Theta Join

Preliminaries

Overview

HIVE-556 requests that Hive support non-equality joins commonly called theta joins. Theta joins cover all kinds of non-equality joins such as <, <=, >, >=, <>, LIKE, RLIKE, and Generic UDF.

Range joins, a specific kind of theta join, is extremely useful in temporal and spatial applications. This work will focus first on range join as it's arguably the most important theta join. After range join is implemented the remaining theta join operators can be implemented with relative ease. Supporting theta join and specifically range join is important as it will makes Hive competitive in additional use cases and removes a deficiency when compared against traditional systems.

Specific Use Cases

Geo-Location

This use case is very common today for many common users of Hive websites, mobile telephone services, online ad marketers. The user wants to join website access logs with geo-location information of ip addresses. This can be implemented as an interval join. Currently this use case is implemented by creating a custom UDF which performs the lookup. Compiling and deploying UDF's is an error prone process and not a common task for most users of Hive.

As an example of this use case, we can utilize the Maxmind GeoIP by country dataset which is 79,980 records. Consider joining that dataset with a very small access log of 5,000,000 entries using a cartesian product and then filtering the result. This approach results in 399,900,000,000 tuples that need to be filtered in the mapper. Hive trunk was modified to implement map-side interval join and the previously discussed join completed in 21 seconds on a single host while the cartesian product then filter approach ran for many hours before it was killed.

Side-Table Similarity

This use case was originally proposed in HIVE-556 by it's creator Min Zhou. The user has a side table with some entries and wants to join the side table with a main table based on similarity. The example provide was using the RLIKE operator but LIKE or a generic boolean UDF could be used as well.

Requirements

A notable non-goal of this work is n-way theta joins. The algorithm to implement n-way theta joins was inspired by the algorithm to implement 2-way theta joins. Therefore n-way theta joins can be implemented in future work.

Goals

- Map-side 2-way theta joins
- Reduce-side 2-way theta join

Specific Non-Goals

- Map-side n-way theta joins
- Reduce-side n-way theta joins
- Additional statistic collection

Literature Review

Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters [1]

This work adds a Merge step to Map-Reduce which allows for easy expression of relational algebra operators. This is interesting but not immediately useful as it requires modification of the Map-Reduce framework it's not immediately useful.

Efficient Parallel Set-Similarity Joins Using MapReduce [2]

This work studies a special type of set similarity, specifically similar strings or bit vectors. This work could be useful in implementing some operators such as LIKE. In addition this method which requires statistics to be calculated at run time results in multiple Map-Reduce jobs.

Processing Theta-Joins using MapReduce [3]

This work proposes an algorithm 1-Bucket-Theta to perform theta joins on two relations in a single Map-Reduce job given some basic statistics, namely the cardinality of the two relations. This approach allows parallel implementation of cartesian product as well. The work also details how additional input statistics can be exploited to improve efficiency. The approach is to partition a join-matrix of the two relations.

Efficient Multi-way Theta-Join Processing Using MapReduce [4]

This work is inspired by [3] and expands the method to N-way joins. The approach used is to partition a hypercube of the relations. An approach to merge the resulting many Map-Reduce jobs into a single job is also discussed with results similar to Y-Smart [5].

Design

Map-side

A large number of theta join use cases have the nice property that only one of the relations is "large". Therefore many theta joins can be converted to map-joins. Presently these use cases utilize a map-side cartesian product with post-join filters. As noted in the geo-location use case above some of these use cases, specifically range joins, can see several orders of magnitude speedup utilizing theta join.

Currently Map-side join utilizes a hashmap and a join is performed when the incoming key matches a key in the hash map. To support range join this will abstracted into a pluggable interface. The plugin can decide how two keys are joined. The equality join interface will continue to utilize a hashmap while range join can use a data structure such as an interval tree. Other such optimizations can be made. For example the not equals join condition <> can use a view on top of a map.

Reduce-side

Reduce-side joins will be implemented via 1-Bucket-Theta as described in [3]. This requires the cardinality of the two relations and therefore to perform a reduce-side theta join statistics must be turned on. Initially if the required statistics do not exist an exception will be thrown indicating the problem. After the initial implementation we can use a method to estimate the cardinality.

As previously mention a detailed description of 1-Bucket-Theta is located [3]. As such the discussion of the internals of the algorithm will be brief. Joining two relations S and T can be viewed as a matrix with the S, the smaller relation, on the left and T on the right.

The matrix is partitioned by r, the number of reducers. An example join matrix follows, with four reducers 1-4 each a separate color:

Row Ids	**T 1**	**T 2**	**T 3**	**T 4**
S 1	1	1	2	2
S 2	1	1	2	2
S 3	3	3	4	4
S 4	3	3	4	4

In the map phase, each tuple in S is sent to all reducers which intersect the tuples' row id. For example the S-tuple with the row id of 2, is sent to reducers 1, and 2. Similarly each tuple in T is sent to all reducers which intersect the tuples' row id. For example, the tuple with rowid 4, is sent to reducers 2 and 4.

In Hive and MapReduce row ids aren't common available. Therefore we choose a random row id between 1 and |S| (cardinality of S) for S and 1 and |T| (cardinality of T) for T. Thus a reduce-side theta join must know the estimated cardinality of each relation and statistics must be enabled. Random row id's will result in well balanced reducer input when processing larger relations. As noted in [3], the partitioning scheme works such that if a relation is much smaller than it's pair the smaller relation will be broadcast two all reducers. As such therefore random-skew which would occur for small relations does not impact the algorithm in practice. Additionally in Hive if a relation is small the join is converted to a map-side join and 1-Bucket-Theta is not utilized.

Mapper

In the mapper the join matrix will be initialized, a random row id chosen, and then the tuple will be emitted for each reducer that intersects the row id. Hive already has a mechanism to set the hash code for a specific tuple which can be re-used here. Additionally the tuples will need to be sorted in such a way so that tuples for S arrive in the reducer first. Luckily Hive already implements this via the JoinReorder class.

Reducer

The reducer is fairly simple, it buffers up the S relation and then performs the requested join on each T tuple that is received.

References

1. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters
2. Efficient Parallel Set-Similarity Joins Using MapReduce
3. Processing Theta-Joins using MapReduce
4. Efficient Multi-way Theta-Join Processing Using MapReduce
5. HIVE-2206