A Comparison of Join Algorithms for log processing in MapReduce
Agenda:

- Map Reduce
- Basic Map Reduce Join Algorithms
- Improvements in performance by pre processing techniques
- Experimental evaluation
- Comparison with Pig framework algorithms.
- Future Work
Brief Introduction of MapReduce

- Used to analyze large volumes of data
- One important type of data analysis done is Log Processing.
- Success of map reduce lies in parallelization, fault tolerance, load balancing in a simple framework
- User defines a map and reduce function
- Map function takes a key value pair (K,V) as the input and generates some other pairs of (K’,V’) as the output.
- The reduce function takes as input a (K’,LIST_V’) pair, where LIST_V’ is a list of all V’ associated with a given K’.
- The Reduce function produces another key value pair as output.
Features of Map Reduce

Parallelization

Fault Tolerance

Load Balancing
Why is Map Reduce preferred over RDBMS for Log processing??

- Amount of data
  Ex: China mobile gathers 5-8 TB of phone call records per day
- Log records always do not follow the same schema
- All the records with in a time period are analyzed together
- Less fault tolerance in RDBMS. Resume capability of a query in Map Reduce.
- The Hadoop implementation of Map Reduce is available open source but RDBMS costs thousands of dollars.
Join Algorithms in Map Reduce

- We use the Map Reduce framework as is, without any modification so that all the features of mapreduce are preserved.
- We consider an equi-join between a log table L and a reference table R on a single column.
- We assume that both L and R as well as the join result are stored in DFS.
- For each strategy, we consider further improving its performance with some preprocessing techniques.
- Although these are well known in RDBMS, implementing them in Map Reduce is not easy.
The declarative query languages appearing on top of Map Reduce, such as Pig from Yahoo!, Hive from Facebook, and Jaql from IBM, already use some of these equi-join strategies but often implement them in a less efficient way.

We assume that each map or reduce task can optionally implement two additional functions init() and close().

A MapReduce job can be configured to be map-only, i.e., no reduce function is specified. In that case, the sort and the merge phase are bypassed and the output of each map task is stored as a separate file directly in the DFS, instead of in the local file system.
Repartition Join

- Most common type of join in Map Reduce
- L and R are dynamically partitioned on the join key (tagged) processed in Mappers and merged, sorted then send corresponding partitions (on same join keys) of L, R to a same node (Reducer), then the corresponding pairs of partitions are joined on a Reducer.
- This join strategy resembles a partitioned sort-merge join in the Parallel RDBMS literature.
- Join algorithm provided in the package of Hadoop (org.apache.hadoop.contrib.utils.join)
- Implemented in one Map Reduce job
- In the map phase, each map task works on a split of either R or L.
To identify which table an input record is from, each map task tags the record with its originating table, and outputs the extracted join key and the tagged record as a (key, value) pair.

- The output is partitioned, merged, and sorted by the framework and grouped based on the join key.
- In the reduce phase, the grouped records based on the join key are given as inputs to the reducer.
- For each join key, the reducer buffers and separates the records into two sets (L and R) and performs a cross join between the records.
Problem

- All the records for a join key are to be buffered from L or R.

- What if the records are in a high number that they don’t fit the memory?
Improved Repartition Join

- First, in the map function, the output key is changed to a composite of the join key and the table tag.
- The table tags are generated in a way that ensures the records of R are sorted ahead of records in L.
- Second, the partitioning function is customized so that the hash code is computed from just the join key part of the composite key.
- The grouping function in the reducer is customized so that records are grouped just on the join key.
- We have a overhead in both algorithms, R and L should be sorted and sent over network.
Preprocessing for Repartition Join

- The shuffle overhead can be overcome if the records in L and R are partitioned based on the join key before the join operation.
- By pre partitioning both L and R before the join operation we can directly join the matching portions from L and R that can be directly joined.
- In contrast to a parallel RDBMS, it is not possible to guarantee that the corresponding partitions from L and R are allocated on the same node. This is because the DFS makes independent decisions on where to store a particular data block.
Broadcast Join

- In most applications, the reference table \( R \) is much smaller than the log table \( L \)
- Send only \( R \) instead of \( L \) and \( R \).
- Avoids sorting of \( L \).
- Avoids the network overhead of moving the bigger table \( L \) over the network.
- Broadcast join is run as a map-only job.
- On each node, all of \( R \) is retrieved from the DFS to avoid sending \( L \) over the network.
- Each map task uses a main-memory hash table to join a split of \( L \) with \( R \).
In the init() function of each map task, broadcast join checks if R is already stored in the local file system.

If not retrieves a copy of R from DFS and partitions and stores in the local file system.

Broadcast join dynamically decides whether to build the hash table on L or R.

If R is smaller than the split of L, the init() function is used to load all the partitions of R into memory to build the hash table.

If the split of L is smaller than R, the join is not done in the map function. Instead, the map function partitions L in the same way as it partitioned R. Then in the close() function, the corresponding partitions of R and L are joined.
Preprocessing for Broadcast Join

- Increasing the replication factor for R we can ensure that most nodes in the cluster have a local copy of R. This can enable broadcast join to avoid retrieving R from the DFS in its init() function.
Semi-Join

- When R is large, many records in R may not be actually referenced by any records in table L.
- Consider Facebook:
  - Its user table has hundreds of millions of records. However, an hour worth of log data likely contains the activities of only a few million unique users and the majority of the users are not present in this log at all.
  - This means that a large portion of the records in R that are shipped across the network (via the DFS) and loaded in the hash table are not used by the join.
Semi-join implemented to avoid sending the records in R over the network that will not join with L.

Semi join implementation has three phases

First Phase: A full Map Reduce job

1. Main-memory hash table is used to determine the set of unique join keys in a split of L. By sending only the unique keys to the Map output, the amount of data that needs to be sorted is decreased.

2. The Reduce task simply outputs each unique join key.

3. One reducer is used to consolidate all the unique keys into a single file L.uk. (L table with all the unique keys)
The Second Phase: of semi-join is similar to the broadcast join (L.uk is broadcasted) and is run as a Map-only job.

1. The init() function loads L.uk into a main-memory hash table.
2. The map() function iterates through each record in R if its each record and outputs it if its join key is found in L.uk.

The Third Phase: of semi-join

1. All the Ri are joined with L using broadcast join.

Although semi-join avoids sending the records in R over the network that will not join with L, it does this at the cost of an extra scan of L (in First Phase)
Preprocessing for Semi-Join

- It is possible to move the first two phases of semi-join to a preprocessing step and only execute the last phase at query time.
- For example, as new records in L are generated, the unique join keys of these records can be accumulated and joined with R to determine the subset of R that will join with L.
One problem with semi-join is that not every record in the filtered version of $R$ will join with a particular split $L_i$ of $L$.

The per-split semi-join also has three phases:

1. The first and the third phases are map-only jobs, while the second phase is a full map-reduce job.
2. The first phase generates the set of unique join keys in a split $L_i$ of $L$ and stores them in the DFS file $L_i.uk$.
3. In the second phase, the map function loads all records from a split of $R$ into a main-memory hash table. The close function of the map task reads the unique keys from each $L_i.uk$ file and probes the hash table for matching records in $R$.
4. Each matched record is outputted with a tag $RL_i$. 
• In the final phase, the file for RLi and Li are joined using the directed join.

• Compared to the basic semi-join, the per-split semi-join makes the third phase even cheaper since it moves just the records in R that will join with each split of L.
Preprocessing for Per-split Semi-join

- Like the basic semi-join, the per-split semi-join algorithm can also benefit from moving its first two phases to a preprocessing step.
- L may be joined with several reference tables on different join keys.
Multi-Way Joins

- L may be joined with several reference tables on different join keys.
- Repartition join would need to run a separate Map Reduce job to join each reference table in this case.
Experimental Evaluation (without Pre processing)
With Pre processing

(a) Referencing 0.1% of $\mathcal{R}$

(b) Referencing 1% of $\mathcal{R}$
Scalability

- We do the scalability performance test with the number of nodes L with 1% R.
- All the algorithms are scaled almost linearly.
- These algorithms naturally spawn in terms of Map Reduce functions as the number of nodes increase.
- They use partitioned parallelism.
Comparison with Pig Join Algorithms

- Repartition join and fragment replicate join in Pig.
- Resemble the Improved Repartition join and broadcast join in Map Reduce.
- Improved repartition vs Pig Repartition:
  - Pig repartition treats both input sources the same. It may have to buffer records from the larger input L.
  - Pig repartition uses a extra secondary scan in the reduce phase to guarantee that the records of same key are ordered by their input tags.
- Improved repartition join is 2.5 times speeder than Pig Repartition join.
Broadcast VS Fragment Replicate Join

- For example for 0.3 million records, Broadcast join is 3 times faster than fragment replicate.

- In Broadcast join copies of R are in every node.

- In fragment replicate, reference table is always reread from DFS.
Conclusion and Future Work

- Evaluated the performance of a 100 node system and shown unique tradeoffs by applying different joins.
- Explained how the preprocessing gives us benefits.
- Compared with different frameworks.

Future Work:
- Evaluating methods for Multi Joins
- Indexing methods to speed up join queries.
- Designing an optimization module to dynamically decide the appropriate join algorithm.
- New programming models for implementing Joins.
Thank you