Phase #3: Implementing Join and Sort in RDF Database

CSE 510: Database Management Systems Implementation

Arizona State University - Spring 2022

GROUP 10:

Sharanya Banerjee
Aishwarya Baalaji Rao
Manisha Nandkumar Phadke
Lalit Kumar Tiwari
Manoj Mysore Srinath
Vivek Keshava

ABSTRACT: This document serves as the report to the third phase of the project. In this phase, we implement Quadruple join and sort (SMJoin) functionality in our RDF Database. BP_Quadruple_Join() operation implements join on the Quadruples created in phase 2 and generates BasicPatterns which are then sorted using BPSort() operation. We also incorporate a few functions into this architecture, namely: Query and Report to query the database and generate analytical parameters based on the tests performed.

KEYWORDS: Minibase, Quadruples, Confidence, BTree, Nodes, Query, Report, Iterator, Buffer, Page count, Read count, Write count, BasicPattern, Join, Sort.

INTRODUCTION

The third phase of the project is intended to modify the Tuple class, to create a *BasicPattern class*, storing nodeIDs. The *BasicPattern class* has a default parameter *Confidence*. The Iterator class is tailored to form *BPIterator class*, which iterates over Basic Patterns. The *BP_Triple_Join class* is implemented to perform sort-merge join operations on the generated Basic Patterns with RDF Quadruples based on Subject or Object node ID. The class contains functions like *get_next(), close(),* etc. to aid with this functionality. The *BPSort class* helps in sorting the Basic Pattern using the BPIterator class based on input parameters (labels of specified nodes or confidence). We also implement the *Query program* to enable query processing using 3 different approaches (details provided below); and *Report program* to enumerate the page reads and writes. This information is also displayed in a graphical manner for user visualization.

PREVIOUS WORK

We implemented the second phase of the project prior to developing the third phase, by modifying the underlying structure of Minibase which uses Tuples, to a new architecture using Quadruples. Each quadruple is formatted to have 4 fields: Subject, Object, Predicate and Confidence stored in some order selected by the user. The records are also indexed based on the user's choice. The system enables the user to insert data into the database using the BatchInsert command. Post insertion, the user can query the db using the Query command to retrieve records from the database. Depending on a few factors like duplicates in data and type of indexing, the number of Page counts, Read counts and Write counts will vary. We leverage several classes in Minibase like BTree, Disk Manager, Global, Heap and Iterator and have also used our own class DBUtils which contains classes to perform operations into the db. We also use the predefined $makedb class to compile the db prior to running commands. Below are the critical things carried forward from Phase 2 implementation to phase 3 implementation.

The assumptions made for implementing the RDF Database are:

1) If there exist quadruples having the same subject, object and predicate then only the quadruple with the highest confidence is stored.
2) For sorting records, the default order is subject, object, predicate, and confidence.
3) For the query program, "*" indicates a null filter.

In implementing the RDF database, records are stored as quadruples in quadruple heap files while entities and predicates are stored in label heap files. The quadrupleheap package is created by modifying the heap package to account for the extended quadruple construct. The major difference between tuple and quadruple is that a tuple is of variable length, while a quadruple has four fixed fields, namely, Subject (identified by EID), Predicate (identified by PID), Object (identified by EID) and a Value (of type attrReal denoting confidence level).

batchinsert: We implemented the program batchinsert which is invoked by the command:

java dbUtils.BatchInsert DATAFILE INDEXOPTION RDFDBNAME

The program first checks if there is an existing DB and deletes all the records present in the DB in order to insert new records based on the index option given. The new data entered from the DATAFILE is first stored in a heap file (*tempresult*). The sorting occurs based on the index option chosen, specified in the INDEXOPTION variable. Based on this INDEXOPTION, we create a BTree file which will be used for retrieving the records later using indexing.

1. The QID values are inserted into the quadruple heap file.
2. The predicate values (labels) are inserted into the predicate heap file.
3. The label values, subjects and objects, are inserted into the entity heap file.

Index Options: The *tempresult* heap file is used to sort the quadruples according to the INDEXOPTION. The indexing options that we implemented are as follows:

Note: <key, value> ordered pair is used for all the indexing options given below. A quadruple BTree file is created (Quadruple_BTreeIndex) where these ordered pairs are stored for indexing.

1. **BTree index file on confidence**: The key in this case is confidence and the values are the matching QIDs formed as <confidence, QIDs>. Whenever there is a query request, the Quadruple_BTreeIndex file is used to fetch the matching QIDs which are indexed by the confidence key.
2. **BTree index file on subject and confidence**: The key in this case is a combination of subject and confidence, and the values are the matching QIDs formed as <subject:confidence, QIDs>. Whenever there is a query request, the Quadruple_BTreeIndex file is used to fetch the matching QIDs which are indexed by subject and confidence key.
3. **BTree index file on object and confidence**: The key for this particular case is a combination of object and confidence, and the values are the matching QIDs formed as <object:confidence, QIDs>.

Whenever there is a query, the Quadruple_BTreeIndex file is used to fetch the matching QIDs which are indexed by object and confidence key.

4. **BTree index file on predicate and confidence**: The key for this particular case is a combination of predicate and confidence, and the values are the matching QIDs formed as <predicate:confidence, QIDs>. Whenever there is a query based on predicate and confidence, the Quadruple_BTreeIndex file is used to fetch the matching QIDs which are indexed by predicate and confidence.
5. **BTree index file on predicate**: The key in this case is the predicate, and the values are the matching QIDs formed as <predicate, QIDs>. Whenever there is a query, the Quadruple_BTreeIndex file is used to fetch the matching QIDs which are indexed by the predicate key.

The openStream method is invoked by passing a subjectFilter, predicateFilter, objectFilter and confidenceFilter. The method returns a stream of quadruples, whose subject label is identical to subjectFilter, predicate label is identical to predicateFilter, object label is same as objectFilter, and confidence is greater than or equal to the confidenceFilter. The order in which the stream of quadruples is returned depends on the orderType which is passed as an argument to the openStream method.
This is done by forming an unclustered BTree sorted heap file or index file on the labels as per the sorting scheme for the given orderType. This is performed after destroying the existing index by deleting the records having that key and QID.

Table 1. given below, correlates orderType to the sorting scheme used:

orderType specified	**Sorting Scheme Used**
1	subject label, predicate label, object label and then confidence
2	predicate label, subject label, object label and then confidence.
3	subject label followed by confidence
4	predicate label followed by confidence
5	object label followed by confidence
6	confidence

Table 1. Sorting Scheme

DESCRIPTION OF IMPLEMENTATION

Basic Pattern: In our RDF (Resource Description Framework) Database, we provide a way to express facts as an Entity-Relationship model where every [Subject-Predicate-Object] set encodes basic facts about entities and their internal relationships. This set is called a **BasicPattern**, and forms the core of SPARQL. Multiple Basic Patterns that are connected together form a **Basic Graph Pattern** or **Query Graph**.

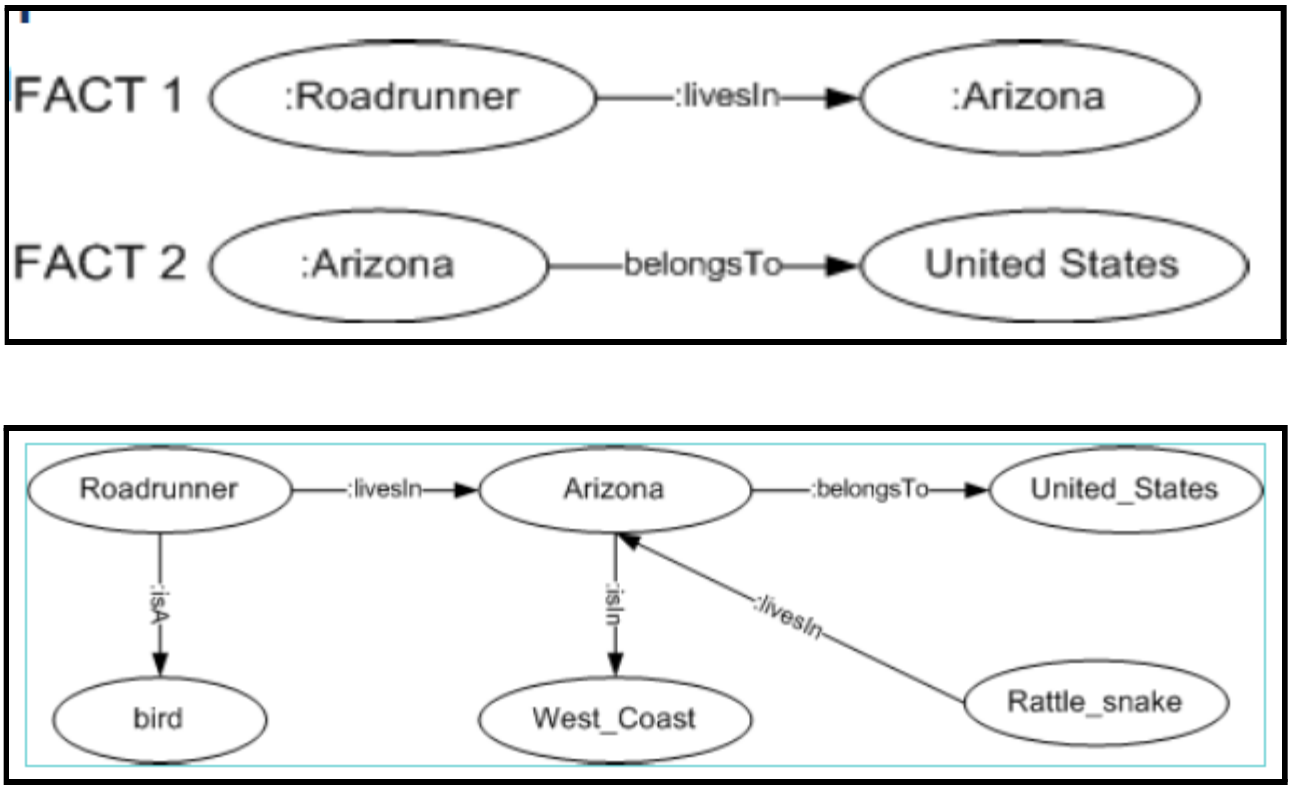

Fig. 1: Graphical depiction of Query Graph comprising of Basic Patterns

From the above query graph, we can get Basic Patterns as: *Roadrunner livesin Arizona, Arizona isin West_coast, Arizona belongsTo United_States, Roadrunner isA bird, etc.*

The most fundamental graph patterns are collections of triple patterns. SPARQL graph pattern matching is described in terms of combining the results of matching fundamental graph patterns. A single basic graph pattern is formed by a succession of triple patterns that are interrupted by a filter.

The assumptions made for implementing the SPARQL join are:

1. The Query is limited to 2 levels of join, adhering to the given format in the requirements.
2. The data file used for testing does not have inconsistencies (spacing is consistent with the data file of phase 2)
3. In our query implementation, '*' depicts all and 'null' depicts none.
4. We have run our tests on 2 datasets: larger dataset of 3148 entries for quantitative analysis and a smaller subset to identify if our join conditions are satisfied properly
5. Each Query is designed to follow the format given below -

Query Format:

The format of a query in this phase of the project is given below.

```
S( J(
        J([SF1,PF1,OF1,CF1],
                JNP,JONO,RSF,RPF,ROF,RCF,LONP,ORS,ORO),
        JNP,JONO,RSF,RPF,ROF,RCF,LONP,ORS,ORO
        )
  SO, SNP, NP
)
```

We can read the above query format like the following example -

```
S( J(
        J([*,lives,*,0.05],
                2,0,*,isin,*,*,1&2,1,1
        ),
        2,0,*,opens,*,*,1&2&3,1,1
        )
2, 1, 1
)
```

The first line signifies the operations to be performed by the query. S denotes Sort, while J denotes join as per the parameters specified in the query. The next line represents the subject, predicate, object and confidence of the basic pattern (records on the left side in Join 1) being joined. In this example, all basic patterns being joined will have predicates as "lives" and confidence as 0.05. The output of the first-level

join (the first join at the bottom of the query tree) forms the basic pattern which is then joined with the filtered quadruples at the next level of join. The confidence of the resulting basic pattern is the minimum of the confidences of the contributing (left) basic pattern and (right) quadruple.

The next line can be described as follows:
2: the second node (object) of the resulting basic pattern of the previous join
0: joining on the right hand side of the join is done on the subject of the filtered quadruples from the database. If it were 1, the join would be done on the object of the filtered quadruples
"isin" is the predicate of all the filtered quadruples present on the right side of the join.
1&2 signifies that the first and second node of the output of the previous join are to be used as the basic pattern on the left side of the join.
1,1 signifies that the subject and object are selected from the filtered quadruples.

The third line represents the third level of join which joins on the predicate "opens" with first, second and third nodes of the output of the second join being selected for the left side of the join. Similar to the previous join, the subject and object are selected from the filtered quadruples.

The last line has 3 parameters, namely, the sort order, sort node position and number of pages used for sorting. The sort order can be 0 for ascending order and 1 for descending order. The sort node position represents the node on which sorting is performed.

Execution Strategy for Join: To perform join in our system, we have implemented the following 3 strategies -

1. ***Tuple Oriented Join (*****derived from *Simple Nested Loop Join)***:
   In this approach, the filtered records from the stream are scanned for each basic pattern. Since this is a brute force approach, the time complexity is polynomial. If m is the number of basic patterns (on the left side of the join) and n is the number of filtered records (on the right side of the join), the join algorithm matches every record on the right side (n) for every basic pattern on the left (m); therefore the number of checks performed is m*n. Thus, the time complexity is O(mn).
   More quantitative analysis is provided in the experimentation section.

2. ***Hash Oriented Join*** **(derived from** ***Block Nested Loop Join*****):**

   This approach involves calling the stream once to retrieve n records, and creating a hashmap for each record. The key of the hash map is the object or subject of the record, represented in String form. The choice of using subject or object as the key depends on which field the join is being performed upon. For example, if the join is being performed on the subject, the hash map stores the subject of the record as the key of that record. The value of every key in the hashmap is an arraylist consisting of the EID of the subject, PID of the predicate, EID of the object and the confidence. There may be multiple EIDs and PIDs corresponding to the same key (subject or object of the record) and the values are stored as an arraylist for each key. If there are M records on the left side of the join and N filtered records on the right side of the join, this algorithm performs the join in constant time O(1). However, if the keys are repeated in the hash map, it leads to hash collision, creating a long tail connected by a linked list. In this situation, the worst case time complexity may be O(N), i.e. linear time. This is not necessarily an undesirable situation, since we want the join output to contain all possible values as per the given conditions. So, this Join works best in situations with high selectivity.

   This is a better approach as compared to the previous strategy, because the worst time complexity of this approach is linear time which is less than the average case time complexity of polynomial time.

   $$O(n) << O(mn)$$

   We got our most optimal solution using Hash oriented join, more quantitative analysis about which is provided in the experimentation section.

3. ***Sort Merge Join:***

   In this join we Sort the Basic Patterns derived from both the first level and second level Join and then Merge them. The expensive operation incase of a Sort Merge join is the sorting phase. We use BPSort to sort on the column that the Basic Pattern will be joined with the filtered results of the right side. We implemented SM join to increase sorting run size in Tuple oriented join, by sorting the DB first before merging. However, for large DB with lerge number of duplicates (low selectivity), SM join takes a considerably long time due to high complexity (Large number of tuples to be compared on either side while sorting). This is eventually addressed by performing Hash oriented join (Option 2). More quantitative analysis is provided in the experimentation section.

Execution Strategy for Sorting:

We are using Heap Sort (External Merge Sort) to sort the results of the basic pattern, which are tuples. In contrast, phase 2 involved sorting quadruples. Heap sort is a sorting algorithm based on comparisons.

Heapsort is similar to selection sort in that it splits the input data into a sorted and an unsorted area and successively decreases the unsorted part by taking the biggest element from it and putting it into the sorted region. Unlike selection sort, heapsort does not spend time scanning the unsorted area in linear time; rather, it stores the unsorted region in a heap data structure in order to identify the biggest element more rapidly in each step.

Heap Sort Algorithm:

The two phases of Heap Sort are:

1) Converting the list of elements into a max heap. In a max heap, the items are stored in an order such that the value in a parent node is greater than the values in its two children nodes.
2) Swapping the first and last elements of the list repeatedly, which reduces the range of values examined in the heap operation by one and filters the new first value into its position in the heap.

The above process is repeated until the range of considered values is reduced to a single value.

The steps followed in the algorithm are as follows:

1. On the input list, call the buildMaxHeap() method. Also known as heapify(), this method constructs a heap from a list in O(n) operations.

```
heapify(array)
        Root = array[0]
        Largest = largest( array[0] , array [2 * 0 + 1]. array[2 * 0 + 2])
        if(Root != Largest)
                Swap(Root, Largest)
```

2. Swap the list's initial and last elements. Reduce the list's considered range by one.
3. To sift the new initial element to its appropriate index in the heap, call the siftDown() function on the given list.
    a. siftDown() - siftDown begins with any node's value. It descends the tree by swapping the value sequentially with the lesser of its two descendants. The action is repeated until the value is smaller than both of its children, or until it reaches a leaf.
    b. Time complexity of this algorithm is O(logN).
4. Keep repeating step 2 until the considered range of the list is reduced to one element.

The buildMaxHeap() action is performed once and takes O(N) time to complete. The siftDown() method takes O(logN) time to execute and is invoked n times. As a result, this method has an overall time

complexity of O(N + N log N) = O(N log N) for best, average and worst cases, and space complexity of O(1).

Report function: We implemented a program which could be run via the command line to display statistics for the RDF database, such as the name of the RDF database, the size of the database, the size of the pages of the database, the number of pages, size of quadruple, total count of entities, subjects, predicates, objects, quadruples, names of the quadruple files, entity files, predicate files and their record count, and the page replacement policy used. The program also returns the number of disk pages read and the number of disk pages written on.

Description of Experiment Sets: We have performed our experiments in accordance with the following plan:

1: Running sample query and recording page reads (on dataset with high selectivity)

2: Running sample query and recording page writes (on dataset with high selectivity)

3: Running sample query and recording elapsed time (on dataset with low selectivity)

4: Changing buffer size <numBuf> and recording page read/writes

5: Toggling join filters and recording page read/writes

Experiment sets:

nested-loop buffer space size- 100					
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**	**sort-exec-time**
1	**15**	**0**	**149**	**25**	**126**
2	**14**	**8**	**87**	**42**	**110**
3	**14**	**8**	**76**	**65**	**103**
4	**15**	**9**	**88**	**46**	**107**
5	**15**	**8**	**73**	**70**	**111**

hash-based-join buffer space size-100					
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**	**sort-exec-time**
1	**15**	**9**	**32**	**30**	**104**
2	**14**	**8**	**73**	**86**	**103**
3	**14**	**8**	**65**	**65**	**75**
4	**15**	**9**	**101**	**53**	**96**
5	**15**	**8**	**36**	**45**	**91**
nested-loop **buffer space size-200**					
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**	**sort-exec-time**
1	**15**	**9**	**104**	**29**	**43**
2	**14**	**8**	**97**	**56**	**95**
hash-based-join **buffer space size-200**					
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**	**sort-exec-time**
1	**15**	**9**	**64**	**58**	**138**
2	**14**	**8**	**41**	**31**	**78**
3148 data					

nested-loop buffer space size-100					
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**	**sort-exec-time**
1	**106543**	**208**	**980**	**878**	**34**
2	**107893**	**210**	**992**	**1122**	**40**
hash-based-join buffer space size-100					
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**	**sort-exec-time**
1	**105309**	**186**	**204**	**143**	**37**
2	**393**	**13**	**357**	**322**	**35**
3	**344**	**6**	**148**	**131**	**11**
4	**274**	**7**	**560**	**795**	**11**
5	**928**	**21**	**40**	**12**	**33**
nested-loop buffer space size-500					
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**	**sort-exec-time**
1	**327**	**171**	**1519**	**1011**	**44**
2	**161**	**8**	**1303**	**2338**	**55**
hash-based-join buffer space size-500					

index-options	page-read	page-write	first-join-exec-time	second-join-exec-time	sort-exec-time
1	0	0	138	124	14
2	0	0	128	117	8
3	0	0	126	127	9
4	0	0	155	147	7
5	0	0	49	19	20
Sort-merge buffer space size-500					
index-options	page-read	page-write	first-join-exec-time	second-join-exec-time	sort-exec-time
1	0	0	1021	1128	2
2	0	0	512	437	3
3	0	0	520	594	2
4	0	0			
5	0	0			
Sort-merge buffer space size-100					
index-options	page-read	page-write	first-join-exec-time	second-join-exec-time	sort-exec-time
1	1847991	2959	1563	1784	2
2	1741	171	499	533	1
3	1779	149	544	594	2
4	1880	69	16	9	3
5	5086	141	20	11	5

all exec-times are in ms				
nested-loop buffer space size- 100				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**15**	**0**	**149**	**25**
2	**14**	**8**	**87**	**42**
hash-based-join buffer space size- 100				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**15**	**9**	**32**	**30**
2	**14**	**8**	**73**	**86**
nested-loop buffer space size- 200				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**15**	**9**	**104**	**29**
2	**14**	**8**	**97**	**56**

all exec-times are in ms				
nested-loop buffer space size- 100				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**15**	**0**	**149**	**25**
2	**14**	**8**	**87**	**42**
hash-based-join buffer space size- 100				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**15**	**9**	**64**	**58**
2	**14**	**8**	**41**	**31**
Dataset Size: 3148				
nested-loop buffer space size- 100				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**106543**	**208**	**980**	**878**
2	**107893**	**210**	**992**	**1122**

all exec-times are in ms				
nested-loop buffer space size- 100				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**15**	**0**	**149**	**25**
2	**14**	**8**	**87**	**42**
hash-based-join buffer space size- 100				
buffer space size: 100				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**105309**	**186**	**204**	**143**
2	**393**	**13**	**357**	**322**
nested-loop buffer space size- 500				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**327**	**171**	**1519**	**1011**
2	**161**	**8**	**1303**	**2338**

all exec-times are in ms				
nested-loop buffer space size- 100				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**15**	**0**	**149**	**25**
2	**14**	**8**	**87**	**42**
hash-based-join buffer space size- 100				
buffer space size: 500				
index-options	**page-read**	**page-write**	**first-join-exec-time**	**second-join-exec-time**
1	**327**	**171**	**214**	**269**
2	**161**	**8**	**302**	**331**

Table 2. Experiment Sets

Explanation of Results:

From the above, we see changes in our observations by tweaking a few parameters. The first two tables depict the system running on a smaller set of data with their corresponding page reads and page writes. When the system is subjected to a larger data set, we see the page reads and writes increase substantially but they level out with increasing numBuf. We also see a linear increase in execution time with increase in size of data. However, the readings with the most clarity are those of hash-based join, where we see that even with an increase in data set size, the performance is equally good or arguably better. This aligns well with our hypothesis that a hash-based join will have a better run-time complexity when compared to tuple-oriented join.

Supporting Graphs:

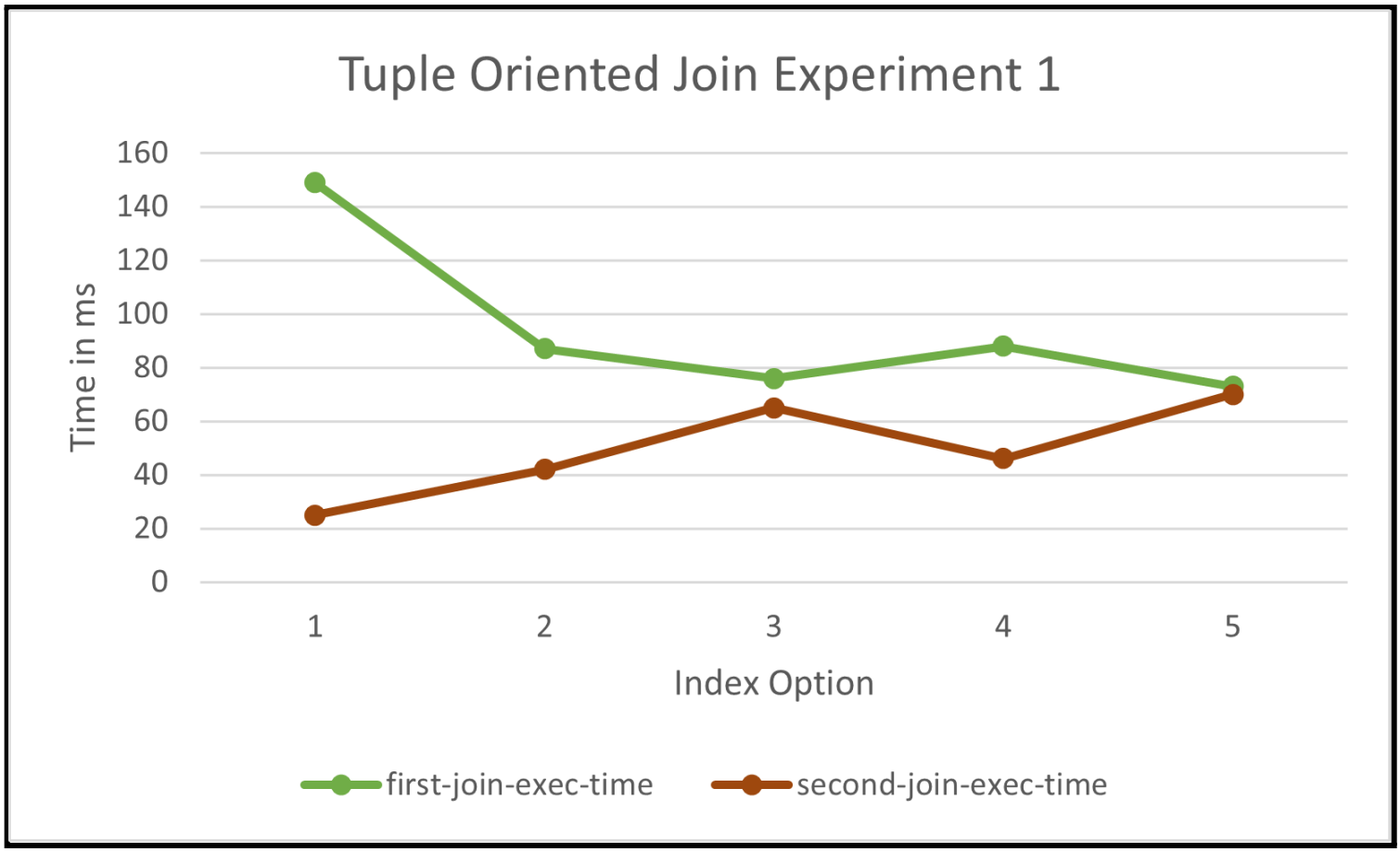

Fig. 2: Tuple Oriented Join Experiment 1

This graph pertains to the first experiment we performed using tuple-oriented nested loop join (buffer size 100). We have plotted the time taken to perform first join and second join against 5 index options. As we can observe, for index option 1, the first join takes considerably more time to perform than the second join, after which the lines taper to demonstrate similar execution times. Even then, it is evident that the execution time of the first join is always greater, howsoever marginally, than that of the second join.

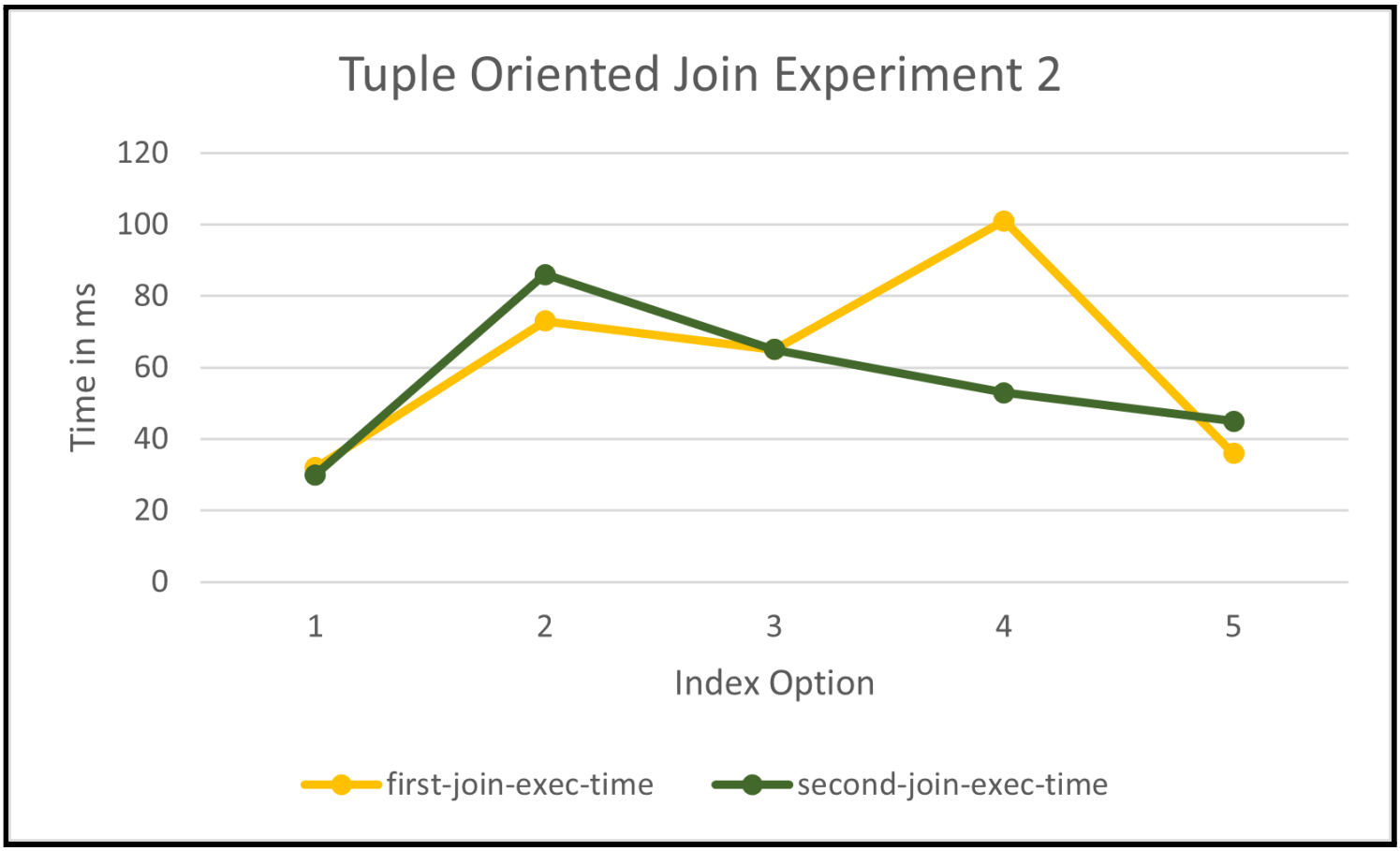

Fig. 3: Tuple Oriented Join Experiment 2

This graph pertains to the second experiment we performed using tuple-oriented nested loop join (buffer size 500). We have plotted the time taken to perform first join and second join against 5 index options. As we can observe, for index option 1, the first join takes almost equal time to perform like the second join. The lines overlap at index 3 but for index 4, the first join takes considerably more time than the second join (possibly due to conflict in data orientation); after which, the times converge again.

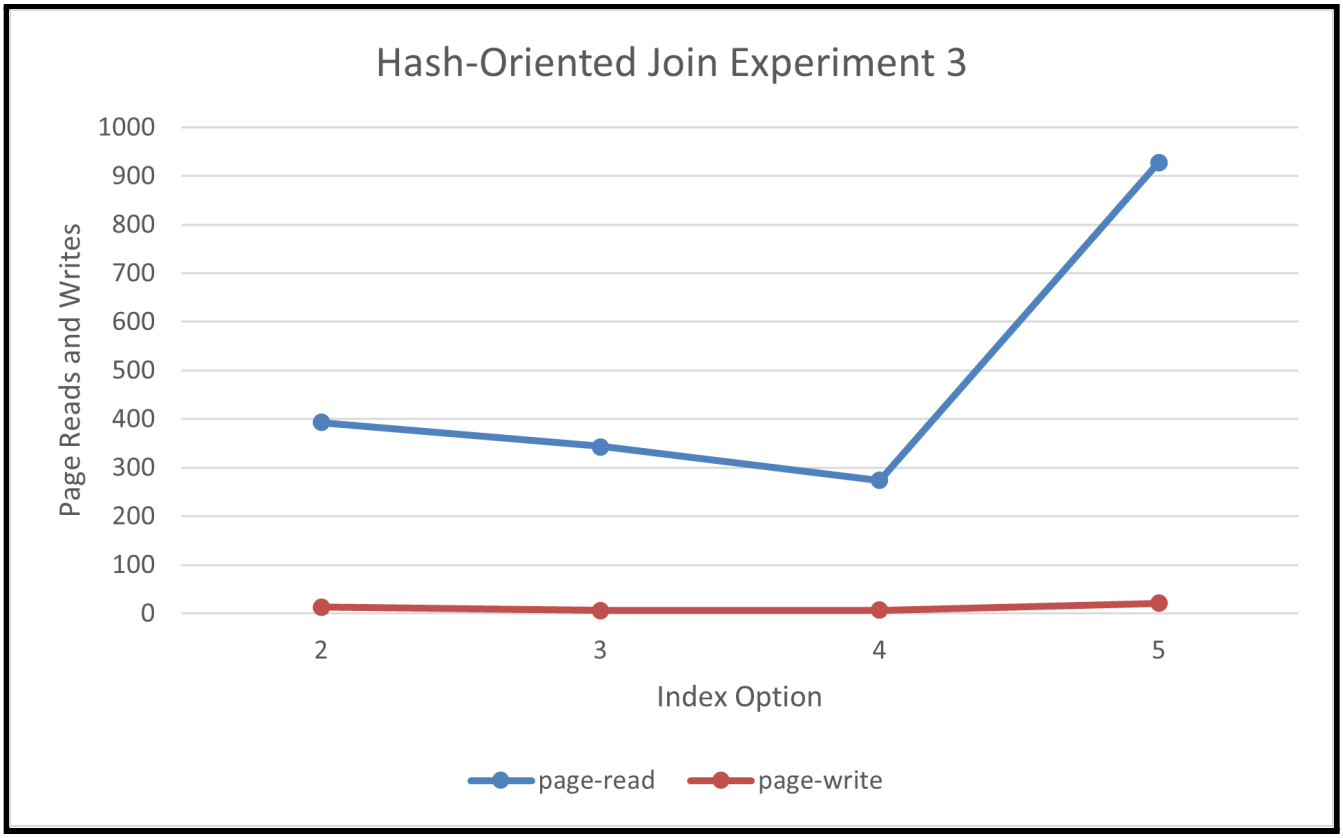

Fig. 4: Hash Oriented Join Experiment 3

This graph pertains to the third experiment we performed using hash-oriented nested loop join (buffer size 100). We have plotted the page reads and writes against 4 index options. As we can observe, the number of writes are negligible because we are performing read operations only without writing anything to disk. For read counter, we see a consistent decrease for options 2 to 4, and for option 5, an abrupt increase which can be justified by duplicate values in the hash table leading to hash collision, creating long tails, hence longer reads.

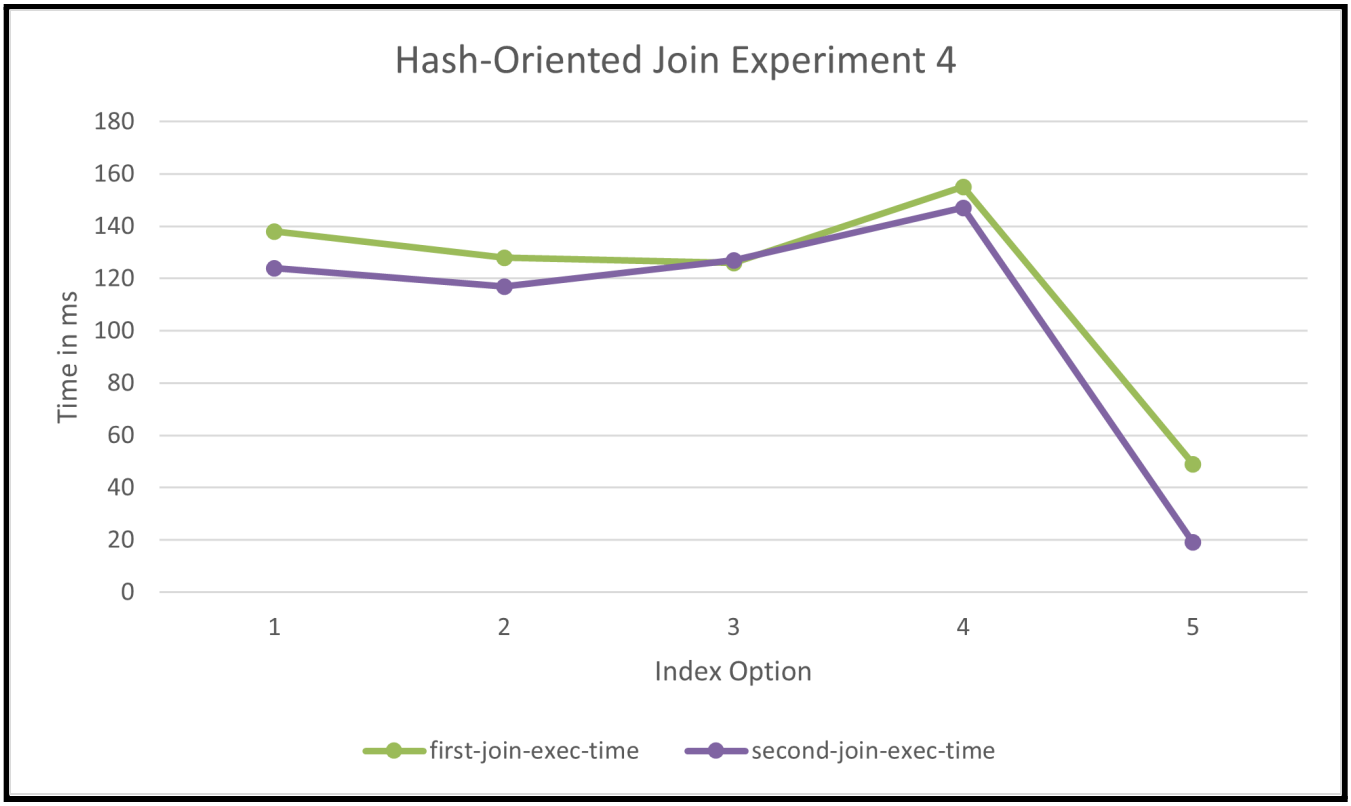

Fig. 5: Hash Oriented Join Experiment 4

This graph pertains to the fourth experiment we performed using hash-oriented nested loop join (buffer size 500). We have plotted the page reads and writes against 5 index options. As we can observe, both joins take almost equivalent time to perform with steady increase and decrease on both. The only time they converge is on index option 3. On index 5, they are lowest, which might indicate that the data is hashed most efficiently with minimum collision.

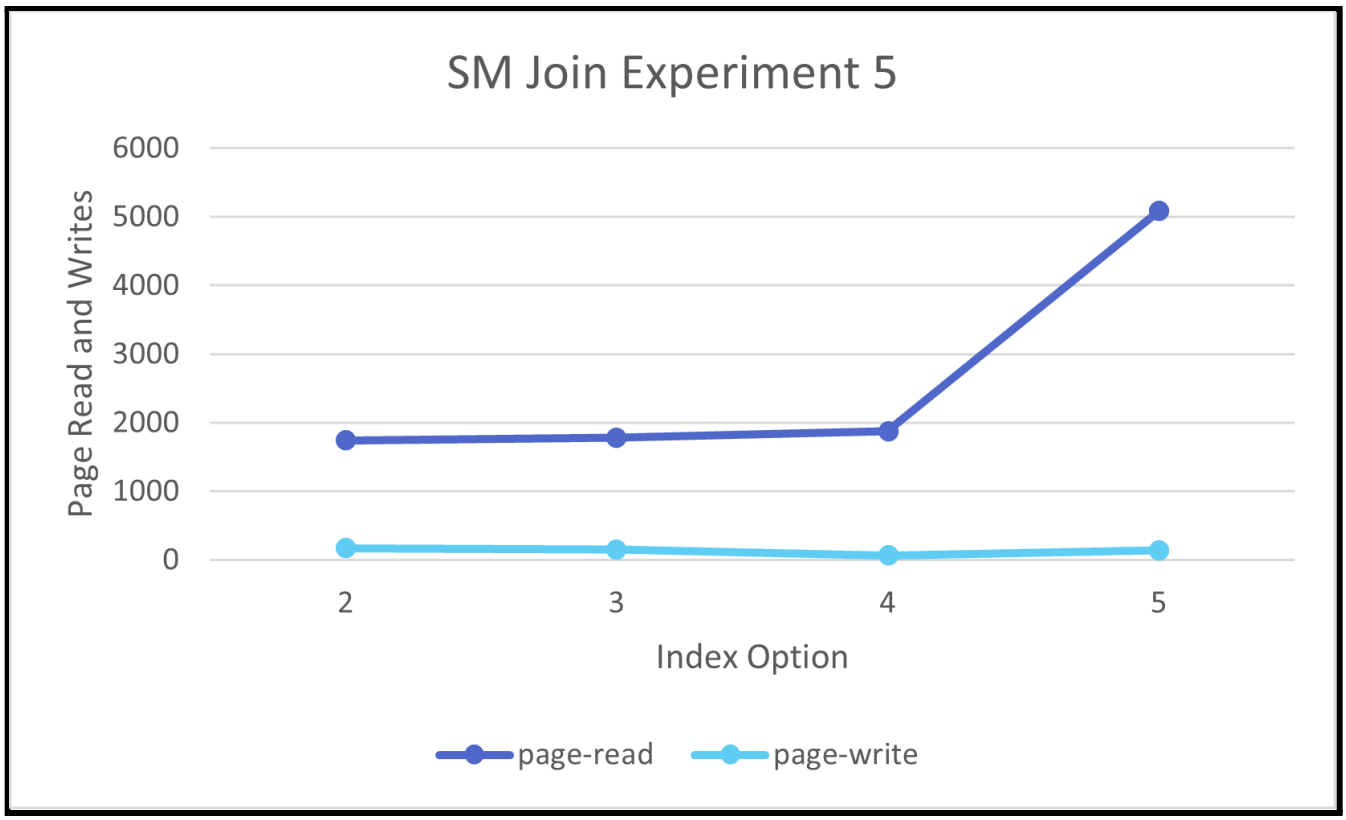

Fig. 6: Sort Merge Join Experiment 5

This graph pertains to the fifth experiment we performed using sort-merge (SM) join (buffer size 100). We have plotted the page reads and writes against 4 index options. From the above graph, we see a linear decline in the number of writes which was negligible. The number of reads maintain a value of roughly 2000 for index options 2, 3 and 4. But for index option 5, there is a steady jump to 5000, which could be explained by the algorithm having to sort a large number of values.

This concludes our experimentation.

INTERFACE SPECIFICATION

The sample query format is given below.

```
S( J(
        J([SF1,PF1,OF1,CF1],
                JNP,JONO,RSF,ROF,RCF,LONP,ORS,ORO
        ),
        JNP,JONO,RSF,ROF,RCF,LONP,ORS,ORO
```

)

SO, NP

)

The acronyms are expanded are follows:

S - select

J - join

SF1 - subject filter

PF1 - predicate filter

OF1 - object filter

CF1 - confidence filter

JNP - join node position

JONO - join on object

RSF - right subject filter

ROF - right object filter

RCF - right confidence filter

LONP - left out node position

ORS - output right subject

ORO - output right object

SO - sort order

NP - number of pages

Our interface is a command line format which is invoked by the following command:

java dbUtils.BatchInsert tests/8-\ phase3_test_data.txt 1 "db14"

java JoinQuery.java "vkdb1_1" dbUtils/queryfile 100

EXAMPLE RESULTS

Opening existing DB

========== BASIC RECORDS THAT MATCH FILTER PATTERN ==========

<< Seattle Free 0.158295679646559 >>

<< Sharanya Seattle 0.228295679646559 >>

<< Paul America 0.338295679646559 >>

<< Ironman Malibu 0.418295679646559 >>

<< Lalit Seattle 0.618295679646559 >>

<< Batman Seattle 0.618295679646559 >>

==========

==========

====

========== BASIC FIRST LEVEL JOIN RESULTS ==========

<< Sharanya Seattle Washington 0.208295679646559 >>

<< Sharanya Seattle ColdPlace 0.228295679646559 >>

<< Ironman Malibu California 0.268295679646559 >>

<< Lalit Seattle Washington 0.208295679646559 >>

<< Lalit Seattle ColdPlace 0.268295679646559 >>

<< Batman Seattle Washington 0.208295679646559 >>

<< Batman Seattle ColdPlace 0.268295679646559 >>

First Level Join Count:7

Time taken: 1713 millis

==========

==========

====

========== BASIC SECOND LEVEL JOIN RESULTS ==========

<< Sharanya Washington Seattle Google 0.208295679646559 >>

<< Sharanya ColdPlace Seattle Google 0.208295679646559 >>

<< Lalit Washington Seattle Google 0.208295679646559 >>

<< Lalit ColdPlace Seattle Google 0.208295679646559 >>

<< Batman Washington Seattle Google 0.208295679646559 >>

<< Batman ColdPlace Seattle Google 0.208295679646559 >>

Second Level Join Count:6

Time taken: 1632 millis

==========

==========

====

========== BASIC SORTED RESULTS ==========

<< Sharanya Washington Seattle Google 0.208295679646559 >>

<< Sharanya ColdPlace Seattle Google 0.208295679646559 >>

<< Lalit Washington Seattle Google 0.208295679646559 >>

<< Lalit ColdPlace Seattle Google 0.208295679646559 >>

<< Batman Washington Seattle Google 0.208295679646559 >>

<< Batman ColdPlace Seattle Google 0.208295679646559 >>

==========

==========

====

Time taken: 5 millis

NET PAGE WRITES: 2769

NET PAGE READS: 3695

========== HASH RECORDS THAT MATCH FILTER PATTERN ==========

<< Seattle Free 0.158295679646559 >>

<< Sharanya Seattle 0.228295679646559 >>

<< Paul America 0.338295679646559 >>

<< Ironman Malibu 0.418295679646559 >>

<< Lalit Seattle 0.618295679646559 >>

<< Batman Seattle 0.618295679646559 >>

==========

==========

==========

==========

========== HASH FIRST LEVEL JOIN RESULTS ==========

<< Sharanya Seattle Washington 0.208295679646559 >>

<< Sharanya Seattle ColdPlace 0.228295679646559 >>

<< Ironman Malibu California 0.268295679646559 >>

<< Lalit Seattle Washington 0.208295679646559 >>

<< Lalit Seattle ColdPlace 0.268295679646559 >>

<< Batman Seattle Washington 0.208295679646559 >>

<< Batman Seattle ColdPlace 0.268295679646559 >>

4

Time taken: 175 millis

==========

==========

==========

==========

========== HASH

SECOND LEVEL JOIN RESULTS

==========

<< Sharanya Washington Seattle Google 0.208295679646559 >>

<< Sharanya ColdPlace Seattle Google 0.208295679646559 >>

<< Lalit Washington Seattle Google 0.208295679646559 >>

<< Lalit ColdPlace Seattle Google 0.208295679646559 >>

<< Batman Washington Seattle Google 0.208295679646559 >>

<< Batman ColdPlace Seattle Google 0.208295679646559 >>

5

Time taken: 151 millis

==========

==========

====

========== HASH

SORTED RESULTS

==========

<< Sharanya Washington Seattle Google 0.208295679646559 >>

<< Sharanya ColdPlace Seattle Google 0.208295679646559 >>

<< Lalit Washington Seattle Google 0.208295679646559 >>

<< Lalit ColdPlace Seattle Google 0.208295679646559 >>

<< Batman Washington Seattle Google 0.208295679646559 >>

<< Batman ColdPlace Seattle Google 0.208295679646559 >>

==========

==========

====

Time taken: 2 millis

Exception occurred while closing stream!quadrupleheap.QFileAlreadyDeletedException: file alread deleted

NET PAGE WRITES: 527

NET PAGE READS: 636

/tmp/vkdb_1 1 null lives null 0.1

Sorting on object

========== SORTED MERGE

RECORDS THAT MATCH FILTER PATTERN

==========

<< Paul America 0.338295679646559 >>

<< Seattle Free 0.158295679646559 >>

<< Ironman Malibu 0.418295679646559 >>

<< Sharanya Seattle 0.228295679646559 >>

<< Lalit Seattle 0.618295679646559 >>

<< Batman Seattle 0.618295679646559 >>

==========

==========

====

========== SortedMerge

FIRST LEVEL JOIN RESULTS

==========

<< Ironman Malibu California 0.268295679646559 >>

<< Sharanya Seattle Washington 0.208295679646559 >>

<< Sharanya Seattle ColdPlace 0.228295679646559 >>

<< Lalit Seattle Washington 0.208295679646559 >>

<< Lalit Seattle ColdPlace 0.268295679646559 >>

<< Batman Seattle Washington 0.208295679646559 >>

<< Batman Seattle ColdPlace 0.268295679646559 >>

First Level Join Count:7

Time taken: 1004 millis

==========

==========

====

7

========== SORT

MERGE SECOND LEVEL JOIN RESULTS

==========

<< Sharanya Washington Seattle Google 0.208295679646559 >>

<< Sharanya ColdPlace Seattle Google 0.208295679646559 >>

<< Lalit Washington Seattle Google 0.208295679646559 >>

<< Lalit ColdPlace Seattle Google 0.208295679646559 >>

<< Batman Washington Seattle Google 0.208295679646559 >>

<< Batman ColdPlace Seattle Google 0.208295679646559 >>

Time taken: 1137 millis

==========

==========

====

========== SORT

MERGE SORTED RESULTS

==========

<< Sharanya Washington Seattle Google 0.208295679646559 >>

<< Sharanya ColdPlace Seattle Google 0.208295679646559 >>

<< Lalit Washington Seattle Google 0.208295679646559 >>

<< Lalit ColdPlace Seattle Google 0.208295679646559 >>

<< Batman Washington Seattle Google 0.208295679646559 >>

<< Batman ColdPlace Seattle Google 0.208295679646559 >>

==========

==========

====

Time taken: 2 millis

NET PAGE WRITES: 2910

NET PAGE READS: 3567

==========

==========

====

INSTALLATION AND RUNTIME

To install and run the system, we extract the zip file. Then, the following commands are run in terminal:

1. ***cd ./src*** - in Terminal
2. ***$make db*** - Compile all the java classes and create .class files for all packages
3. ***java dbUtils.BatchInsert DATAFILEFILE INDEXOPTION RDFDBNAME*** - Execute this command to batch insert all the data from any text file consisting of inputs, followed by the indexing option, followed by the db name. Eg: *java dbUtils.BatchInsert tests/8-\ phase2_test_data.txt 1 "vkrdfdb1_5"*
4. **S(J(**

   J([SF1,PF1,OF1,CF1],

   JNP,JONO,RSF,ROF,RCF,LONP,ORS,ORO

   ),

   JNP,JONO,RSF,ROF,RCF,LONP,ORS,ORO

)

   SO, NP

) - Execute this command to query the database and perform join. The exposition to this query is given above. This query can be run by invoking the following command: *java JoinQuery.java "vkdb1_1" dbUtils/queryfile 100*
5. ***report RDFDBNAME INDEXOPTION*** - Run this command to provide the statistics of the database like Subject count, Object count, Number of pages, etc.

RELATED WORK

To understand the Minibase architecture, we spent some time going through the Minibase documentation from their homepage, in addition to our work in Phase 1 and Phase 2. We also spent some time understanding what indexing using B and B+ trees is and how we can leverage different indexing schemes in our system. This also entailed a comprehensive analysis of the differences between BTrees and B+Trees and how they are populated in a Bottom-up architecture. The links to these materials are listed in the bibliography section. In addition to this, we have made minor changes to our phase 2 RDF database to fit our phase 3 implementation.

CONCLUSION

From our above system description, we can get a fair idea about the systems architecture. As we can see from the directives, we are successfully able to insert records into the db as well as query information from the db based on user requirements. We can also retrieve information per run like page, read and write counts and number of subjects, objects, predicates, etc. and perform high-level analysis on them to identify root causes for the varying count values. Built on top of our phase 2 implementation, we have implemented 3 strategies to perform join in our RDF database. For every join strategy, we have built a testbench of varying data size and buffer size and recorded the observations. We have then plotted the observations and drawn interesting observations from the graphs. This concludes the third phase of the project.

BIBLIOGRAPHY

1. Source file: ../../users/sharanyabanerjee/Desktop/Assignments/CSE510 DBMSI/minjava/javaminibase/src/javadoc/overview-frame.html (Hosted on local system).
2. https://research.cs.wisc.edu/coral/minibase/minibase.html
3. https://research.cs.wisc.edu/coral/minibase/project.html
4. https://stackoverflow.com/questions/1108/how-does-database-indexing-work
5. https://dzone.com/articles/database-btree-indexing-in-sqlite
6. https://www.tutorialspoint.com/unix_commands/make.html

APPENDIX

Individual Contribution:

1. **Sharanya Banerjee** - Tuple oriented Join, Hash oriented join
2. **Aishwarya Baalaji Rao** - Tuple Oriented Join, Sort
3. **Manisha Nandkumar Phadke** - Tuple oriented Join, Sort
4. **Lalit Kumar Tiwari** - Hash oriented join, SM Join
5. **Manoj Mysore Srinath** - SM Join, Sort
6. **Vivek Keshava** - Hash oriented join, SM Join
7. **Experimentation -** All team members
8. **Report and Presentation** - All team members