**Parallel and Continuous Join Processing for Data Stream**

Hello everyone, my name is Song Ge, I am a Ph.D. candidate from MICS library of CentraleSupelec University of Paris-Saclay. Thank you all for coming to my Ph.D. oral defense. I hope that you will enjoy my presentation.

My dissertation title is Parallel and Continuous Join Processing for Data Stream. And this is a large topic, so I divide it into two parts. The first part is Data Driven Stream Join which we take the algorithm of finding the nearest neighbors as our use case. And the second part is Query Driven Stream Join, for this one we take Semantic Web as an example. And I will have a short introduction about the background of big data and data stream at the beginning. And a conclusion and future research directions at the end.

We are all familiar with the words “Big Data”, because this is one of the hottest topics in the IT industry for the past 5 years. But how big is the data? Let’s have a short review. Google processes 24 PB of data every day. Facebook has 10 million photos and 3 billion “likes” each data. YouTube has 800 million visitors every month. And Twitter is doubling its size every year. So the most significant issue for processing these data comes from the size of data. And the flip side of size is speed. Normally, these data can not be processed in a single machine, so it often requires a parallel and distributed solution. But the network communication in the parallel and distributed processing system may cause a bottleneck of the system. The last issue is the dynamic of data. Data is changing all the time, and usually the newest ones have the most useful values. So the best way of describing data is data streams.

Different from a persistent static relation which requires a batch oriented data processing, the transient dynamic data streams require a real-time stream processing. In order to provide a stream processing, we need to be able to add or remove computational nodes based on the current load in the architecture level; and I also want to be able to withdraw old results and take new coming data into account in the application level.

The purpose of this dissertation is to study about the process for join operation in a parallel and continuous way for data streams. Join is a very popular and often used operation in the big data area. And when the data becomes large, we usually require a parallel join. According to the two different types of parallelism, we divide the joins studied in this dissertation into two different types, the first one is Data Driven join based on Data Parallelism and the second one is Query Driven join based on Task Parallelism.

For the data driven stream join we will take the kNN algorithm as our use case. This is the outline of this part. I would like to start with a short introduction about kNN. Given a set of query points R and a set of reference points S, a k nearest neighbor join is an operation which, for each point in R, discovers the k nearest neighbors in S. This algorithm is often used for classifications and classifier, it has many applications in social network such as finding similar users, recommendation systems like finding the nearest restaurants, or search engines etc.

The reason we call it a Data driven join is because that the query never changes. But the format of data does, especially the dimension of data. For example, in our experiences, we have used GPS data which is in 2 dimensions, twitter data which is in 77 dimensions, and also images data which is in 128 dimensions, etc.

The basic idea to solve this problem is to adopt a nested loop. The idea is straightforward, it is for every element in R calculate the distance in S, the complexity for this step is O n squired. Then sort the distances for each element in R to find the top k smallest ones. The step of sort often has a log complexity, for example log(n) for priority queues and nlog(n) for quick sort or merge sort.

This algorithm is both data intensive and computation intensive. So when the data grows bigger, one machine can not execute it and return the results in a reasonable time. That is why a parallel workflow is required.

We divide the parallel workflow into 3 steps. The first step is data preprocessing, this step is used to reduce the dimension of data, or to select the central points of data clusters.

The strategies of reducing the dimension of data rely on the projection of the high dimension data into a low dimension space. We introduced two methods in this dissertation. The first one is called Z-Value which is a kind of space filling curve. The z-value is a value in one dimension. It is calculated by interleaving the binary representation of the coordinates to form a new number. The Z in this figure indicates the neighborhood after the projection.

Another projection strategy is called LSH which is short for Locality Sensitive Hashing. This method uses several hash functions to map the neighbor data into the same bucket with a high probability. The bucket is represented by rectangles in this figure. And this example used 4 hash families H1, H2, H3 and H4.

The other thing which can be done in the preprocessing step is to select the central points of data clusters. We call them the pivots. This selection will be further used for data partitions. We introduced 3 selection methods. The random selection generates a set of samples, then calculates the pairwise distance of the points in the sample. The sample with the biggest sum of distances is chosen as the set of pivots. The furthest selection method randomly chooses the first pivot, and calculates the furthest point to this chosen pivot as the second pivot. And repeat this step until having the desired number of pivots. The k-mean selection applies the k-means method on a data sample to update the centroid of each cluster as the new pivots until the set of pivots stabilizes.

The second step is data partitioning. In this step, we have distance based partitioning strategy and size based partitioning strategy. Data partition is a very important step, because it affects the performance especially for the amount of data transmitted through network. If we don’t use any advanced partitioning strategy, and just partition data randomly, we will waste a lot of resources. For example, like in this figure, we randomly partition R and S, and distributed R on rows, and S on columns, then in order to make each subset of R calculate with every subset of S, we need to replicate both partitions of R and S n times, which leads to a total number of n square tasks for calculating the pairwise distances. This replication leads to low efficiency of performance.

So the key to improve the performance is to preserve spatial locality of objects when decomposing data for tasks. More precisely, what we want is: for every partition Ri, find the only corresponding partition Sj, where the kNN join of Ri and S equals to the kNN join of Ri and Sj. And at the same time, we also need to make sure that the kNN join of R and S, is the union of the kNN join of all the partitions of R and S. So after using the advanced partitioning strategies, we can reduce the number of partitions from n square to n.

The first type of advanced partitioning strategy we want to introduce is the Distance Based Partitioning Strategy. This strategy wants to have the most relevant points in each partition. To use this strategy, we first select the pivots in the query point R, which has been done in the preprocessing step. We then partition R using these pivots. To partition R, we just calculate the distance between each point and the pivots. And the point will be sent to the partition of the nearest pivots. Then an upper bound is created for each partition, by using the distance between the pivot and the furthest point as the radius to draw a cycle. Then reference points from S which drop inside the upper bound are considered as the corresponding partition of S.

The second type of advanced partitioning strategy is size based partitioning strategy. We can use z-value to achieve this strategy. As we presented, z-value is a one dimensional value. In order to partition data, we first need to do a sample of R. We then find the quantiles in this sample. And these quantiles are considered as the boundaries of each partition in R. The corresponding partition in S consists of 3 parts. The first part is the reference points inside the boundaries of this partition. But this is not enough, in order to make sure every point in the partition of R has enough kNN candidates, we need also to copy the k previous points and the k succeeding points into the partition of S.

We can also use LSH to do size based partition. In this method, we also first partition R, and have the buckets of R. Then we use the same hash families to map S into the buckets. And also the boundary points need to be replicated in each bucket in order to make sure that each point has enough kNN candidates.

You may notice that, in the advanced partitioning strategies both distance-based, and size-based, we all first partition R, and find the corresponding partitions of S for each R. Please remember this, I will explain later why we do it this way.

The third step of the workflow is computation. We can use only one job to directly get the global results. We can also use two consecutive jobs, to first get the local results, then merge the local results to have the global ones. The purpose of using multiple rounds of jobs is to reduce the number of elements to be sorted.

Let’s see an example. In this example, we want to find the nearest city in S for each city in R. So R is our query set, and S is the reference set. If we use only one job. We need to calculate the distance in the map stage, and sort these distance to have the nearest city in the reduce stage. The number of cities to be sorted is 4 for each city. This number is the size of the reference size S. So, when S is large, we can not load all these elements into memory. That’s why in we want have two jobs. Because in the first job, we can sort the local results, and in the second job, we merge the local results, and sort them to have global ones. So we have two sort stages, but each stage we have less elements to sort.

This figure shows the workflow, we first give the raw data to the preprocessing step in order to select the pivots or to reduce the dimension. We then use partitioning strategy to partition data into distance based partitions or size based partitions. And finally we calculate the kNN join in either one round of job or 2 rounds of jobs.

Next, I want to talk about the theoretical analysis in the aspect of load balance, accuracy and complexity.

First, what is load balance. Load balance wants to make each task process the same amount of data. Because in a parallel processing, the processing time is execution time of the longest task. In our case, we want to have this which indicates the same amount of computation. But this is difficult to achieve. So we can accept a sub-optimal option, that is, if we have either partitioned R into equal sized or S into equal sized, we think that we can achieve load balance. But which one is better? Let’s calculate the worst case complexity. If we have partitioned R into equal sized, then in the worst case, we may need to have the whole S for each partition of R. The worst case complexity is this. And if we have partitioned S into equal sized, then in the worst case, we may need to have the whole R for each partition of S. And the complexity is this. As the number of partition is far smaller than the size of S. So the first one is better. Remember that in the advanced partitioning strategies, we all first partition R, then find the corresponding S for each R. This is the reason.

In order to process stream in a sliding window, we need to design the re-execution strategy and the data invalidation strategy. Both have egger manner and lazy manner. In the eager manner, the sliding window generates new results and moves forward upon arrival of new data. In the lazy manner, it re-execute new data and removes old data periodically. The lazy one is more practical in real applications. So we choose to use the lazy manner. And the period of re-execution and expiration is called generation.

In the advanced method, we want to only have n tasks for each generation. But we only want to partition the new data items without moving the old ones. So suppose we already have several partitions for the first generation. Then, we use naïve bayes theory to partition the new generation. We calculate the conditional probability of the new coming data be partitioned to each partition. And the data will be sent to the partition with the highest conditional probability.

The reason we call it a query driven join is because that the data format does not change, but the query does. Each query is a user defined search.

There are 4 different processing manners for RDF join. The first one is a centralize one. But as the data grows bigger, we need some parallel solutions. But as we explained, the network communication may be a bottleneck of the parallel system. So in order to minimize the communication, several solutions propose to only distribute the data, but give each sub set of data the whole query to process. And some other solutions propose to not distribute data but replicate the whole data set several times, and decompose the query into sub queries for each replication of data. But these two methods both have problems. For example, the first one will stop working when the query is too complex. And the second one may not work if the data set is too big to be stored in a single machine. So both distribute data and decompose query is the best solution. And our proposal adopts this manner.

Data partition is also an important method for this process. There are two different types of partitioning strategies. The first one is based on the vertex partitioning methods for graphs. But due to the high overhead of loading the big graphs, this one is not good for big data. Besides, it is not suitable for stream processing neither, because it requires the entire information in order to make decision which we can not get in a stream environment. Another method is hash partitioning based on indexes. This one is also difficult to be reused by data streams. Because, look, since there are 3 different parts in an RDF triple, we need to index every combination of these parts, which leads to a total number of 15 indexes. This is both time and space consuming, which is not good for stream processing.

The general step of processing RDF streams should be, first partition RDF streams, and distributed data. Then decompose the queries into sub-queries and assign these sub queries to the appropriate nodes. Finally, we need to rapidly reply to the changes of data and return the results in real-time.

In our proposal, we first decompose the queries into triple patterns, and send each triple pattern to different machines.

We use an advanced data structure Bloom Filter to reduce the communication overhead among nodes. Bloom Filter is often used for membership queries. In our proposal we do not send triples, but we send Bloom Filters. The physical structure of a Bloom Filter is bit vectors. This bit vector is initially set to 0. To form a Bloom Filter, we also need to use several hash functions. For example, in this Bloom Filter, we have 20 bits, and we use 3 hash functions. Then we want to insert 3 elements. To insert an element, we need to apply the hash functions to this element. And we change the corresponding bit of the hash value from 0 to 1. When we want to check whether an element is contained by the Bloom Filter, we also need to apply these hash functions to this element. And if all the bits are 1, we consider this element to be contained by this Bloom Filter. And if at least one bit is not 1, then this element is not contained by the Bloom Filter. But due to the hash collisions, sometimes, we can have some false positive results. For example, this one will be considered as belong to the Bloom Filter, while it does not. The probability of having false positive results is a function of n, k and m.

According to the different operation on Bloom Filters, we divide the triple patterns into Builders and Probers. The triple patterns used to insert the elements to the Bloom Filters are called Builders. And the triple patterns used to check the existence in the Bloom Filters are called Probers. Question, which triple patterns should be Builders and which ones should be Probers?

To solve this problem, we create 3 rules based on the structure of the query.

Rule 1 is for 1-variable join. The 1-variable join only has one variable, which is the join place. So the triple pattern consist this join can be considered as having the same status. Then we can choose any one of them as the Prober, and the others as the Builders.

Rule no. 2 is for 2-variable join. The 2-variable join has two variables. One more besides the join place. Since the Bloom Filter can only carry the information of the join place, if the triple pattern with 2 variables has been chosen as a Builder, then the information of the other variable will be lost. So for this type of join, we need to choose the triple pattern with two variables as the Prober, and the other triple patterns as the Builders. The other process is similar to the 1-varialbe join.

Rule no. 3 is created for multiple-variable join. A multiple variable join has more than 2 triple patterns with 2 variable parts. As we explained, the triple pattern with two variables need to be chosen as Probers for keeping the information of the other variable parts. But at the same time, they also need to be Builders, in order to communicate its information of the join place with the other triple patterns. So they need to be both Builders and Probers. Which means they need to both send Bloom Filters and Receive Bloom Filters. Another question: what is the sending and receiving order?

To determine this order, we create the last rule, which is called Query Topological Sort. Query Topological Sort is a topological sort for the query graphs, where the constant nodes on the graph have higher priority than the variable nodes at the same level.

Let’s see an example, this is our query. We then omit the predicates, because they are useless. The topological sort of a graph determines the dependencies order of this graph. This is the results of the query topological sort. The dependencies of a node are all the nodes before it. So in order to execute a variable node and send its results, we need to receive all the information of its dependencies.

We then extend this method for processing continuous join of RDF streams using the sliding window model and the sliding bloom filters. We re-use the rules we explained before for each generation of data. We send the Bloom Filters after each generation, and keep them in the prober side. The probers need to check with the Bloom Filters for the current window with the new results of the current generation.

We then implement our query engine on Apache Storm. Storm is a parallel and distributed stream processing system. A job on Storm is called a Topology. The topologies consist two different components. The spout and the bolt. The Spouts are used to distribute resources, and the Bolts are used to process data. Take the previous query as an example, we need to use 3 Spouts to receive the triples for these 3 predicates. Then we have two different types of Bolt, the Builder Bolts and the Prober Bolts. The streams are passed to the corresponding Bolts. The parallelism of each Bolts is determined by the number of triples it needs to process, and we can dynamically adjust this number.

In the end, we also run several experiments to evaluate the system. The experiments are run on Grid 5000, with 11 nodes. We use Apache Jena to read RDF triples.

We evaluated both synthetic data and LUBM benchmarks. In the synthetic data, we randomly generate RDF triples in the Spouts. Two sub-sets of data are generated, the first sub-set contains only the results of the join, and the second one contains only the non-results of the join. We do this in order to evaluate the accuracy.

LUBM is a benchmark contains 14 queries, we evaluate 3 of them, query No.1 is the 1-variable join, Query No.3 is the 2-variable joins, and Query No.4 is a multiple variable join. It consists of a university domain. The data can be customizable and repeatable.

We consider two impacts, they are the sliding window size and the number of generations. And we record 4 metrics, the execution latency which is the difference between the time an element is generated and the time it is emitted as a result. The process latency, which is the difference between the time an element is generated and the tie it begins to be processed, the amount of data transmitted through network, and the accuracy.

This is the result of the 1V join. In this experiment, we fix the sliding window size at 800, we then change the number of generation from 2 to 8, and we compare the execution latency. We can see that this latency decreases as the number of generation increases, because then we have less elements to process in each generation.

In this experiment, we fix the number of generation at 6, we then change the sliding window size from 200 to 800, and we record the execution latency. We can see that the execution latency increases when the sliding window size increases, because we have more elements to process in each generation. This is the data transmission for 1 variable join and multiple variable join. We can see that multiple variable join has larger communication overhead than 1 variable join, because it transmits more Bloom Filters. But the highest communication overhead is only 1 KB. This overhead will be 400 times more without using Bloom Filters.

For the accuracy, we got 100% correct results which is a surprise. Because we use the false positive rate, the number of element to be inserted to determine the number of bits in the Bloom Filter. The number of elements is set to the number of elements received by the triple pattern, while the real number should be the number of results after matching the triple pattern which is much less. So the real false positive rate is ignorable.

**C**

**B**

**A**

**Z(R)**

**Z(S)**

**K Previous**

**K Succeeding**

R

S