Proof-of-Concept for Query-based parallel outer joins under highly skewed datasets.

Written and Implemented by Sami Hussein Hamid Saeed
(ID 31195261)

Introduction

The paper on query-based outer join [1] highlights the significance of efficient join algorithms for improving the performance of data-intensive operations on skewed datasets in a cloud environment. As data in applications expands in scale, cloud environments play a vital role in facilitating the scaling out of large applications through parallelization and increased storage. Additionally, outer joins are the most common type of query operations in databases [2], frequently observed in web domains and major e-commerce websites where customer ids are associated with their corresponding data [3]. However, a typical challenge encountered in data handling is the inherent skewness of the data, following the Zipf distribution, and its resulting consequences in an outer join query [4]. Consequently, the researchers propose an algorithm to optimize the performance of outer joins performed with highly skewed data in a shared-nothing architecture.

Methodology

High level design

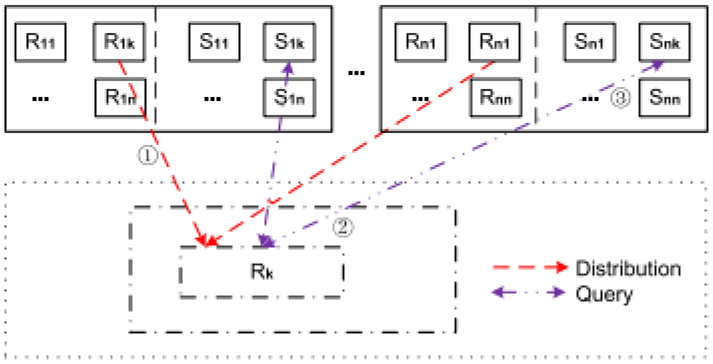


Fig. 1.The query-based approach for outer joins. The dashed square refers to the remote computation nodes and objects [1, Fig.3]

The high level design shown in fig.1 of the proposed query-based outer join algorithm can be summarized into the four following phases:

1) ***R distribution****: Data in Table R is distributed based on the hash value of the joining attribute.*
2) ***Pushing Query keys****: Unique keys of S are extracted. Unique keys are then grouped based on the hash value of the unique key. Grouped keys are distributed across each node.*
3) ***Local Join****: Perform a local left outer join with the partition of table R from step 1 and the distributed keys from step 2. Matched tuples are returned to the nodes from which the keys originate.*
4) ***Inner Join****: After pushing back the tuples, an inner join with Si and the matched tuples is performed per node, giving us the matched part of the solution. The final outer join result is the result of combining the non-matched & matched tuples.*

According to the researchers, this methodology asserts that by pushing the unique keys in step 2, the algorithm effectively eliminates the impact of skew in the outer join. This is achieved by condensing or representing all duplicate values as a single unique value, resulting in a reduction in the number of operations and runtime. Additionally, they argue that the algorithm's distribution of unique keys leads to a decrease in network load within the shared-nothing architecture. They contend that the network cost of communicating keys is significantly lower compared to transmitting entire tuples.

Proof-of-Concept implementation:

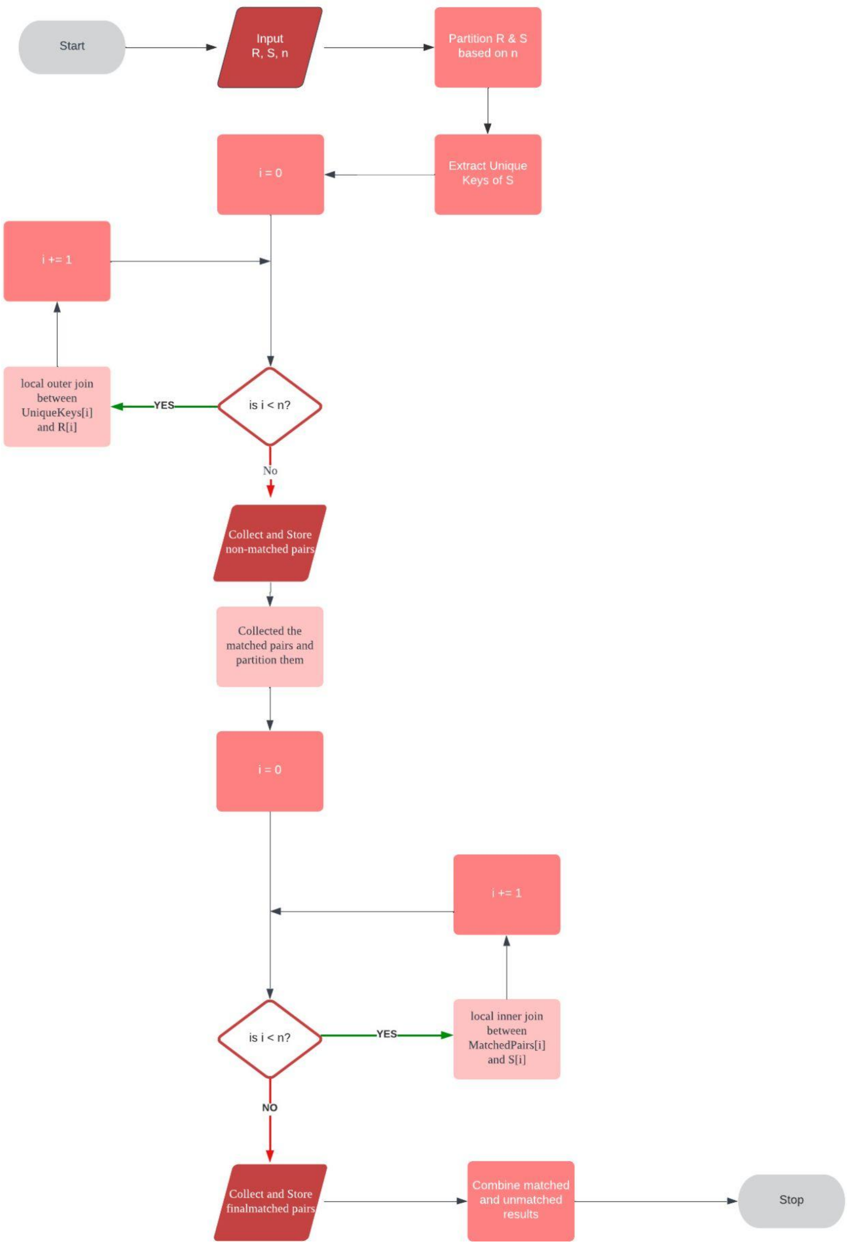
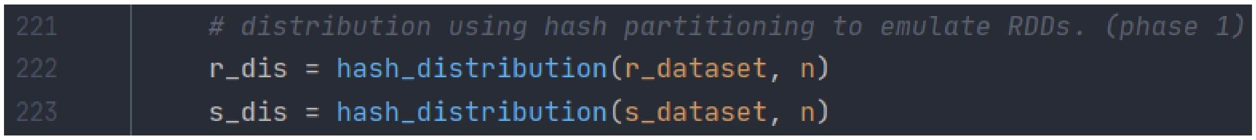

Start
Input
R, S, n
Partition R & S based on n
Extract Unique Keys of S
i = 0
i += 1
local outer join between UniqueKeys[i] and R[i]
YES
is i < n?
No
Collect and Store non-matched pairs
Collected the matched pairs and partition them
i = 0
i += 1
is i < n?
YES
local inner join between MatchedPairs[i] and S[i]
NO
Collect and Store finalmatched pairs
Combine matched and unmatched results
Stop

Fig. 2. Flow chart of the proposed Proof-of-Concept for the query-based outer join.

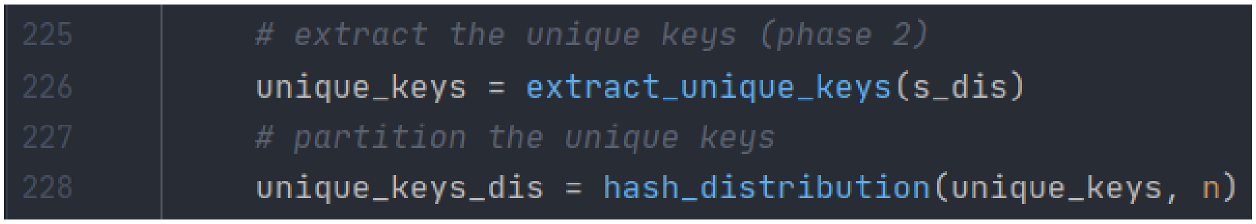
The Proof of concept has been implemented using python, the implementation follows the same concepts used in the parallel outer join lab in week 4 [5]. The overall flow of the algorithm can be summarized in fig.2. However, it is important to note that the inner and outer joins are actually performed in parallel and cannot be simply implemented sequentially.

First we start by partitioning the relations R and S, this acts as a substitute to the RDDS used in the query-based algorithm provided.



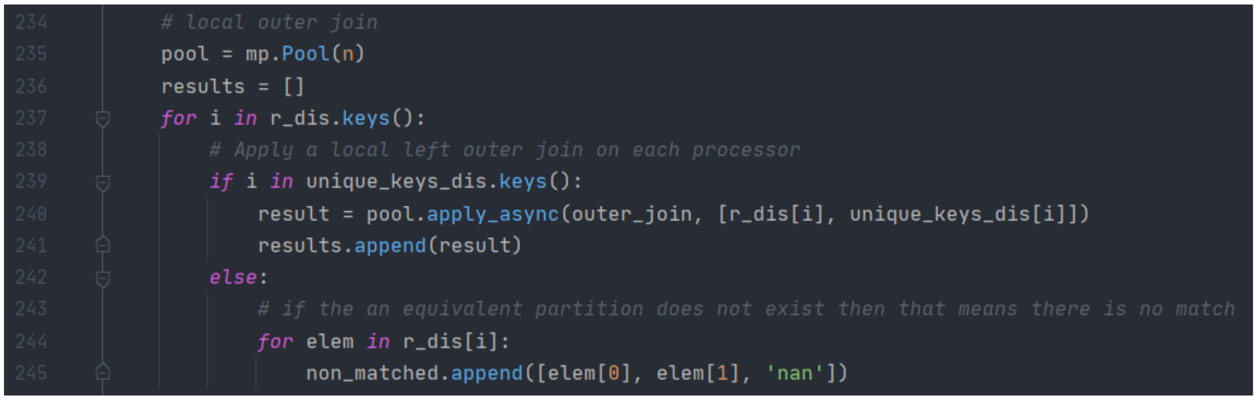
```python
    # distribution using hash partitioning to emulate RDDs. (phase 1)
    r_dis = hash_distribution(r_dataset, n)
    s_dis = hash_distribution(s_dataset, n)
```

After partitioning R and S, we extract the unique keys of the relation S, followed by partitioning the extracted keys in order to outer join the unique distribution with the relation R.



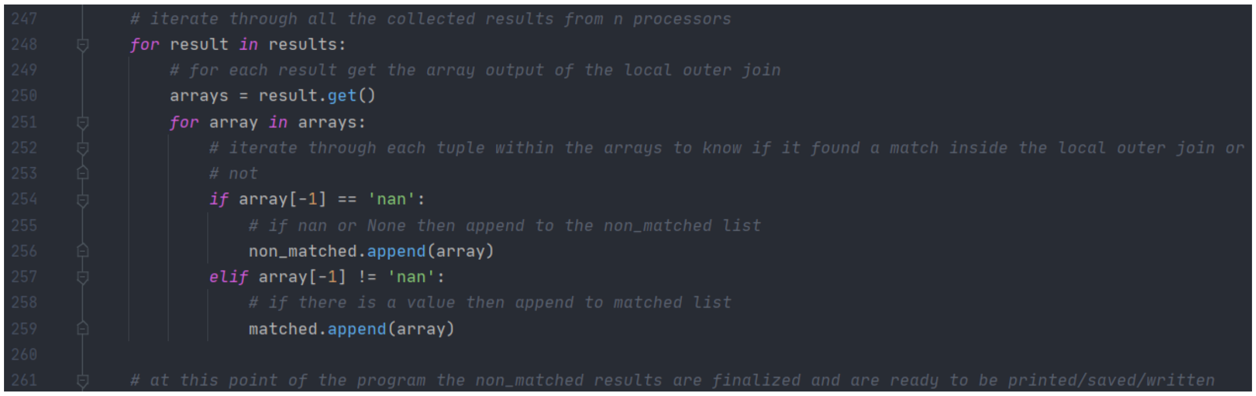
```python
    # extract the unique keys (phase 2)
    unique_keys = extract_unique_keys(s_dis)
    # partition the unique keys
    unique_keys_dis = hash_distribution(unique_keys, n)
```

A left outer join is performed between the relation R and the unique keys, this is the implementation of the local outer join done in phase 3.



```python
    # local outer join
    pool = mp.Pool(n)
    results = []
    for i in r_dis.keys():
        # Apply a local left outer join on each processor
        if i in unique_keys_dis.keys():
            result = pool.apply_async(outer_join, [r_dis[i], unique_keys_dis[i]])
            results.append(result)
        else:
            # if the an equivalent partition does not exist then that means there is no match
            for elem in r_dis[i]:
                non_matched.append([elem[0], elem[1], 'nan'])
```

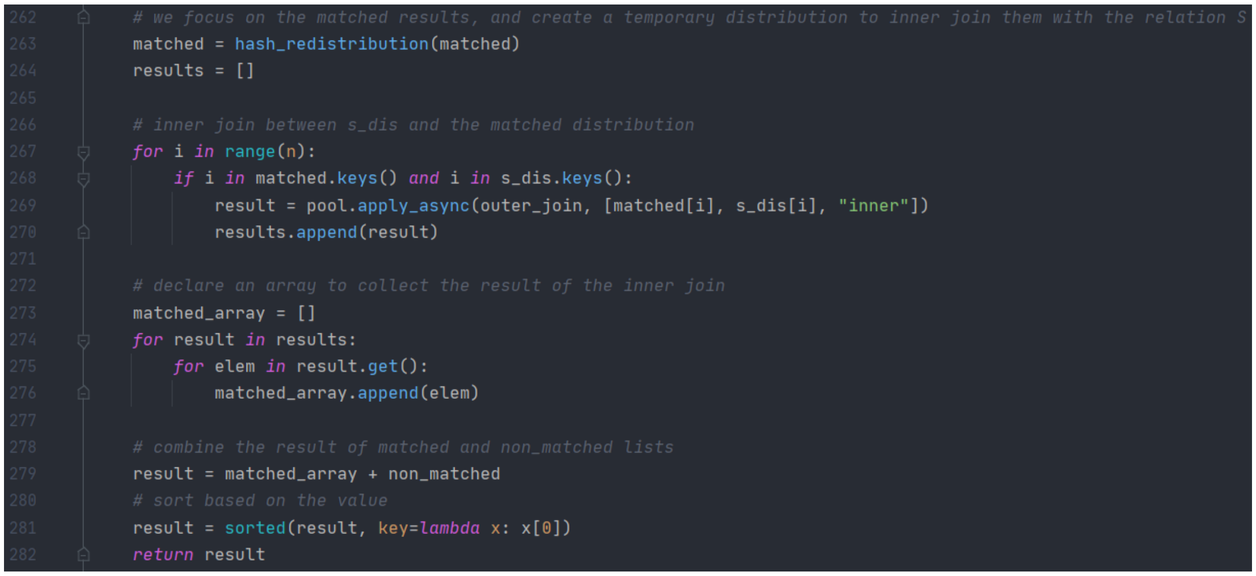
In the following segment of the code we segregate the matched and unmatched results of the outer join. This is done in order to finalize the non matched results for storing and extracting the matched in order to perform the final inner join.



```python
    # iterate through all the collected results from n processors
    for result in results:
        # for each result get the array output of the local outer join
        arrays = result.get()
        for array in arrays:
            # iterate through each tuple within the arrays to know if it found a match inside the local outer join or
            # not
            if array[-1] == 'nan':
                # if nan or None then append to the non_matched list
                non_matched.append(array)
            elif array[-1] != 'nan':
                # if there is a value then append to matched list
                matched.append(array)

    # at this point of the program the non_matched results are finalized and are ready to be printed/saved/written
```

Before the final inner join in phase 4 we must redistribute the matched results based on the index value that we retrieved in the outer join. Finally, a local inner join is performed to finalize the matched results and then we combine both matched and unmatched results and sort them before returning.



```python
    # we focus on the matched results, and create a temporary distribution to inner join them with the relation S
    matched = hash_redistribution(matched)
    results = []

    # inner join between s_dis and the matched distribution
    for i in range(n):
        if i in matched.keys() and i in s_dis.keys():
            result = pool.apply_async(outer_join, [matched[i], s_dis[i], "inner"])
            results.append(result)

    # declare an array to collect the result of the inner join
    matched_array = []
    for result in results:
        for elem in result.get():
            matched_array.append(elem)

    # combine the result of matched and non_matched lists
    result = matched_array + non_matched
    # sort based on the value
    result = sorted(result, key=lambda x: x[0])
    return result
```

Utility Functions

During the implementation of the query-based outer join algorithm several utility functions are used for different purposes such as extracting the unique keys of relation S, distributing and re-distributing data, as well as performing the local outer & inner joins, for these algorithms I have added detailed comments explaining about their purpose and usage within the code.

Analysis of results:

Due to limited resources available on my personal laptop, replicating all of the results in terms of scalability, network load, and load balancing, as the researchers did in the paper, is extremely challenging. Therefore, for this proof of concept, I will be focusing on the most critical factor for evaluating an outer join algorithm: analyzing the query-based algorithm's performance in terms of time. I will conduct this analysis using five different datasets with varying levels of skew, while setting the join selectivity to both 100% and 0%, as described in the research paper [1].

To maintain consistency, I will ensure that both relations in the join operation have the same dataset size. The join operation will be performed with five processors and partitions, and I will test the runtime using a hundred thousand tuples for each relation, aiming to emulate the paper's results as closely as possible.

S Dataset	Size	Unique keys	Skew level
1	100,000	100,000	No Skew
2	100,000	50,000	Low Skew
3	100,000	25,000	Moderate Skew
4	100,000	12,000	High Skew
5	100,000	6,000	

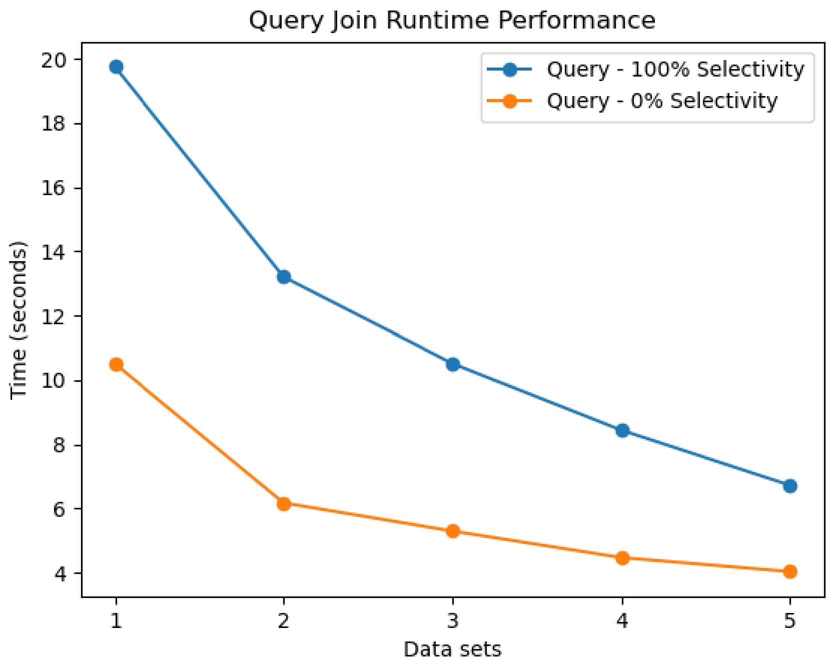


Fig. 3. Query-based join PoC runtime with datasets of varying skew and join selectivity.

The graph depicted in figure 3 illustrates a notable trend: as the skewness of the dataset S increases, the overall runtime experiences a significant decrease. This observation aligns with the claims made by the researchers in their analysis of the algorithm.

The decrease in runtime can be attributed to the diminishing number of unique keys as the skewness of data set S increases. Consequently, during the execution of the local outer join, the algorithm joins progressively smaller datasets. This subsequently leads to a reduction in the size of the data involved in the inner join operation since we only perform an inner join with the matched tuples.

Moreover, in the case of 0% selectivity, the program can terminate the entire join process after performing the local outer join with the unique keys alone, effectively bypassing the unnecessary inner join. This optimization further contributes to a decrease in runtime.

Instructions to run

The program is written in python and can be executed using python. Any IDE that supports python can be used to interpret and run the code. To run the demonstrated code, you can follow these steps:

1. Set up the environment: Make sure you have python installed on your system and configure the IDE to use a python interpreter, if you are using an IDE. Since the code is written in python.

2. Import the necessary libraries: The code utilizes several libraries, including numpy, matplotlib, multiprocessing, time, random, and string. Ensure that these libraries are installed. If any library is missing, you can install it using the pip package manager.

3. Run the driver program **test.py** using the IDE GUI or terminal by entering: **python test.py**.

Conclusion

In conclusion the proof of concept has proven that the query-based algorithm does indeed perform significantly better whilst under high levels of largely skewed datasets. However, due to computational limitations, some results could not be thoroughly tested therefore, further tests using a larger scale implementation can help thoroughly test the algorithm in terms of network load, scalability, and load balancing. Moreover, due to the overall time constraint of the assignment I was unable to implement other algorithms such as PRPD+DER which is based on DER [3] and further improved by the researchers [1] to give us PRPD+Dup [6]. Therefore, in the future it would be of high value to further investigate the performance of the algorithm with respect to the aforementioned algorithms.

References

[1] L. Cheng and S. Kotoulas, "Efficient Skew Handling for Outer Joins in a Cloud Computing Environment," in IEEE Transactions on Cloud Computing, vol. 6, no. 2, pp. 558-571, 1 April-June 2018, doi: 10.1109/TCC.2015.2487965.

[2] M. Atre, “Left bit right: For SPARQL join queries with OPTIONAL patterns (Left-outer-joins),” in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2015, pp. 1793–1808.

[3] Y. Xu and P. Kostamaa, “A new algorithm for small-large table outer joins in parallel DBMS,” in Proc. IEEE 26th Int. Conf. Data Eng., 2010, pp. 1018–1024.

[4] S. Kotoulas, E. Oren, and F. van Harmelen, “Mind the data skew: Distributed inferencing by speeddating in elastic regions,” in Proc. 19th Int. Conf. World Wide Web, 2010, pp. 531–540.

[5] FIT3182 teaching team, "FIT3182-Parallel_outer_join_Solution.ipynb," 2023. Available: https://lms.monash.edu/course/view.php?id=149594&section=8 . Accessed on: 10/5/2023

[6] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, “Handling data skew in parallel joins in Shared-nothing systems,” in Proc. ACM SIG MOD Int. Conf. Manage. Data, 2008, pp. 1043–1052.