum – minimum). If there is histogram, we can calculate the filter factor by adding up the density of histogram buckets between current column minimal values and the constant value. Also the constant of the condition becomes the new maximal column value.

- Less Than or Equal To condition: This is similar to 'less than condition' by adding the boundary condition (which is equal to constant value).

- Greater Than Condition: We need to check if the constant value of the condition is greater than the current maximal column value. If so, then the filter factor is 0.0. Otherwise, we need to compute the filter factor based on the available information. If there is no histogram, the filter factor is set to (maximum - constant)/(maximum – minimum). If there is histogram, we can calculate the filter factor by adding up the density of histogram buckets between the constant value and current column maximal value. Also the constant of the condition becomes the new minimal column value.

- Greater Than or Equal To condition: This is similar to 'greater than condition' by adding the boundary condition (which is equal to constant value).

- In condition: For SQL statement with IN operator followed by a list of literals, conceptually it is equivalent to multiple equal-to conditions without overlapping. We can first compute the total valid size by combining all the literals. The filter factor is the total valid size divided by the number of distinct values in a qualified column interval.

- Like Condition: For string operator LIKE, we first utilize Java library to transform a regular expression into a Java pattern object. Then we compare the pattern with the range for each histogram bucket one-by-one to see if there is a match. We accumulate the filter factor when we traverse the histogram buckets.

7) Cardinality Estimation of Execution Operators

When computing the cost factor of each execution operator, we use the following variable to hold the cost estimate in `LogicalPlan` class:

```
lazy val costEstimate: Double = _
```

We use a lazy val variable to hold the cost estimate because we only need to estimate its value only once for each plan node.

We essentially compute the number of output records and output size in bytes for each execution node. The query execution plan is basically a DAG (Directed Acyclic Graph) with child nodes executed before parent nodes. The output of a child node is immediately used as the input of its parent node. We compute the output cardinalities of execution nodes and propagate them up along the query plan DAG in a post-order traversal.

We now describe the cardinality estimate algorithm used in each database operator.

Filter Operator

For filter operator, our goal is to compute the filter to find out the percentage portion of the previous (or child) operator's output after applying the filter condition. The filter factor is a double number between 0.0 and 1.0. The number of output rows for filter operator is basically the number of its child node's output times the filter factor. Its output size is the its child node's output size times the filter factor.

A filter condition is the predicate expression specified in the WHERE clause of a SQL select statement. The predicate expression can be quite complex when we evaluate the overall filter factor. We described how we handle expression statistics in section 6.

Join Operator

Before we compute the cardinality of a two-table join output, we should already have the output cardinalities of its child nodes on both sides. The cardinality of each join side is no longer the number of records in the original join table. Rather it is the number of qualified records after applying all execution operators before this join operator.

At this point, we should already have the number of distinct values for a qualified range. With histogram information on join column, we can compute the number of records per distinct value on both left join side and right join side for qualified range. Based on join type, we can compute join output cardinality. For example,

For equi-join: <join-output-cardinality> = <number-of-records-per-distinct-value-on-left-side>
X <number-of-distinct-values-within-qualified-range-on-right-side>

For Cartesian-product-join: <join-output-cardinality> =

<number-of-records-per-distinct-value-on-left-side>
X <number-of-records-per-distinct-value-on-right-side>
X <number-of-distinct-values-within-qualified-range>

If a user collects join column statistics, then we know the number of distinct values for each join column. Since we also know the number of records on the join relation, we can tell whether or not a join column is a unique key. We can compute the ratio of number of distinct values on join column over the number of records in join relation. If the ratio is close to 1.0 (say greater than 0.95), then we can assume the join column is unique. Therefore, we can precisely determine the number of records per distinct value if a join column is unique.

At the later phase, we decide the actual join algorithm to use for a join query. If we get reliable statistics information on both join sides, then we can properly assign the smaller join input relation as the build side of a hash join operation. In `CanBroadcast` object, we revised the logic to properly decide which join side can be broadcast using the computed statistics saved in `PlanStatistics` structure of each join side.

Aggregate (or Group-by) Operator

To compute the output cardinality of a group-by operator, we need to know the number of distinct values within the qualified query range. If there are multiple columns in the group-by clause of a SQL select statement and all the group-by columns are not correlated, we can compute the output cardinality by multiplying the number of distinct values for each group-by column.

If the group-by columns are correlated, we find out if any group-by column is unique. As mentioned in the previous sub-section, we can determine a column's uniqueness by finding the ratio of number-of-distinct-values in a group-by column over the number of record in its table. If a group-by column is unique, we should ignore those non-unique group-by columns in the same table because a non-unique column is dependent on a unique (or primary) key column. We now use TPC-H query 10 as an example:

```
SELECT c_custkey, c_name, sum(l_extendedprice * (1 - l_discount))
     AS revenue, c_acctbal, n_name, c_address, c_phone, c_comment
  FROM nation JOIN customer JOIN orders JOIN lineitem
 WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey
   AND o_orderdate >= '1993-10-01' AND o_orderdate < '1994-01-01'
   AND l_returnflag = 'R' AND c_nationkey =  n_nationkey
 GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address,
c_comment
 ORDER BY revenue DESC limit 20
```

In the above query, column `c_custkey` is unique. Then we can ignore columns c_name, c_acctbal, c_phone, c_address, c_comment which are all in same customer table. We can obtain the group-by output cardinality by multiplying the number of distinct values in c_custkey and the number of distinct values in n_name.

Union Operator

The output cardinality of a union operator is the summation of output cardinalities of its two child nodes. However, we need to adjust those basic statistics such as the minimal value, maximal values, and number of distinct values, etc.

Project Operator

The output cardinality of a project operator is same as the output cardinality of its child node.

Limit Operator

The output cardinality of a limit operator is set to the LIMIT constant specified in SQL statement.

Sort Operator

The output cardinality of a sort operator is same as the output cardinality of its child node.

8) References:

[Baus2012] Daniel Bausch, Ilia Petrov, Alejandro Buchmann, "Making Cost-Based Query Optimization Asymmetry-Aware", the 8th International Workshop on Data Management on New Hardware, May 21, 2012, https://www.dvs.tu-darmstadt.de/publications/pdf/BauschDaMoN2012.pdf

[BenH2010] Yael Ben-Haim and Elad Tom-Tov, "A streaming parallel decision tree algorithm", Journal of Machine Learning Research 11 (2010), page 849--872. http://www.jmlr.org/papers/volume11/ben-haim10a/ben-haim10a.pdf

[Hu2016] Ron Hu, Fang Cao, Min Qiu, Yizhen Liu, "Enhancing Spark SQL Optimizer with Reliable Statistics", Spark Summit 2016 Conference, June 8, 2016, San Francisco, California, https://spark-summit.org/2016/events/enhancing-spark-sql-optimizer-with-reliable-statistics/ .

[Mokh2015] Mostafa Mokhtar, "Hive 0.14 Cost Based Optimizer (CBO) Technical Overiew", March 2, 2015, http://hortonworks.com/blog/hive-0-14-cost-based-optimizer-cbo-technical-overview/

[Pull2013] John Pullokkaran, "Introducing Cost Based Optimizer to Apache Hive", published at https://cwiki.apache.org/confluence/download/attachments/27362075/CBO-2.pdf , 10/08/2013

[Spar2016] Spark issue 16026, "Cost-Based Optimization Framework", reported by Reynold Xin, 6/17/2016, https://issues.apache.org/jira/browse/SPARK-16026

9) Appendix A: Summary of New/Updated Source Files

For ease of reference, we summarize the new and changed source code files in this section.

Parser

- SparkSQLParser.scala: parse compute command
- commands.scala: implementation of `AnalyeTableCommand` and `AnalyzeTableColumnCommand`
- SQLConf.scala: defined new parameter `'spark.sql.cbo'`

System Catalog

- ClientWrapper.scala: added functions `updateTableColumnStats` and `getTableColumnStats`

Statistics

- LogicalPlan.scala: added `planStat` val variable
- Statistics.scala: added `PlanStatistics` class

- New file histograms.scala

Estimation of Expression Evaluation

- New file FilterCardEstimator.scala

Cardinality Estimation of Execution Operators

- SQLContext.scala: inserted `CostEstimation` strategy
- SparkStrategies.scala: defined `CostEstimation` class, updated `CanBroadcast` class
- HiveMetastoreCatalog.scala: set table scan cardinality
- New file CardinalityEstimator.scala
- New file AggregateCardEstimator.scala
- New file FilterCardEstimator.scala
- New file JoinCardEstimator.scala
- New file UnionCardEstimator.scala

We want to check in source code one component a time in order to avoid overriding same file again and again. We plan to check-in our source code files in multiple pull requests as needed.

10) Appendix B: New Configuration Parameters

- spark.sql.cbo: A Boolean field. Set this parameter to true if you want to enable CBO. Set it to false if you want to disable CBO. The default value is false.