Advanced Databases and Information Systems Project II

Implementation of Join Algorithms for SPARQL Query Processing

Omar Swelam (os132@uni-freiburg.de)
Jumshaid Khan (jk1308@uni-freiburg.de)

Professor: Dr. Fang Wei-Kleiner

University of Freiburg

Faculty of Engineering

Department of Computer Science

July 31th, 2023

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Problem Statement

The increasing popularity of Semantic Web technologies has led to the widespread use of RDF (Resource Description Framework) datasets and SPARQL queries for representing and querying knowledge on the web.

RDF datasets comprise triples representing subject-predicate-object relationships, forming a connected data graph. SPARQL queries enable users to retrieve specific information from RDF datasets by expressing patterns using variables and properties.

1.1 Problem

The task is to implement two join algorithms, Hash join and Sort-merge join, for evaluating a specific SPARQL query over RDF datasets. The goal is to efficiently retrieve the list of mapped values for the variables in the given SPARQL query.

(?a)—follows—>(?b)—friendOf—>(?c)—likes—>(?d)—hasReview—>(?e)

Efficient query processing for SPARQL evaluation on RDF datasets is relevant for practical applications and has implications for Semantic Web research and future joint algorithm innovations.

By implementing Hash Join, Sort-merge Join and Radix Join algorithm, our goal for this project is to analyse the performance of each join algorithm.

1.2 Aims and Objectives

This project aims to implement Hash join, and Sort-merge join algorithms for evaluating a specific SPARQL query on RDF datasets. The primary objective is to assess the efficiency and scalability of these join algorithms when processing large-scale RDF data.

Another objective focuses on pre-processing the RDF data using a vertically partitioned approach. The data will be divided into distinct relations, where each property corresponds to a separate table with 'Subject' and 'Object' columns. Additionally, an optional step will construct a dictionary of string values occurring in the triple store and convert them into integers to enhance the join algorithm's efficiency.

Finally, a radix join algorithm is implemented to optimize the Hash and Sort-merge join algorithms. The project will culminate in a comprehensive report documenting the implementation process, join algorithm analysis, and optimization results, contributing valuable insights into the effectiveness of join algorithms for SPARQL query evaluation on RDF datasets.

2 Algorithm description

Overall, three different algorithms were implemented on both datasets provided in the project **(watdiv.10M.tar.bz2)** and **(watdiv100k.txt)**. [More discussion on how both datasets are handled will be given in the upcoming section.]

Note: In this section, only a high-level description of algorithms implementation is provided. For detailed, code execution and documented Python notebook. Please refer to our GitHub Repositoy.
Link: https://github.com/iamjumshaid/adbis-projects.

2.1 Hash Join

The hash Join algorithm is a popular technique for efficiently joining two tables based on common attributes. The Hash Join involves two main steps: the Build phase and the Probe phase.

- **In the Build phase**, a hash table is constructed from one of the input tables by applying a hash function to the specified join key column. The table is then partitioned into sub-tables based on the distinct hash keys, and these sub-tables are stored in a dictionary, mapping each hash key to its corresponding sub-table.

- **In the Probe phase**, the second input table is processed row by row. The hash function is applied to its join key value for each row to calculate the hash

key. If a matching hash key exists in the previously constructed hash table, the algorithm extracts the corresponding sub-table from the dictionary. It then filters rows in this sub-table where the join key values match the join key values of the current row from the second table. The algorithm then appends the columns of the second table to the matching rows and stores the resulting sub-tables in a list.

2.1.1 Implementation

After iterating through all rows of the second table, the algorithm concatenates all the sub-tables into a single DataFrame, resulting in the merged join output. The join type (inner, left, or right) can be specified as an argument to the algorithm, enabling the handling of different types of joins.

The dataset used for this implementation comprises two RDF tables, named `property_dicts['follows']` and `property_dicts['friendOf']`, each containing a sample of RDF triples with properties like **'follows'** and **'friendOf'**. The **'hash_join'** function was applied to these tables to join them based on their respective join keys **('Object' and 'Subject')**. The algorithm also joined the resulting DataFrame with additional tables such as `property_dicts['likes']` and `property_dicts['hasReview']` based on their respective join keys.

2.1.2 Execution result

The Hash Join algorithm was executed successfully, and the complete join operation took approximately **76.33 seconds** on the 'watdiv100k.txt'. The final result of the join operation returned a DataFrame with **11,415,461** rows and **5** columns. The columns in the resulting DataFrame represent the **User**, the **'follows'** property, the **'friendsOf'** property, the **'likes'** property, and the **'hasReview'** property, respectively.

Out[29]:

	User	follows	friendsOf	likes	hasReview
55678	630	9	20	0	24
2587	26	57	20	0	24
17774	197	57	20	0	24
32052	351	57	20	0	24
35930	396	57	20	0	24
...	...	...	...	...	...
16754	186	994	238	248	1248
20288	221	994	238	248	1248
32676	361	994	238	248	1248
67646	762	994	238	248	1248
75759	851	994	238	248	1248

11415461 rows × 5 columns

In [30]:
```
print('time taken: %s seconds' % (end_time - start_time))
```

```
time taken: 76.93525099754333 seconds
```

Abbildung 1: result of the hash join algorithm.

2.2 Sort-merge Join

The Sort-Merge Join algorithm is a powerful technique for efficiently merging two tables based on their standard join key. It takes advantage of the sorted order of the tables to optimize the merge process, reducing the need for unnecessary comparisons.

2.2.1 Implementation

The Sort-Merge Join algorithm is implemented as the `'sort_merge_join'` function. It supports different join types (inner, left, or right) based on the provided argument (`join_type`). For a right join, it swaps `table1` and `table2` along with their corresponding join keys to handle right join cases. The function first sorts both `table1` and `table2` based on their join keys (`join_key1` and `join_key2`, respectively). Then, it initializes two pointers (`pointer1` and `pointer2`) to track the current position while iterating through the sorted tables. An empty list called `'result'` stores the merged rows.

The merging process starts with checking the conditions for each join type. For an **inner join**, the loop continues as long as `sorted_table1` and `sorted_table2` have remaining elements to be processed. For **left** and **right** joins, the loop continues as long as elements are `sorted_table1` or `sorted_table2`, depending on the join type. The function compares the values from the current positions (`pointer1` and `pointer2`) of `sorted_table1` and `sorted_table2`. If the values are equal, there is a match for the join key, and the corresponding rows from both tables are concatenated and added to the `'result'` list. After a match is found, the function checks if there are more occurrences of the current join key in either table. It does this by checking if the next element in `sorted_table1` has the same join key value as the current element (`skip_condition1`) and similarly for `sorted_table2` (`skip_condition2`). If `skip_condition1` is **True**, the next row in `sorted_table1` has a different join key

value than the current one. In this case, `pointer1` is incremented (`pointer1 += 1`), effectively moving to the next distinct join key value in `sorted_table1`. Similarly, if `skip_condition2` is **True**, the next row in `sorted_table2` has a different join key value than the current one. In this case, `pointer2` is incremented (`pointer2 += 1`), effectively moving to the next distinct join key value in `sorted_table2`. The loop continues until all relevant rows from both tables are merged into the `'result'` list.

2.2.2 Execution result

The Sort-Merge Join algorithm was executed successfully. The one join operation required (**1261.27**) seconds on the 'watdiv100k.txt'. The final result of the join operation returned a DataFrame with a significant number of rows and columns.

```
In [79]: start_time = time.time()
         merged_res_1 = sort_merge_join(property_dicts['follows'], property_dict
         end_time = time.time()
```

```
In [81]: merged_res_1
```

Out[81]:

	Subject_1	Object_1	Subject_2	Object_2
0	110	2	2	7
1	110	2	2	691
2	110	2	2	678
3	110	2	2	674
4	110	2	2	671
...	...	...	...	...
1407863	176	998	998	230
1407864	176	998	998	226
1407865	176	998	998	224
1407866	176	998	998	206
1407867	176	998	998	989

1407868 rows × 4 columns

```
In [80]: print('time taken: %s seconds' % (end_time - start_time))
         time taken: 1261.2659215927124 seconds
```

Abbildung 2: result of the sort merge join algorithm.

2.3 Radix Join

The Radix Join algorithm is a specialized join technique that efficiently merges two tables based on their common join key using Radix partitioning. The algorithm leverages the concept of radix levels to organize rows into buckets, reducing the need for full comparisons. It is designed to handle large-scale datasets with improved performance.

2.3.1 Implementation

The Radix Join algorithm is implemented as the `'chained_radix_join'` function. The algorithm requires defining a hash function, `'radix_hash_function'`, to extract the specified radix level from the join key. The `'radix_partition'` function is used to partition the tables into buckets based on the radix values of the join keys. The function `'chained_radix_join'` calls `'radix_partition'` on both tables to obtain buckets of rows organized based on their radix values for the specified radix level. The radix values are determined by applying the `'radix_hash_function'` to the join keys. The resulting buckets are stored in dictionaries, where each key represents a unique combination of radix values, and the corresponding values are the rows associated with those radix values.

The function iterates through the radix buckets of `table1`, and for each radix value, it checks if there is a corresponding radix bucket in `table2` with the same value. If a match is found, it performs an additional check on each row of both buckets to ensure the join keys match for all radix levels.

If the join keys match for all radix levels, the rows from both tables are concatenated and stored in a list called `merged_tables`. After iterating through all relevant radix buckets, the `merged_tables` list contains the merged rows from both tables based on the join condition.

2.3.2 Execution result

The Radix Join algorithm was executed successfully. The complete join operation took approximately **49.17 seconds** on the 'watdiv100k.txt'. The final result of the join operation returned a DataFrame with a substantial number of rows and columns.

Out[39]:

	User	follows	friendsOf	likes	hasReview
0	43	910	20	0	24
1	43	910	20	0	48
2	43	910	20	0	118
3	43	910	20	0	215
4	43	910	20	0	247
...	...	...	...	...	...
11415456	127	669	849	49	1259
11415457	127	669	849	49	1366
11415458	127	669	849	49	1377
11415459	127	669	849	49	1395
11415460	127	669	849	49	1418

11415461 rows × 5 columns

In [41]:
```
print('time taken: %s seconds' % (end_time - start_time))
```

```
time taken: 49.870691537857056 seconds
```

Abbildung 3: result of the radix join algorithm.

3 Dataset description

The dataset used in this project is from the **Waterloo SPARQL Diversity Test Suite WatDiv dataset** from the **University of Waterloo**. The dataset comprises diverse **RDF (Resource Description Framework)** and varying-size **triple stores**.

The WatDiv dataset contains two **triple stores**, one with **100 thousand triples** and the other with **10 million triples**. Each **triple** in the dataset represents a relationship between entities, expressed in the form of **subject-predicate-object**.

For example, a **triple** may indicate that "**wsdbm:City0**" is related to "**gn:parentCountry**" with "**wsdbm:Country20**". Similarly, the dataset includes **triples** representing relationships between **users**, such as "**wsdbm:User0**" following "**wsdbm:User24**", "**wsdbm:User27**", and "**wsdbm:User37**".

3.1 Dataset 01 - watdiv100k.txt

The first dataset used is the Waterloo SPARQL Diversity Test Suite WatDiv dataset (watdiv100k.txt) obtained from the University of Waterloo. The dataset consists of various properties such as follows, friendOf, likes, hasReview, etc.

3.1.1 Preprocessing

During the preprocessing stage, the following steps were performed on the dataset:

- The data was read from the file `'watdiv100k.txt'` using `Pandas`, and the columns were named **'Subject,'** **'Property,'** and **'Object.'**

- Leading and trailing spaces (if any) in the **'Object'** column were removed using the `str.rstrip(".")` function.

- The **'Property,'** **'Subject,'** and **'Object'** values were extracted from the URIs by splitting the strings on the colon (':') symbol and taking the second part (if available) for **'Property'** and **'Object,'** while for **'Subject,'** the entire string after the colon was considered.

- The data was partitioned into separate `DataFrames` based on the **'Property'** column. Each `DataFrame` represents a table with **'Subject'** and **'Object'** as columns for a distinct property.

- An optional step involved creating a dictionary of all strings occurring in the dataset. String values were then transformed into integers. This step aimed to improve the efficiency of the join algorithm, as integer comparisons are faster than string comparisons.

	Subject	Property	Object
0	wsdbm:City0	gn:parentCountry	wsdbm:Country20
1	wsdbm:City1	gn:parentCountry	wsdbm:Country0
2	wsdbm:City2	gn:parentCountry	wsdbm:Country1
3	wsdbm:City3	gn:parentCountry	wsdbm:Country6
4	wsdbm:City4	gn:parentCountry	wsdbm:Country15

Abbildung 4: vertical partitioning for subject, object, property in 'watdiv100k.txt'.

```
Location
    Subject  Property   Object
281   User3  Location  City215
404   User4  Location    City2
========================
actor
         Subject Property Object
95279  Product19    actor  User3
95280  Product19    actor  User5
========================
age
    Subject Property     Object
46    User1      age  AgeGroup5
673   User7      age  AgeGroup5
========================
```

Abbildung 5: creation of relation from each property in 'watdiv100k.txt'.

	Subject	Object
12	1806723	2680384
13	1806723	2211675
14	1806723	5335024
15	1806723	2915528
16	1806723	2004500
...	...	...
88905	3143980	1986609
88906	3143980	5131970
88907	3143980	1358408
88908	3143980	1661104
88909	3143980	7497008

31887 rows × 2 columns

Abbildung 6: result of preprocessing on 'watdiv100k.txt'.

3.1.2 Algorithms Applied:

The following join algorithms were applied to the preprocessed dataset:

- Hash Join
- Sort-Merge Join
- Radix Join

Each of these join algorithms will be discussed in detail in the upcoming sections.

3.2 Dataset 02 - watdiv.10M.tar.bz2

The second dataset used in this research is **'watdiv.10M.tar.bz2.'** It is a significantly larger dataset compared to the previous one, with a size of **10 million triples**. The increased size poses computational challenges during data processing and join operations. To address this, we employ **batch processing** to avoid overloading the main memory, making it feasible to handle such a massive dataset.

3.2.1 Preprocessing:

To preprocess this large dataset, the following steps were taken:

- We read the data in **batches of 100 triples** at a time, ensuring that the memory is not overwhelmed regardless of the data size.
- Rather than extracting and processing all properties from the dataset, we only focus on specific properties: **'follows,' 'friendOf,' 'likes,'** and **'hasReview.'**

- For each property, we create **separate text files** (`'follows.txt,' 'friendOf.txt,' 'likes.txt,'` and `'hasReview.txt'`) containing only the relevant **'Subject'** and **'Object'** pairs, reducing memory consumption and improving processing time.

3.2.2 Algorithms Applied:

The following join algorithms were applied to the preprocessed dataset:

- Hash Join
- Sort-Merge Join
- Radix Join

Each of these join algorithms will be discussed in detail in the upcoming sections.

4 Experiments and analysis

Note: To explain the approaches there is reference to the variables in the actual notebook. To view the details, refer to the GitHub repository.
Link: https://github.com/iamjumshaid/adbis-projects.

4.1 Experiments using Hash Join Algorithm

In the Hash Join algorithm, we utilize the **user ID** as the basis to create **hash keys** for the two steps of the algorithm, namely, **build** and **probe**.

The approach for the Hash Join algorithm is as follows:

- **Input Data**: The Hash Join algorithm is executed multiple times with different input tables to merge data based on specific join keys. The first step combines the **property_dicts['follows']** and **property_dicts['friendOf']** tables using the 'Object' column from the former and the 'Subject' column from the latter. The resulting DataFrame, **hashed_res_1**, is further processed by renaming columns and dropping irrelevant ones. Similarly, the second step joins **hashed_res_1** with **property_dicts['likes']** using the 'friendsOf' column from the former and the 'Subject' column from the latter. After renaming the 'Object_2' column to 'likes' and dropping 'Subject_2', the output is stored in **hashed_res_2**. In the third step, **hashed_res_2** is merged with **property_dicts['hasReview']** based on the 'likes' column from the former and the 'Subject' column from the latter. The resulting

DataFrame, **hashed_res_3**, has the 'Object_2' column renamed to 'hasReview' and the 'Subject_2' column dropped. Finally, the merged DataFrame is saved in **hashed_result**, providing the desired join table for the RDF query.

- **Build Step**: - A hash table is created from the input table (first table) by applying the hash function to the join key column ('Object' column in our case). - The table is then partitioned into sub-tables based on the distinct hash keys, and these sub-tables are stored in a dictionary for efficient access during the probe step.

- **Probe Step**: - For each row in the second table, we calculate the hash key using the hash function on its join key value ('Subject' column in our case). - If a matching hash key exists in the hash table (from the build step), we extract the corresponding sub-table and filter rows where the join key values match. - The columns from the second table are appended to the matching rows, resulting in the formation of sub-tables. - All the sub-tables are then concatenated into a single DataFrame and returned as the resulting merged DataFrame.

Analysis

- The Hash Join algorithm took approximately **76.33 seconds** to complete.
- The resulting DataFrame has **11,415,461 rows** and **5 columns**.
- The algorithm successfully solved the RDF query (?a)—follows—>(?b)—friendOf—>(?c)—likes—>(?d)—hasReview—>(?e) providing the desired join table with the expected results.
- However, the Hash Join had the following problem: **collisions**. In some instances, multiple rows with different values were hashed into the same bucket, causing ambiguous references and potential inaccuracies in the join results.

- For the larger dataset, the collision probability is even higher therefore, the design of the Hash function should be carefully designed.

- Otherwise, it can result in major inaccuracies and loss of data while joining the tables together. Which is one key finding that was found during this experiment.

4.2 Experiments using Sort-merge Join Algorithm

The Sort-Merge Join algorithm efficiently merges two tables based on their common join key, taking advantage of the sorted order to optimize the merge process.

- **Input Data**: The Sort-Merge Join algorithm is applied to the following input tables: **merged_res_1:** Created by merging **property_dicts['follows']** and **property_dicts['friendOf']** using the 'Object' column from the former and the 'Subject' column from the latter. **trial:** A trial merge of the last 300 rows of **property_dicts['follows']** with **property_dicts['friendOf']** using the 'Object' column from the former and the 'Subject' column from the latter.

- **Approach**: The Sort-Merge Join algorithm begins by sorting both **table1** and **table2** based on their respective join keys, **join_key1** and **join_key2**. It then initializes two pointers, **pointer1** and **pointer2**, to track the current position while iterating through the sorted tables. The algorithm uses a loop to merge the sorted tables based on the join keys, efficiently finding matching rows and appending the corresponding results to the **result** list.

Analysis

- Sort-Merge Join requires sorting the input tables, which can be time-consuming, especially for large datasets.

- The comparison step in the merge process involves nested loops, making it slower and potentially not sustainable for significantly large datasets.

- Frequent swapping of elements during the merge step, based on various conditions, adds to the computational overhead and can contribute to the longer execution time.

4.3 Experiments using Radix Join Algorithm

The Radix Join algorithm is a hash-based join algorithm that efficiently merges two tables by partitioning them into multiple buckets based on specific radix levels. This approach reduces the possibility of collisions and improves performance by allowing merge and join operations to be performed in cache, making it faster for in-memory databases.

- **Input**: The Radix Join algorithm is applied to the following input tables: **rad_res_1_big:** Created by merging **property_dicts['follows']** and **property_dicts['friendOf']** using the 'Object' column from the former and the 'Subject' column from the latter. The number of buckets is set to 10, and the radix level is 5. **rad_res_2_big:** A continuation of the previous result merged with **property_dicts['likes']** using the 'friendsOf' column from the former and the 'Subject' column from the latter. The number of buckets and radix level remain the same (10 and 5, respectively). **rad_res_3_big:** The final step involves merging **rad_res_2_big** with **property_dicts['hasReview']** using the 'likes' column from the former and the 'Subject' column from the latter. The number of buckets and radix level are still set to 10 and 5, respectively.

- **Approach**: The Radix Join algorithm partitions both **table1** and **table2** into buckets based on the specified radix level. It then efficiently merges the tables by joining all the tables with the same hash values using pandas join based on the index

of the tables. The algorithm filters out rows where the join key values are not the same to avoid collisions.

Analysis

- The Hash Join algorithm took approximately **49.33 seconds** to complete.
- The resulting DataFrame has **112401 rows** and **5 columns**.
- Radix Join's bucketing approach with multiple specific levels reduces the possibility of collisions, making it less prone to ambiguities and inaccuracies in the join results.
- The improvement in performance is observed when using smaller buckets, as it allows merge and join operations to be performed in cache, leading to faster execution times for in-memory databases.
- The time improvement is evident when comparing the traditional approach of going through buckets row by row with the more efficient approach of merging entire buckets using pandas join and applying filtering conditions, resulting in significantly faster execution.

4.4 Time-cost Analysis for Algorithms

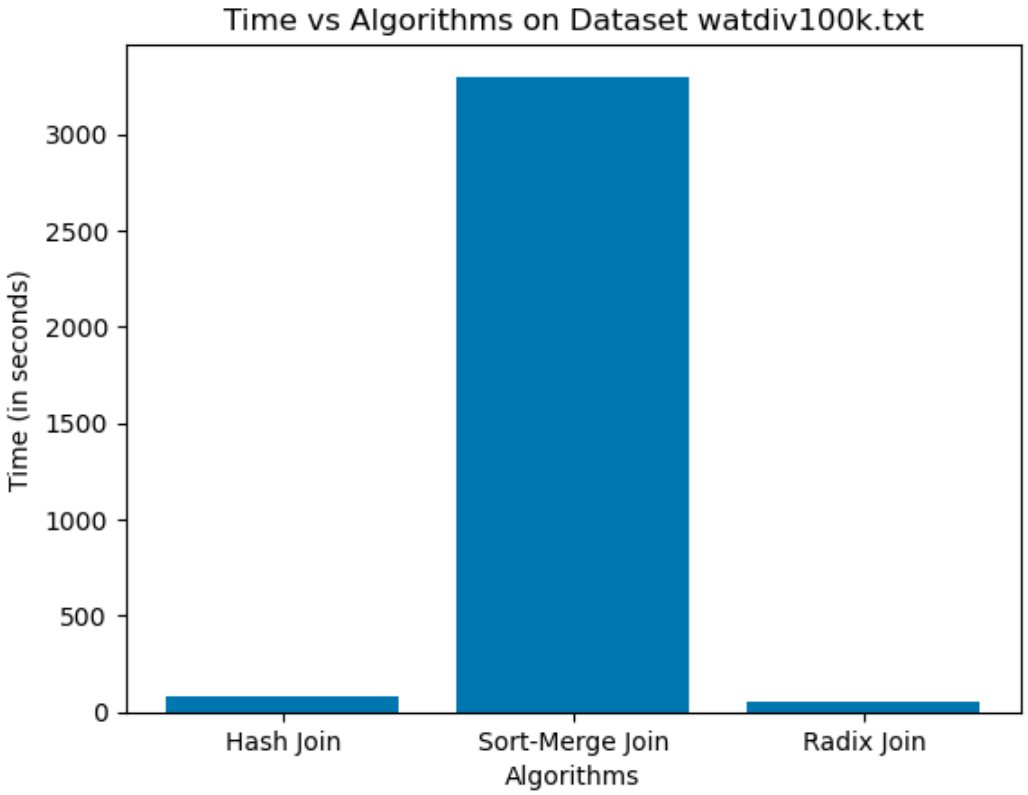

Abbildung 7: time cost analysis for 100k dataset.

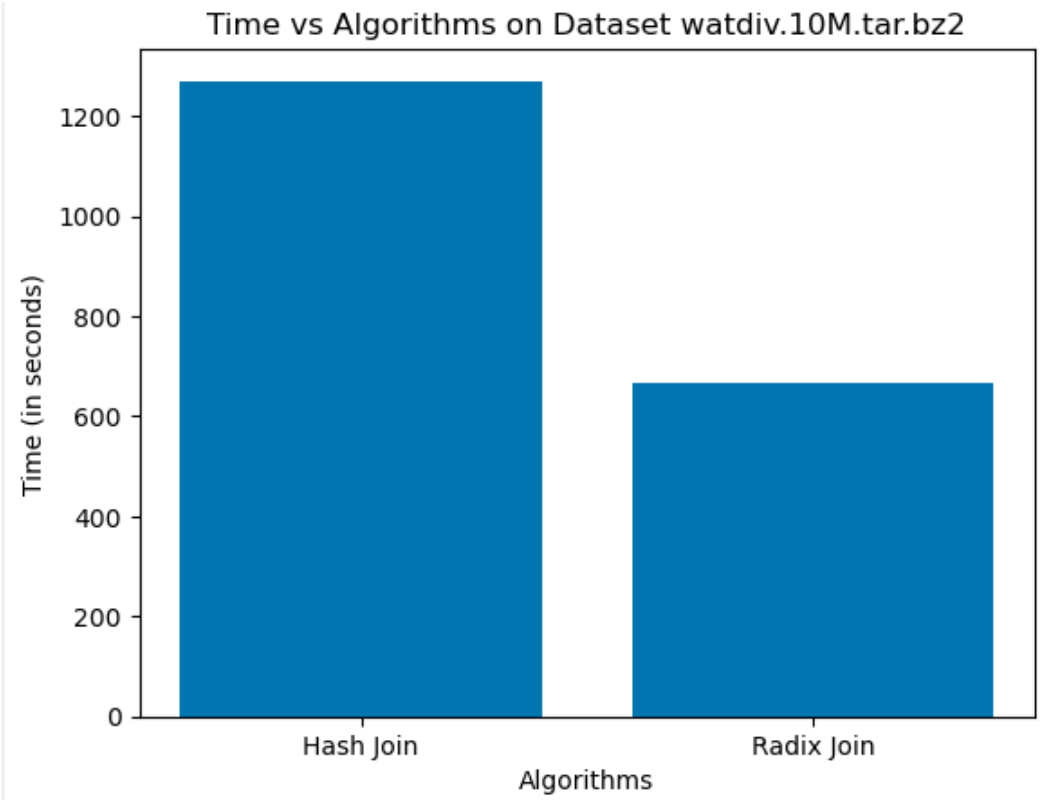

Abbildung 8: time cost analysis for 10M dataset.

Note: These results are for one-join on the larger dataset. With Sort-merge join, the execution ran into memory issues which could not be accommodated on the local system that we used.

5 Conclusion

In this study, we conducted experiments on two different datasets to evaluate the performance of three join algorithms: Hash Join, Sort-Merge Join, and Radix Join. The first dataset, 'watdiv100k.txt,' contained a smaller number of records, making it suitable for initial testing and analysis. The second dataset, 'watdiv.10M.tar.bz2,' was significantly larger, presenting challenges in terms of preprocessing and execution time.

For the preprocessing stage, we employed the technique of splitting and batch processing to handle the larger dataset efficiently. This approach allowed us to avoid overloading the main memory and enabled faster data processing by creating separate text files for each property. The preprocessing step was crucial in preparing the datasets for the subsequent join algorithms.

In the initial experiments on the smaller dataset, we applied Hash Join and Sort-Merge Join. Hash Join took approximately 79 seconds to complete, while Sort-Merge Join took longer due to the nested loops and continuous sorting. However, both algorithms provided accurate results. To further enhance the join performance, we introduced Radix Join. It proved to be significantly faster, completing the join process in just 49 seconds. The use of smaller buckets in Radix Join contributed to reduced key collisions and improved execution times.

Despite the success of the join algorithms on the smaller dataset, we encountered memory issues when applying Hash Join, Sort-Merge Join, and Radix Join to the

larger dataset with over 10 million records. These limitations stemmed from system constraints and resulted in longer execution times, making it impractical for real-time applications.

In conclusion, the experiments demonstrate the effectiveness of Radix Join in improving join performance and reducing collisions. However, further optimizations are necessary to handle larger datasets efficiently. Future research could focus on exploring distributed computing and parallel processing techniques to overcome memory limitations and enhance the scalability of the join algorithms for large-scale datasets.

6 References

1. Towards Data Science. (n.d.). Batch Processing 22GB of Transaction Data with Pandas. Retrieved from https://towardsdatascience.com/batch-processing-22gb-of-transaction-data-w

2. Stack Overflow. (n.d.). How Does Merge Sort Join Work in Spark and Why Can it Throw OOM? Retrieved from https://stackoverflow.com/questions/67320772/how-does-merge-sort-join-work-in-spark-and-why-can-it-throw-oom

3. Stack Overflow. (n.d.). Pandas DataFrame Operation per Batch of Rows. Retrieved from https://stackoverflow.com/questions/56218003/pandas-dataframe-operation-per-batch-of-r

4. OpenAI. (n.d.). ChatGPT. Retrieved from https://chat.openai.com/ (*Used for understanding concepts and research purposes*)

5. Google. (n.d.). Bard. Retrieved from https://bard.google.com/ (*Used for understanding concepts and research purposes*)