Reference

1. Processing Theta-Joins using MapReduce, Alper Okcan, Mirek Riedewald Northeastern University
    1.1. Video of presentation on above paper
2. Optimizing Joins in map-reduce environment F.N. Afrati, J.D. Ullman
3. Efficient Multi-way Theta-Join Processing Using MapReduce http://vldb.org/pvldb/vol5/p1184_xiaofeizhang_vldb2012.pdf
4. Hive theta join proposal from Brock - https://cwiki.apache.org/confluence/display/Hive/Theta+Join
5. spark theta join pull request - https://github.com/apache/spark/pull/2939
6. Optimizing Theta-Joins in a MapReduce Environment Changchun Zhang

Reference 1 -
The paper proposes algorithms to minimize the max runtime of reduce task in a join. It classifies joins as input dominated vs output dominated. It uses a matrix to represent the join output, and makes it a problem of dividing the matrix into r equal areas, where r is the number of reducers available.
For cartesian products and output dominated cross products, it proposes 1-Bucket-Theta join algorithm. To minimize the max reducer inputs as well, it tries to divide the matrix into equal squares of size sqrt(|s|*|t|/r). An input record is assigned a random point (on x axis for input 1, on y axis for input 2), and that record is duplicated for each region it belongs to, with a reduce key corresponding to the region. The algorithm requires knowledge (or estimate) of the number of input records for each join input.

1-bucket-theta ends up duplicating the input records, as it sends them to multiple reducers. As a result, it is not optimal, if the join is input size dominated.
For input dominated joins (the join predicate being highly selective), it recommends m-bucket-I . This needs **equi-depth** histograms (finer the better). This works only for simpler join conditions where you can use the histogram to figure out the areas in matrix that will have results, based on the histogram. Then the problem becomes one of creating equal areas around regions with output, and send one to each reducer. It proposes an algorithm/heuristic for it.

It also suggests creating more reduce tasks to deal with problems in keeping reducer inputs in memory.

Reference 2 - This paper is more about multi-way joins and how to optimize communication costs. It focuses on equi-join. (It might be interesting to find what it has to offer anyway)

1. Proposal for cross product implementation

Query Planning Phase (hive client)

1. Check if stats are available. The number of records is essential. Any statistics that help in estimating memory footprint will also be useful.

TBD : If stats are not available, should we collect run time stats for inputs to join ? we might see OOM without it. what does broadcast join do ?

2. Matrix division (in hive client) - (this part needs to be optimized further so that the total records being shuffled is minimized)
    a. Split smaller side into chunks that fit into memory. (not much choice here, unless we implement a more complex algorithm that writes to disk and reads back from it - even then the perf benefits of it are not very clear).
    b. Split larger side to increase the number of tasks as needed. (somewhat larger splits than usual ? - increasing the number of splits of larger size increases (shuffle) replication factor of smaller side)
3. Write matrix division results to query plan. (This would be very small, < 1kb).

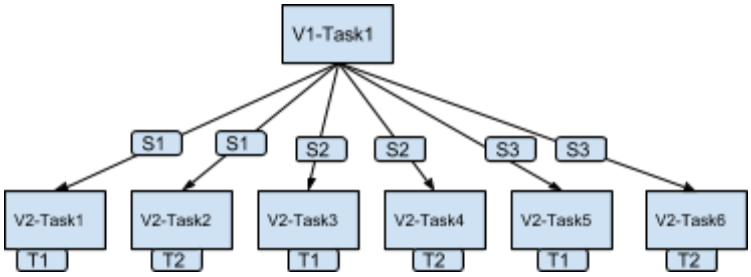
Example of division : Inputs S, T . Smaller input S divided into S1,S2 , S3. Larger input T divided into splits T1, T2

	T1	T2
S1	Task1	Task2
S2	Task3	Task4
S3	Task5	Task6

4. Wiring up the plan. The plan details are described in the discussion of execution phase.

Execution Phase

There will be 2 sequential vertices associated with this. The above matrix would translate into the tasks in the figure below -



The first vertex will process the smaller input (S in example above) and send the inputs to a subset of of reduce tasks. For example input S1 is sent to Reducer1, Reducer 2. This would need a custom edge ?
The division of input S into S1 - S3 would be done in a round robin fashion.

The 2nd vertex would use input T as its split-input (/probing input), and gets inputs from vertex 1 using a custom? edge.
The inputs from first vertex will be stored in a list like data structure. (we could also use broadcast-join implementation by creating an artificial join key, but its memory footprint is going to be higher.). Every record in the probing input would be joined against every record in the List.

2. Proposal for general theta-join implementation

Using a cross-product + filter is possible for implementing theta-joins, but if the output of the join is a small subset of cross-product, then this is very inefficient. Also, that approach only works for inner-joins.
A more efficient solution will be to have a modified join operator instead of the cross-product operator. This way the filter can be applied before the joined records are created.
The implementation would be same as cross product implementation described above, except for the difference in this operator.
The case for outer joins needs more thought, since we are duplicating the records across tasks during the join, we might have to have restrictions on the parallelism or do some additional processing to support outer theta joins.

3. Proposal for optimizing keys joined during theta-join, for certain types join conditions

For certain join conditions (!= , < , > etc) we don't have to send inputs required to compute a cross product. We can use equi-depth histograms on both inputs for these join conditions to find a subset the buckets (represented in histogram) that need to be joined with each other.

This optimization will be provide significant gains in performance depending on the join selectivity is low (the smaller the output, the more gains because of this optimization).
The requirement for equi-depth histogram in this case is similar to the requirement for stats for parallel sorting of data.
Histograms are not available as part of hive stats today.
This is based on the m-bucket-I algorithm proposed in the theta-join paper. The algorithm is proposed for inner-joins. Implementation for outer join needs more thought.

Query Planning Phase (hive client)

1. Check if the join condition can be applied on the buckets in a histogram to find the records that need to be joined against each other. If not, use the above unoptimized join algorithm.
2. If join column histograms are not available, compute histogram on the input. We also need to explore operators passing down modified stats. The finer the stats are, the better.
3. Use the join condition and join key histograms to find the regions in the matrix that will produce join output.
4. Divide the regions providing join output into reducer regions using heuristic based algorithm proposed in the m-bucket-I paper, also restrict region size, so that smaller side fits into memory.

Rest of the query planning and execution phase remains same as the general theta-join case.

4. Proposal for theta-join operator, for certain types join conditions

In case of certain join conditions that involve range operations such as " < " or " >", instead of comparing all rows, we can optimize the join by using a range-tree or similar data structure (Kd-tree seems better option when more than one join-key is used).

Query Planning Phase (hive client)

The additional steps required for this optimization -

1. Some transformations to normalize the join condition might be necessary
2. In choosing the small (in-memory) input of join it might make sense to also consider the number of dimension keys that would need to be stored in the range data structure. The memory footprint of this data structure needs to be estimated differently (compared to the general theta-join case discussed above).
3. This needs a modified version of broadcast-join operator that creates the data structure optimized for range lookups, and uses it for the lookups. Add new operator to query plan.

Execution

Modified broadcast-join operator is used during execution.

Estimates for cross product + theta join

==========

Support parallel cross product - 41 days

==========

0. logical optimizations for cross product is same as any inner join (push up filter etc). Treat cross product differently only when it is converted to Tez plan.

Optimizer similar to ConvertJoinMapJoin is invoked when Tez plan is being generated.
This is what optimizer does -
- More details of changes - 5 days
- switch join operator with a cross-product operator - 1 day
- estimate number of records for both inputs (what additional work is involved here?) - 1 day
- matrix division algorithm
- estimate max number of records of smaller size that would fit into memory (there is some code that does similar work, estimating based on schema, might need refactoring) - 2 days
- matrix division algorthm - 3 days
- Setup reduce sink operators (tez custom edges) to divide work appropriately based on the matrix division - 8 days
- Changes to serialize this information in reduce sink operators - 2 days
- GenTezWork issues in other operators (risk) - 5 days
- implementing CrossProduct operator - 10 days
- Testing at scale - 4 days

==========

Convert theta-join into cross product + filter = 16 days

- Any parser changes to support > , < etc in join condition - 4 days
- optimizer changes - (John's estimate) - 10 days
- Testing at scale - 2 days

==========

Optimize theta-join execution using datastructures optimized for range queries - 47 days

==========

- applicable to certain theta-join conditions
- more design - 5 days
- identify if the join condition is appropriate for this optimization - 10 days
- research appropriate data structure - 3 days

- model the memory footprint of the data structure - 3 days
- modify matrix computation based on the estimated memory footprint - 3 days
- modify join operator to store new kind of join expression - 3 days
- implement optimized join operator - 15 days
- testing at scale - 5 days

==========
Prune the records being shuffled (reduce input to the theta join operation) - 28 days
==========
- query plan for histogram compute job - 5 days
- run query to compute histogram on columns - 10 days
- compute matrix based on the histogram - 5 days
- update partitioning to happen based on the value - 3 days
- Testing at scale - 5 days