Join Algorithms

ARISTOTLE UNIVERSITY OF THESSALONIKI

Distributed Data Processing
MSc Data and Web Sciences

Zisis Flokas
flokaszisis@csd.auth.gr
AEM: 73

Spring 2022

Introduction

Join Algorithms is a simple Java maven project that implements three different join algorithms over key-value pairs stored in two different Redis databases. Other than the actual implementation of the algorithms in Java, the project provides the infrastructure and methods to provision it required to demonstrate the join operations. The deployment of the Redis stores can be achieved with the use of the provided docker-compose while there is also a Docker image to deploy join operations as Docker containers.

In the next sections the join algorithms and their implementations are described in detail. Those implementations can be found under the **auth.dws.dpp.joins** package of the project. The code related to the interaction with Redis databases can be found in the **auth.dws.dpp.db** package.

Detailed instructions to use Join Algorithms can be found in the README.md file of the project. Github repository

Join Algorithms

In the next sections the implementation of the join algorithms will be presented. For each algorithm we also calculate the execution time of it. The execution time starts right after connections with both Redis databases are established and finishes right before the connections are closed at the end of the execution.

Pipelined Hash Join

Pipelined Hash Join is a join algorithm that can work in streaming fashion. It constructs two hash tables one for each input. For each input tuple the hash table of the input is updated and the hash table of the other relation is probed in order to get a possible join pair. Compared to the simple Hash Join algorithm this one requires more memory as it needs to maintain two hash tables in memory instead of one.

Implementation

The implementation of the Pipelined Hash Join can be found in the **auth.dws.ddp.joins.PipelinedHashJoin** class of the project.
The algorithm starts by setting up the connection with the two Redis databases and then for each one of them a hash table is initialized.
In order to simulate the stream of tuples from both databases we perform the scan operation by initializing the cursor with value 0. Scan operation returns an iterator with an arbitrary number of keys which by default is 10, for this reason we store this iterator using the Relation class and perform next() method to get a key at a time. Two objects of type Relation are initialized one for each Redis store using the cursor value, the iterator of Redis keys and a null LatestKeyValuePair key-value pair. Relation class objects are used to update the state of the relation after each iteration of the algorithm providing some abstraction over the Jedis API. The algorithm can start consuming keys from any of the Redis stores, which is regulated through the startFromRelation1 boolean variable.

While both relations are not fully explored we make a loop and at each step we consume a key from each relation every time (by changing startFromRelation1 variable’s value). When a key is consumed, first we perform the **updateHashTable** function which updates the input hash table and returns a Relation object with the latest state of the relation. Then the **probeAndJoin** function is used which fetches the latest key-value pair inserted in the input hash table and probes the key to the other hash table in order to find a possible join pair. If such a pair is found it is printed in the console in the following format.

key: (value of input relation, value of probed relation)

When a relation is fully explored which means cursor 0 has been reached again and there are no items left in the current keys iterator then a new loop continues to consume the other relation until it is also fully explored. If both relations are fully explored then the program terminates.

Semi Join

Semi Join algorithm is a special case of join where the keys of the smallest relation (in terms of number of tuples) are retrieved and moved to the bigger relation in order to query it using the join keys to find the join pairs. As an operator Semi Join returns only the schema of the left relation for the join keys that it has in common with the right relation. In this implementation for the common keys we return the values of both key-value pairs as we do also in the other join algorithms. Semi join will save us a lot of computation time in cases where the small relation is relatively small or way smaller than the big one.

Implementation

The implementation of the Semi Join can be found in the **auth.dws.ddp.joins.SemiJoin** class of the project.
Again we start by setting up connections with the two Redis stores. Then **getKeysOfSmallRelation** method is used to first determine the number of keys that each redis store has using dbSize() method implemented in Jedis client and then we return the keys of the smallest one (if there is one, in case they have the same size returns the keys of redis1). We loop through the keys and for each one of them we query the two Redis stores. In case both queries return a non null value then a join pair is emitted in the console in the same format as described in Pipelined Hash Join.

Intersection Bloom Filter Join

Intersection Bloom Filter Join is a join algorithm that makes use of bloom filter probabilistic data structure to introduce intersection bloom filter which is the intersection result of two bloom filters each one populated with the join keys of each relation. This way we can probe the intersection bloom filter and quickly exclude keys from join candidates when they are not found in it. In case a key exists in the intersection bloom filter we are not 100% sure that it will produce a join pair, we need to further check if it exists in both relations. Bloom filters are also efficient in terms of memory usage as they only store bits.

Implementation

The implementation of the Intersection Bloom Filter Join can be found in the **auth.dws.ddp.joins.IntersectionBloomFliterJoin** class of the project.
Again we start by setting up connections with the two Redis stores. Then the keys of both Redis stores are retrieved and we also compute their union and store it in a third list. Using the BloomFilter implementation of Google's guava library we create a BloomFilter for each set of keys. To do so we first initialize a BloomFilterConfig config for each BloomFilter we will create. The BloomFilter constructor requires specifying the number of estimated inserts which is set to the length of the keys list and a false positive probability which refers to the probability of collision of the bits in the bloom filter when hashing the keys. We set this to 3%, smaller false positive probability means more hash functions will be used. Furthermore we initialize an intersection BloomFilter which has expected insertions equal to the sum of keys from both relations.
First we populate the two bloom filters with each relation's keys using **generateIntersectionBF** method. The two output bloom filters are used to construct the intersection bloom filter. There is no method provided by guava library that implements the intersection of two bloom filters so we generate it by iterating through the keys of one relation and checking if each key exists to the bloom filter of the other. In case it does then this key is inserted in the intersection bloom filter as it is a join candidate, if it doesn't we skip that key it means that it is not a join candidate. The same process is done for both relations to finally get the intersection bloom filter using **findPossibleCommonElements** method. At last we iterate through the union of the two key lists (which provides all keys uniquely) and we probe the intersection bloom filter. If a key doesn't exist in the intersection bloom filter it certainly doesn't make up a join pair, in case it exists we need to query both Redis stores with it and if both values are non null we emit the join pair in the console.

Note
Generating union of keys (all unique keys between both relations) can be considered a non feasible decision as it makes the implementation more memory intensive (by storing this third list in memory) but we improve on compute time as we do not need to double check for the mutual join keys.

Experiment

In order to compare the three different algorithms in terms of performance we conduct an experiment. We populate both Redis databases using **generateKeyValuesPairFromList** method of DataIngestor which creates a pool of 50k keys and for each Redis store randomly picks 10k of them and ingests them along with random values. From this process each Redis store had 10k keys where 2078 of them were common (this was checked after performing a join operation and reviewing results). For this setup the execution time for each algorithm was the following.

- Semi Join: 2166ms
- Pipelined Hash Join: 2784ms
- Intersection Bloom Filter Join: 1176ms

Intersection Bloom Filter Join seems to be by far the fastest one. Also, this experiment setup doesn’t favor Semi Join as both relations have the same size while Pipelined Hash Join is the slowest of them all. It would be interesting to compare memory usage of these algorithms over computation time to have a better overview. Computation time alone cannot be the factor to nominate a join algorithm as the best because they have different unique characteristics to perform best in different scenarios.