Advanced Databases and Information Systems Project II

Implementation of Join Algorithms for SPARQL Query Processing

Sven Pfitzer, David Eckel
5509091, 5556445

July 26, 2023

Contents

1

Problem Statement

This project focuses on the implementation of the two most frequently used join algorithms Hash Join and Sort Merge Join for a specific SPARQL query. These two methods are implemented in the programming language Python and then compared based on their runtime.

SPARQL is a RDF query language which allows manipulating and retrieving data stored in RDF (Resource Description Framework) format, which is the case for this project. The RDF format is generally used for describing graph data with nodes, edges and properties instead of standard tables. This is especially useful when describing relations between several datastores.

In the setting of this project the two previously mentioned join algorithms along with the yannakis algorithm for speed up were used for calculating the output of the following SPARQL query:

```
SELECT ?a, ?b, ?c, ?d, ?e WHERE {
?a follows ?b . ?b friendOf ?c .
?c likes ?d . ?d hasReview ?e
}
```

Listing 1.1: SPARQL Query

This can also be expressed in standard SQL form:

```
SELECT follows.subject, follows.object, friendOf.object,
likes.object, hasReview.object
FROM follows, friendOf, likes, hasReview
WHERE follows.object = friendOf.subject
AND friendOf.object = likes.subject
AND likes.object = hasReview.subject
```

Listing 1.2: SQL Query

The following sections will focus used join algorithms, the used dataset and experiments run with these two.

The code for this project can be found on Github.

2

Algorithms

2.1 Hash Join

The Hash Join algorithm is used for joining two relations R and S by calculating a hash value for the join key. It can generally be divided into the two following phases:

- Build Phase: Based on one of the two relations R and S, a hash map partition over the join key is constructed. The pseudo Python code implementation of this is shown in the following snippet:

```
    hash_table = HashMap()
    for item in R:
        hash = self.hash_function(item.join_key)
        hash_table.insert(hash, item)

```

Listing 2.1: Hash Join Build Phase

- Probe Phase: The left over relation of the join is probed over the hash map based on the join key. If the hash of the join key matches with one in the hash map, the data stored in the map gets added to the output. Hereby one has to account for possible hash collisions, which means the data stored under the selected hash key has to be compared with the actual join key to be sure it is the same. The pseudo Python code for this is again illustrated below:

```
    output = []
    for item in S:
        hash = self.hash_function(item.join_key)
        matches = hash_table.get(hash)
        for match in matches:
            if item.join_key == match.join_key:
                output.append((item, match))

```

Listing 2.2: Hash Join Probe Phase

The exact implementation of this method differs slightly due to performance reasons. The theoretical runtime complexity of Hash Join is $O(n+m)$ with $n = |R|, m = |S|$.

2.2 Sort-Merge Join

The Sort-Merge Join algorithm performs a join over two relations R and S by sorting the relations based on the join keys. Similar to Hash Join the algorithm can be divided into two phases:

- Sorting Phase: Sort the two relations R and S by their respective join attribute. The pseudo Python Code is shown below:

```
    sorted_R = sorted(R, key=R.join_key)
    sorted_S = sorted(S, key=S.join_key)

```

Listing 2.3: Sort-Merge Join Sort Phase

- Merge Phase: Merge the two sorted relations R and S based on the join condition. This can be done using the following pseudo Python code:

```
    output = []
    r, s = 0, 0
    while r < len(sorted_R) and s < len(sorted_S):
        item_R, item_S = R[r], S[s]
        if item_R.join_key == item_S.join_key:
            output.append((item_R, item_S))
            r += 1
            s += 1
        elif item_R.join_key > item_S.join_key:
            s += 1
        else:
            r += 1

```

Listing 2.4: Sort-Merge Join Merge Phase

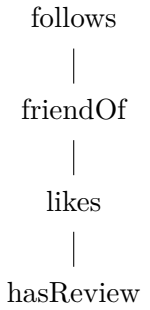
As for the Hash Join Algorithm, the exact implementation used differs slightly to increase performance. The theoretical runtime complexity of Sort-Merge Join is $O(n * log(n) + m * log(m))$ with $n = |R|, m = |S|$.

2.3 Optimization: Yannakis Algorithm

As a runtime optimization algorithm for calculating the given query the Yannakis algorithm was used. This algorithm aims to reduce the memory needed by taking advantage of semi-joins to reduce the amount of data before actually joining the relations. This procedure works sequentially and is especially useful when the a tree can be constructed based on the join conditions. When adapting this method to the query used in this project, the algorithm can be divided into the following steps:

- Construct the join tree with relevant properties.
- For each unique property, create a table with the corresponding subject, object tuples.
- Traverse the tree from leaf to root and calculate the semi joins between current node and child node based on the join condition.
- Run any standard join algorithm (Hash Join, Sort-Merge Join) starting from the root node

The exact implementation of the yannakis algorithm differs slightly due to performance reasons. The join tree used for this specific query is shown below:



3

Dataset Description

In this report the WatDiv dataset is used. It is a RDF triple store and can be download in different sizes from *https://dsg.uwaterloo.ca/watdiv/*.

3.1 Data Structure

The WatDiv dataset stores its data as RDF triple store. This means that each entry is a triplet consisting of *subject*, *property* and *object*.
The object can be a value or an entity itself (like the subject). Therefore the dataset resembles a graph. An example of four entries is shown in Table 3.1. The left column is the subject, the middle column the property and the right column is the object.

wsdbm:User0	wsdbm:makesPurchase	wsdbm:Purchase132 .
wsdbm:User1	wsdbm:follows	wsdbm:User241 .
wsdbm:User130	wsdbm:follows	wsdbm:User987 .
wsdbm:User130	rdf:type	wsdbm:Role1 .

Table 3.1: Example WatDiv triplestore dataset entries

3.2 Data Partitioning

The triplets are partitioned into relations with the vertically partitioned approach. There each table describes the relations of a property and therefore has entries for Subject (left column) and Object (right column). The triplets from Table 3.1 would be partitioned into the relations of Table 3.2.

wsdbm:makesPurchase

wsdbm:User0	wsdbm:Purchase132

wsdbm:follows

wsdbm:User1	wsdbm:User241
wsdbm:User130	wsdbm:User987

rdf:type

wsdbm:User130	wsdbm:Role1

Table 3.2: Example WatDiv relations from vertically partitioned approach

In the implementation every entry in the dataset is handled by extracting subject and object to the corresponding property table and removing the namespace prefix (like wsdbm). Additionally the strings of Subject and Object are converted to integer values to speed up later processing.

4

Experiments and Analysis

The following chapter focuses on the different experiments run within the scope of the given query as well as the analysis of these experiments. This includes the comparison between different implementations and their improvement counterparts. Lastly the runtime of the two main algorithms of this project will be compared and the implemented improvement of the yannakis algorithm will be discussed.

4.1 Strings vs. Integer Encoding

A first important decision when implementing any of the two join algorithms is associated with the preprocessing of the data. When partitioning the textual data one must store the data as matching subject-object pairs inside a partition table. Here it makes sense to use the early described vertically partitioned approach and use one table for each property.

In this setting the most straight forward approach is to simply store the strings occurring in the input file directly into the corresponding partition table. This is also what was initially implemented and used for early experiments. Therefore when the actual join algorithm is executed, the join keys of both relations are also stored as strings which must be matched against each other. This string comparison step is obviously quite slow when compared to the much simpler integer comparison.

Due to this finding the integer conversion of subjects and objects was implemented as an improvement of the data preprocessing. Here, before storing subjects and objects into the partition tables, the subjects and objects get converted into unique integers. Meaning each unique string value maps to exactly one integer value and vise versa. In the implementation this is done by using a dictionary that keeps track of all strings and their associated integer encodings. Whenever a new string value is encountered the dictionary is extended and the string is encoded with the next higher positive integer value. This improvement significantly reduces the runtime

of both join algorithms as the it reduces the overhead of the comparison phase between the join keys. It is to note that additionally to the integer representations, a reverse form of the encoding dictionary has to be stored in order to decode the the results after the join phase.

There are two additional benefits that result from using this integer representation. As a first point integers do require significantly less memory than strings which helps with potential memory issues for large datasets. The second point is related to the sort-merge join algorithm. Here, as mentioned earlier, the join tables are first sorted by their join key before actually joining them. Sorting by integers is also noticeably faster than sorting by string values, which is why this algorithm also directly gains from this improvement.

4.2 Full Store vs. Generators

Another crucial aspect of join algorithms, especially in the case of large datasets, is the memory consumption. When joining two relations, although the join condition is being restrictive, it is quite common that multiple elements of relation two join on one element of relation one. Therefore the output of this join is often larger than the relation used as input. As a result when using multiple cascaded join conditions, as done here in the project, it is crucial to be memory efficient in the implementation to not run into any memory problems.

In an initial implementation of the two join algorithms the complete output of each join operation was stored in memory and then reused in the next join. This resulted in a fast growing output over the joins. After the last join condition, a single variable holds all information about the joined results and gets directly written to the output file. It is clear that this approach could lead to problems regarding memory consumption due to permanently storing all the output results in memory. This was also observed during early experiments in this project. For the smaller dataset with about 100k entries the algorithms worked without problem, although one could already see the increasing memory usage when looking at the RAM usage. The 10M dataset however caused bigger problems as it caused the algorithm to fail with an OutOfMemmory error. From what could be seen in the increase in RAM usage, it looks like this approach would need around 40GB of RAM to sore all results, although only 32GB were available on the local machine. This clearly shows that the approach is not well suited as the amount of output data could be even higher depending on input data and computed query.

As a result of this finding a different approach was tested which only stores the necessary data for the next join in memory and later collects each result to be stored in the output file one after another. This is implemented

by taking advantage of the tree like connected structure of the query. The left relation of the join only stores the objects to be joined on while the right relation stores the join subjects along with the corresponding objects. Whenever a join match is found the object of the right relation is stored in the output so that the output of this join can be directly used as the left relation of the next join. After evaluating all join conditions the resulting output is a set objects which can be used to collect all results. In order to get one result at a time and not store everything in memory a Python generator is used. This generator is built by reversing through all used relations and extracting the relevant subjects and object based on the results of the previous join condition. When writing to the output the created generator can be iterated over yielding the individual results for one line of the output file.

This methods leads to a massive reduction of memory usage which can also be seen when running the algorithms while monitoring the RAM usage. Even for the 10M dataset there is only a neglectable amount of memory used, indicating that even much larger datasets could be processed with this method.

4.3 Runtime Analysis: Hash Join vs. Sort-Merge Join

For analyzing the performance of the two implemented join algorithms Hash Join and Sort-Merge Join, the execution time for the different datsets was meassured and compared. In this setting the runtime of the algorithms is described solely by the duration of the actual join calculation. The preprocessing of the data as well as the output collection and file writing can be omitted for this analysis as they are exactly the same for both algorithms.

For both datsets, the 100k and the 10M one, it can be seen that the Sort-Merge Join algorithm performs better in the sense that it requires less time for calculating the join results. This is surprising as the superior time complexity of the Hash Join indicates that it should be faster for unsorted relations. This phenomenon is most likey due to a suboptimal implementation of the Hash Join algorithm, especially the used hash table. The hash map is implemented a custom class which internally uses a dictionary for storing the data under the hash index. This way of storing and accessing the data in the build and probe phase can be quite slow when compared to more sophisticated hash map implementations. On the other hand the sorting ahead of the merge phase of the Sort-Merge Join algorithm is done using the built in sorted function of Python. As this is an already optimized function this could also explain the difference in runtime.

Additionally the hash function used for the Hash Join was not deeply analyzed. Therefore a high number of hash collision could occur which would

	Hash join	Sort-merge join
100k	0.003	0.002s
10M	0.90s	0.48s

Table 4.1: Runtime for hash and sort-merge join on different sized datasets.

also slow down the method during the probe phase.

The exact runtime results of both algorithms on the 100k and 10M dataset can be seen in Table 4.1.

4.4 Runtime Analysis: Yannakis Optimization

In order to analyze the improvement in performance achieved by the yannakis algorithm, the runtime of both join algorithms is compared with and without using it.

The results of this experiment can be seen in Figure 4.1 and show significant improvement for the Hash Join algorithm as well as the Sort-Merge Join. This is due to the reason that pruning the data before calculating the join drastically decreases the amount of data to be analysed. The number of data points stored in each of the used partition tables before the join is shown as an example in Figure 4.2 once with and once without using the yannakis pruning.

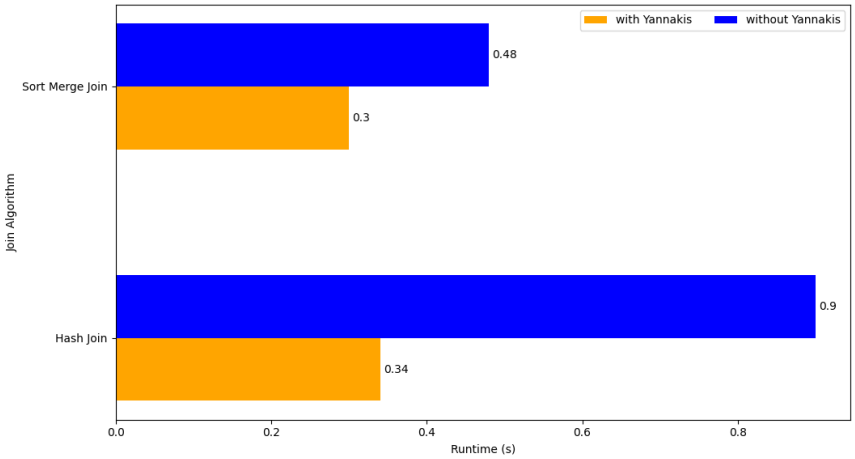


Figure 4.1: Comparison of runtime for the 10M dataset with and without yannakis

These results indicate that the yannakis algorithm is a very valid improvement for joining relations which can be used ahead of any join algo-

rithm to reduce time and memory consumption.

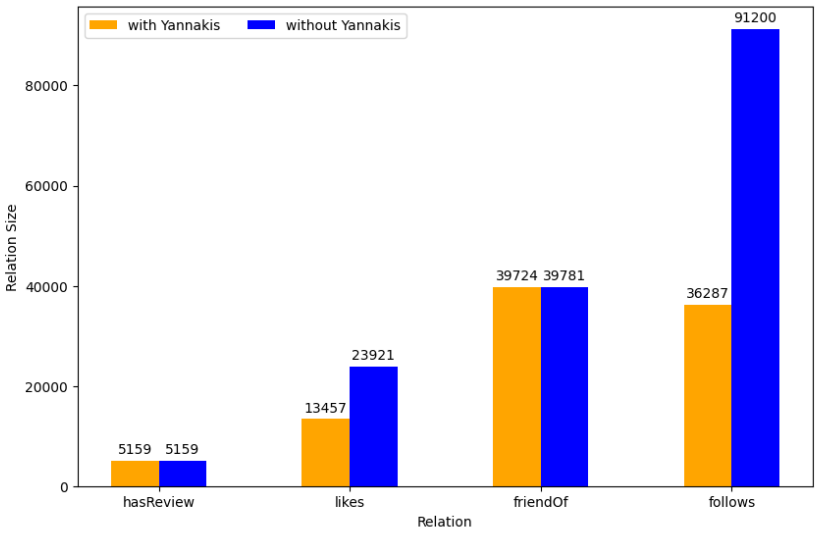


Figure 4.2: Comparison of relation sizes for the 10M dataset with and without yannakis

5

Conclusion

In the course of project the two most commonly used algorithms for joining two relations, Hash Join and Sort-Merge Join, were implemented. The correctness of the respective algorithm was verified by applying them to the query specified in the beginning of this report. As both variants yield the same output results, it can be assumed that they work correctly.

Furthermore the performance difference of the two algorithms was analyzed based on their runtime given the query. With these specific implementations and query it could be observed that the Sort-Merge Join performs better that the Hash Join. Possible explanations for this result were given in the experiments section.

Additionally different methods for calculating the query faster were implemented and tested. These included optimizations like encoding the subjects and objects of the relations as integers instead of directly using the string values, as well as using generators for collecting the results instead of permanently keeping track of them.

The most important optimization was the implementation of the yannakis algorithm, which gains in time performance by pruning the data before the actual join. This improvement can be used for both join algorithms and the time gain was analyzed on the same query.

To conclude both algorithms were implemented and analyzed successfully. Together with several optimizations they could even be improved in terms of runtime and memory consumption. Still there is more room for improvment for the implementation, especially regarding the Hash Join algorithm and its hash table.