Implementation and comparison of join algorithms for SPARQL query processing

Ahmet Cemal Alıcıoğlu[1] and Mert Çıkla[1]

Dept of Computer Science, University of Freiburg, Freiburg im Breisgau 79110, Germany

1 Problem Description

SPARQL is a query language designed for querying data on the Semantic Web. As the volume of RDF (Resource Description Framework) data continues to grow, efficient query processing becomes essential. [4] This project explores the implementation and comparative analysis of various join algorithms for SPARQL queries, aiming to optimize query performance.

In this project, we implemented some of the most popular join algorithms: Sort-merge join and Hash join algorithms. We then propose an improved algorithm that has better performance than both algorithms. Finally, we benchmarked each algorithm in terms of their performance. These benchmarks include executing identical queries on the same dataset using each algorithm.

In this report, we detail the algorithms implemented as part of this project, present the benchmark results, and conduct an analysis of the outcomes. Furthermore, we aim to detect the strengths and weaknesses of each algorithm, shedding light on their respective performance characteristics.

2 Algorithms

Here, we present the join algorithms featured in this project; namely sort-merge join, hash-join, and hash-partitioned joins.

We implemented the algorithms described here in Java. The source code of this project is available at https://github.com/Ciklamert/JOINSPARQL.

2.1 Sort-Merge Join

The sort-merge join involves a two-stage process. Initially, the two tables are sorted based on their join attributes. Subsequently, both of the relations are scanned in the order they were sorted, and tuples that meet the join condition are merged and added to the output table. [3]

For tables $T1$ of size m and $T2$ of size n, the sort-merge join is executed as shown in Algorithm 1

Algorithm 1 Sort-Merge Join

```
procedure SORT-MERGE JOIN(T1, i1, T2, i2)
    sort T1 based on its join attribute i1                    ▷ First Stage
    sort T2 based on its join attribute i2

    k ← 0                                                     ▷ Second Stage
    l ← 0
    Initialize 2D List T3 ← ∅
    while k < m & l < n do
        v1 ← T1[k][i1]
        v2 ← T2[l][i2]
        if v1 < v2 then
            k ← k + 1
        else if v1 > v2 then
            l ← l + 1
        else
            l_old ← l
            while l < n & v1 = v2 do
                Merge T1[k] and T2[l] on their join index and add to the table T3
                l ← l + 1
                v2 ← T2[l][i2]
            l ← l_old
            k ← k + 1
    return T3
```

The runtime of the sort-merge algorithm, as given in Algorithm 1, is $O(m\log(m) + n\log(n))$. It should be noted that the primary bottleneck for its speed lies in the initial stage of the algorithm. Sorting cannot be accomplished in linear time, whereas the second stage operates at a more efficient $O(m+n)$. Therefore it is best to focus on the first stage for performance improvements as it is the bottleneck of the computation.

2.2 Hash Join

The hash-join algorithm makes use of hash-maps to efficiently join two tables. It includes two phases: build and probe. The build phase consists of building a hash-map from one of the tables. This hash-map maps the values on each element's join attribute to the element itself. This build phase allows the program to later quickly find all elements which have a certain value in its join attribute. The probing phase after building is a simple process of scanning through the table that was not used to build the hash-map and matching the elements there. [3]

For tables $T1$ of size m and $T2$ of size n, the hash join algorithm is shown in Algorithm 2.

Algorithm 2 Hash Join

```
procedure HASH JOIN(T1, i1, T2, i2)
    Initialize 2D list T3 ← ∅
    Initialize empty hash-map H that maps integers to 2D lists

    for E1 ∈ T1 do                                                    ▷ Build
        Add E1 to H[E1[i1]]

    for E2 ∈ T2 do                                                    ▷ Probe
        for E1 ∈ H[E2[i2]] do
            Merge E1 and E2 on their join index and add to the table T3
    return T3
```

The hash join algorithm shown above has the worst case complexity $O(m \cdot n)$. This is because the run time of this algorithm highly depends on the efficiency of the hash function used and the number of cache misses. However, if we can assume minimal cache misses and average $O(1)$ time retrieval from the hash-map, the average time complexity of hash join becomes $\Theta(m + n)$. This makes hash join algorithm asymptotically faster than the sort-merge join if and only if retrieval is done in constant time. Consequently, it is best focusing on minimizing the cache misses to enhance performance in this algorithm.

During the implementation of hash join, in order to minimize cache misses, the smaller table was selected to build the hash map; and the larger table was scanned. Some other performance improvements on the hash join can also be made by improving the hash function or hashing scheme, and partitioning the data before joining, which is implemented in the parallel partitioning hash join algorithm.

2.3 Parallel Partitioning Hash Join

We have decided to implement the parallel partitioning hash join algorithm as an improvement algorithm. The parallel partitioning hash join algorithm starts by partitioning both tables on the same hash function. It then hash-joins all partitions of one table by their counter-parts (i.e partition of other table with the same hash value). The resulting table is then the union of the result of every mini hash-join procedure. [3] An important part of this algorithm is to run the hash-join procedures parallel in different threads. This way, each thread is responsible for a smaller and independent part of the procedure and this will result in fewer cache-misses and better CPU efficiency. Basically, we applied non-blocking partitioning on private partitions for each thread. [2]

The pseudocode description of the parallel partitioning hash join algorithm is presented in Algorithm 3.

Algorithm 3 Parallel Partitioning Hash Join

procedure PARALLEL PARTITIONING HASH JOIN JOIN($T1, i1, T2, i2, num\_partitions, hash$)
  Initialize 2D List $T3 \leftarrow \emptyset$
  Initialize 3D List $P1 \leftarrow \emptyset$
  Initialize 3D List $P2 \leftarrow \emptyset$

  for $E1 \in T1$ **do** ▷ Partition
    calculate $h \leftarrow hash(E1[i1])$
    Add $E1$ to $P1[h]$
  for $E2 \in T2$ **do**
    calculate $h \leftarrow hash(E2[i2])$
    Add $E2$ to $P2[h]$

  Parallelly run hash_join($P1[i], i1, P2[i], i2$) for $i \in [0, num\_partitions]$ and extend to $T3$ the results.
  return $T3$

During the implementation of the parallel partitioning hash join algorithm described above, we have used a very simple hash function $hash(x) = x \bmod num\_partitions$.

This algorithm is a slightly improved version of hash join algorithm and therefore has the same asymptotic time complexity analysis.

3 Dataset

The datasets we have used to benchmark the algorithms presented in this report are two selected datasets from The Waterloo SPARQL Diversity Test Suite WatDiv dataset [1].

These datasets contain RDF triples, where each triple consists of three components: object, predicate, and subject; where the middle one indicate the relation between the other two entities. RDF is very common for representing data in a graph format, as using these subject-predicate-object expressions efficiently describe the relationships between entities.

The size of the first selected dataset is 100 thousand triples, while the second selected dataset comprises 10 million triples. As a comparison, the second dataset is 100 times larger than the first one.

To make the algorithms run faster, we have preprocessed the filtered data. This preproccessing includes encoding the values in strings into 32-bit unsigned integers. Since the integer comparison is a faster process than string comparison, this has resulted in an overall increase in performance.

A string can be divided into two sections, **prefix** and **id**. The **prefix** determines its entity type, while **id** differentiates between the elements of the same entity (e.g for "User760", prefix = "User" and id = 760.)

Since there are three different types of prefixes in this query, the above formula uses the first two bits of the unsigned integer to differentiate its prefix, and uses the remaining 30 bits for the id. The mapped prefixes are User (1), Product (2), and Review (3).

Finally, we use the following function for encoding:

$$encode(\text{prefix, id}) = code(\text{prefix}) \cdot 2^{30} + \text{id}$$

where *code* is the mapping of the prefixes to 2-bit numbers.

4 Experiment and Analysis

In our program, we have measured the results on the first dataset which includes 100 thousand triples. However, we could not calculate the results for the larger dataset with 10 million triples due to memory issues; as the result of the given query for this dataset is more than 100 GB.

We have conducted the benchmarks on a device with **AMD Ryzen 7 5800H** CPU (8 cores, 3.2 Ghz) and 16 GB RAMs. The heap space allocated For JVM (Java Virtual Machine) was 8 GBs and a total of 16 threads were used.

To test the performance, the three programs are given the following join query to compute:

follows $\bowtie_{follows.obj=friendOf.sub}$ friendOf $\bowtie_{friendOf.obj=likes.sub}$ likes $\bowtie_{likes.obj=hasReview.sub}$ hasReview

The above query was computed 100 times by each algorithm to calculate the average runtime. The results are shown in Table 1 and Figure 1.

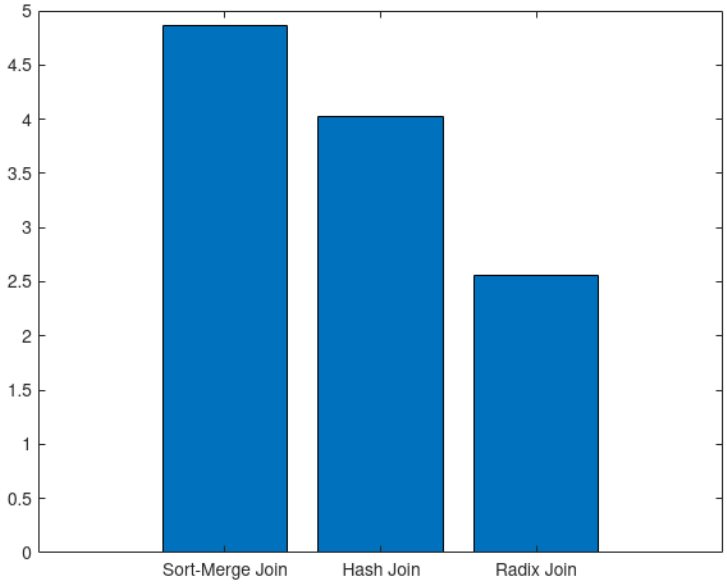

Fig. 1. Average runtime of each algorithm over 100 runs

Algorithm	Runtime (seconds)
Sort-Merge Join	4.86
Hash Join	4.02
Parallel Partitioning Hash Join	2.59

Table 1. Average runtime of each algorithm over 100 runs

Here, the experiments showed that the parallel partitioning hash join is 55.2% faster than hash join, and 87.6% faster than sort-merge join. One observation here is that the runtime difference between parallel partitioning hash join and hash join is greater than that of hash join and sort-merge join despite the asymptotic time difference. We believe this is the result of the performance enhancement of partitioning and parallel computing on a multi-core CPU having a larger effect than saving a factor of $O(n/log(n))$ for this relatively small dataset. We therefore think this would not be the case for the larger dataset of 10 million triples.

5 Conclusion

After exploring and implementing various join algorithms, and conducting a comparison of their performance, this project found clear and insightful results. We have selected to propose the parallel partitioning hash join algorithm for an "improvement algorithm," where we partition the tables before joining and use multiple threads for parallellization of the work with multi-core CPU. This improvement we described and implemented proved to be considerably more efficient than the other two algorithms.

While memory limitations prevented us from testing the algorithms on a larger dataset, we are confident that employing the parallel partitioning hash join on such data would yield even more promising results in comparison. Moreover, we believe that, with more small performance improvements, this algorithm has potential to evolve into the efficient join algorithm that is needed to handle the challenges with growing datasets.

References

1. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, Khuzaima", e.P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C., Vrandečić, D., Groth, P., Noy, N., Janowicz, K., Goble, C.: Diversified stress testing of rdf data management systems. In: The Semantic Web – ISWC 2014. pp. 197–212. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-11964-9_13
2. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core cpus. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. p. 37–48. SIGMOD '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1989323.1989328
3. Mishra, P., Eich, M.H.: Join processing in relational databases **24**(1) (1992). https://doi.org/10.1145/128762.128764
4. Valle, E.D., Ceri, S.: Querying the Semantic Web: SPARQL, pp. 299–363. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-540-92913-0_8