REPORT ON :-

CREATING DERBY AND MONGODB DATABASES AND INDEXES , RUNNING RELEVANT QUERIES ON THEM AND IMPLEMENTING A B+ TREE INDEX IN JAVA

COSC2406

DATABASE SYSTEMS

KAUSHAL GAWRI

s3777121

RMIT UNIVERSITY

TABLE OF CONTENTS

ABSTRACT

The report accommodate the experiments conducted to load data to multiple databases based on *Derby* and *MongoDb* followed by creation of secondary index on certain fields with appropriate explanations. Apart from that, four queries, each for Derby and *MongoDb* databases were created, two of them using the secondary index for certain scanning operations whereas the rest two are the queries that do not use the secondary index. The major focus of the experiment is to compute the time taken for different scan operation by such queries and provided relevant analysis. The report also consists of the experiments related to generating a B+ tree index structure for the heap file generated in assignment 1, by querying over the B+ tree along with the heap file and comparing and analysing the results based on the time taken by each search operation on the heap file and B+ tree. The report also explains the approaches taken to generate the B+ tree using a relevant index field. The report also includes the design and methodology followed along with relevant conclusions derived.

TASK 1 : DERBY

In this task, it is required to load the open source data provided to a derby database that follows normalization along with the creation of relevant indices based on different tables of our database. As Derby uses a Relational Database Management System, the structure used for assignment 1 is used to create indices such that normalization is maintained and thus complexity and data redundancy is reduced while efficiency is optimized.

Following the assignment 1 submission, the structure of the database provided consisting of multiple tables is provided below :

PedestrianTime(ID, Date_Time, Year, Month, Mdate, Day, Time)

PedestrianSensor(ID, Sensor_ID, Sensor_Name)

PedestrianCount(ID, Hourly_Count)

Here, for the *PedestrianTime* table, the secondary index is based on "Year" field, for the *PedestrianSensor* table, the secondary index is based on "*Sensor_ID*" field and for the *PedestrianCount* table, the secondary index is based on "*Hourly_Count*" field whereas "ID" is the primary key chosen for each table.

The "CREATE INDEX" query was used to create a non-clustered index , taking into consideration the real life application of this database, it is recommended to not overload our database with large number of indexes as with a high amount of inserting, updating or deleting any row on the given table of our database will also require to provide the same changes to all the indexes in that table thus reducing the efficiency. So, it was decided to choose a single field for index creation on each of the table as stated above. Also, as ID is the primary key and mostly expected to be used for JOIN conditions, the indices were chosen based on the fields that will be used most of time, such as the columns used in WHERE, LIKE or ORDER BY clauses. As it makes more sense to search a record based on the year for the *PedestrianTime*, given broader results and thus better data can be collected considering a real life example, year was chosen as the secondary index. Apart from that, searching for a sensor makes more sense if its ID is used, so its is assumed that *Sensor_ID* will be a good choice

for secondary index on *PedestrianSensor* and as the whole database is about the hourly count of the light sensors, searching for a certain range of counts or a particular count as well is more expected which helped in choosing *Hourly_Count* as the secondary index for *PedestrianCount* table. The relevant observations recorded for each query is provided in the observation section.

OBSERVATIONS

TIME TAKEN BY EACH OF THE QUERY WHEN DATA IS JUST LOADED ON DERBY BASED ON INDEX :

The following queries were used, two of them using the index for search operation whereas two of them do not use the index, and the time taken by each query was taken into account. These queries can also be found in the code of task1 in class DerbyQuery.java

Queries using secondary index :

Query 1 :

SELECT PEDESTRIANCOUNT.ID, PEDESTRIANSENSOR.SENSOR_ID, PEDESTRIANSENSOR.SENSOR_NAME, PEDESTRIANCOUNT.HOURLY_COUNT FROM PEDESTRIANCOUNT INNER JOIN PEDESTRIANSENSOR ON PEDESTRIANCOUNT.ID = PEDESTRIANSENSOR.ID WHERE PEDESTRIANCOUNT.HOURLY_COUNT LIKE '100' FETCH FIRST 30 ROWS ONLY;

Query 2 :

SELECT PEDESTRIANTIME.M_DATE, PEDESTRIANTIME.MONTH, PEDESTRIANTIME.DATE_TIME, PEDESTRIANSENSOR.SENSOR_NAME , PEDESTRIANSENSOR.SENSOR_ID FROM PEDESTRIANSENSOR INNER JOIN PEDESTRIANTIME ON PEDESTRIANSENSOR.ID = PEDESTRIANTIME.ID WHERE PEDESTRIANTIME.YE_AR like '2010' AND PEDESTRIANSENSOR.SENSOR_ID like '10';

Queries not using secondary index

Query 3:

SELECT PEDESTRIANTIME.ID, PEDESTRIANTIME.YE_AR, PEDESTRIANSENSOR.SENSOR_ID, PEDESTRIANSENSOR.SENSOR_NAME, PEDESTRIANTIME.TIME, PEDESTRIANCOUNT.HOURLY_COUNT FROM PEDESTRIANTIME INNER JOIN PEDESTRIANSENSOR ON PEDESTRIANTIME.ID = PEDESTRIANSENSOR.ID INNER JOIN PEDESTRIANCOUNT ON PEDESTRIANCOUNT.ID = PEDESTRIANSENSOR.ID WHERE PEDESTRIANTIME.DAY LIKE 'TUESDAY' FETCH FIRST 30 ROWS ONLY;

Query 4:

SELECT PEDESTRIANTIME.ID, PEDESTRIANTIME.DATE_TIME, PEDESTRIANCOUNT.HOURLY_COUNT FROM PEDESTRIANTIME INNER JOIN PEDESTRIANCOUNT ON PEDESTRIANTIME.ID = PEDESTRIANCOUNT.ID INNER JOIN PEDESTRIANSENSOR ON PEDESTRIANSENSOR.ID = PEDESTRIANCOUNT.ID WHERE PEDESTRIANSENSOR.SENSOR_NAME LIKE 'Alfred Place';

Observations for time taken just when data is loaded :

Queries using index :

	Test 1 (ms)	Test 2(ms)	Test 3(ms)
Query 1	241	243	250
Query 2	65	46	43

Queries not using index :

	Test 1 (ms)	Test 2 (ms)	Test 3 (ms)
Query 3	2864	2692	2621
Query 4	1840	1689	1768

Observations for time taken after rebooting :

Queries using index :

	Test 1 (ms)	Test 2(ms)	Test 3(ms)
Query 1	244	246	238
Query 2	43	40	52

Queries not using index :

	Test 1 (ms)	Test 2 (ms)	Test 3 (ms)
Query 3	2046	1740	1948
Query 4	1396	1457	1524

Observations for time taken after 1 hour :

	Test 1 (ms)	Test 2 (ms)
Query 1	247	254
Query 2	41	43
Query 3	1480	1324
Query 4	687	742

Figure 1 : Time taken by queries on derby after 1 hour

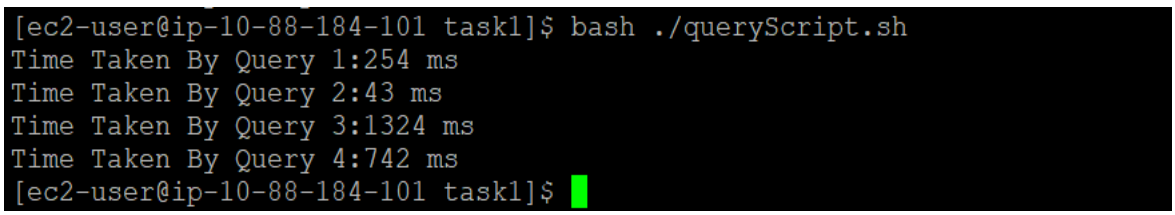

Figure 2 : Time taken by queries on derby after 1 hour

Queries on Derby Database when index was not created :

	Test 1 (ms)	Test 2(ms)	Test 3(ms)
Query 1	100	132	124
Query 2	287	368	312
Query 3	4659	4208	4288
Query 4	5511	5829	5604

```
Total Time For Loading Data: 4528874ms
Time Taken By Query 1:100 ms
Time Taken By Query 2:287 ms
Time Taken By Query 3:4659 ms
Time Taken By Query 4:5511 ms
```

Figure 3 : Time taken by derby to run queries for non-indexed database at creation time

CONCLUSION

The database followed similar structure to that of assignment 1 submission such that normalization is sustained. By following rigorous experiments both on the database with index and without index, with readings recorded at three different interval, once when the data was recorded, other when the system is rebooted and the last one after running it for one hour, certain observations were made. It is observed from the experiments that the two queries that used secondary index in one of the conditional clause, recorded the least time overall :

- *Query 1 taking 244 ms on average when database just created, 242 ms on average on rebooting and 250 ms after 1 hour.*
- *Query 2 taking 51 ms on average when database just create, 45 ms on average on rebooting and 42 ms after 1 hour.*

The query time was much faster on the indexed database when the index were used in the conditional queries on all of the three different time frame. Apart from that, the queries took less time on the indexed database as compared to database on which no index was created. Also, the time taken by other queries that do not use index for conditions on multiple time frames :

- *Query 3 taking 2727 ms, 1911 ms and 1402 ms for each of the three time frames.*
- *Query 4 taking 1765 ms, 1459 ms and 714 ms for each of the three time frames.*

The query time was a lot more as compared to the queries using index for all the three time frames for each of the two queries. Also, the query took less time on the indexed database as compared to database on which no index was created even though the queries didn't have any role of indices in general.

Another observation made was between the difference between the query time for queries not using an index when database was just created and when the observations were noted after an hour. Typically, the difference between the time taken by both type of queries wasn't as crucial as the time increased, that is the query time for non-indexed based queries reduced after the 1 hour mark as compared to initial database creation state. This further helps us in concluding that with and without the secondary index for query, the biggest difference was when derby just started running whereas after an hour mark the difference became less significant which might states that the secondary index is pre-eminently made at the beginning of derby for the maximum output based on performance and efficiency.

It can also be concluded that indexing has utmost important when querying or finding a record as query with no index lets database scan the whole table whereas using indexing lets database use index records which are a record point to a bucket that contains pointers to all the records having their specific search-key value. Thus, the total number of I/O operations are reduced and the data retrieval is much faster. Though, as every coin has two sides, the major drawbacks of using an indexed database could be the time taken to load the data as they take up more space and if the data is modified on regular intervals using operations like INSERT, DELETE and UPDATE, it will require updating all the indices which will further result in slowing down the performance.

TASK 2 : MongoDB

For this task, the json format provided as part of assignment solution was used to load the data in *MongoDb*. The time taken to load data in *MongoDb* is much faster than *Derby* as there is no overhead of shredding of data into structure storage units, as it will be written as a single piece, an observation which was also observed in the first assignment.

The structure used for the mongoDB database is similar to (provided as Assignment1 solution) :

```
{
"_id" : ObjectId("606c9320453926be50f54e07"),
"Sensor_ID":34,
"Sensor_Name": "Flinders St-Spark La",
"Values": {
      "Date_Time": "12/15/2014 04:00:00 PM",
      "Year": 2014,
      "Month": "December",
      "Mdate": 15,
      "Day": "Monday",
      "Time": 16,
      "Hourly_Counts": 219
}
}
```

The secondary index for the *MongoDb* database was created using *"db.COLLECTION_NAME.createIndex()"* on the "*Hourly_Counts*" field as the whole database is about the hourly count of the light sensors, searching for a certain range of counts or a particular count as well is more expected.

OBSERVATIONS

TIME TAKEN BY EACH OF THE QUERY WHEN DATA IS JUST LOADED ON DERBY BASED ON INDEX :

The following queries were used, two of them using the index for search operation whereas two of them do not use the index, and the time taken by each query was taken into account.

Queries using Secondary Index Field :

Query 1 :

```
db.Pedestrian.explain("executionStats").aggregate([
 {
  $project: {
   Sensor_name: "$Sensor_name",
   data: {
    $filter: {
     input: "$data",
     as: "data",
     cond: {
      $gte: [
       "$$data.Hourly_Counts",
       100
      ]
     }
    }
   }
  }
 }
])
```

Query 2 :

```
db.Pedestrian.aggregate([
 {
  $match: {
   Sensor_ID: "62"
  }
 },
 {
  $project: {
   data: {
    $filter: {
     input: "$data",
     as: "data",
     cond: {
      $eq: [
       "$$data.Hourly_Counts",
       2691
      ]
     }
    }
   }
  }
 }
])
```

Queries not using secondary index field :

Query 3: Return details like day and year from collection Pedestrian with sensor id 10

db.Pedestrian.find({"Sensor_ID": 10}, "Details":{$elemMatch: {"Day": "Friday", "Year": "2008"}} , {ID:1, Date_Time:1, Sensor_Name: 1}).pretty().explain("executionStats")

Query 4: Return results from collection Pedestrian where sensor id of record is 30 and date is greater than 10 with month equal to "November".

```
db.Pedestrian.aggregate([
 {
  $match: {
   "Sensor_ID": "30"
  }
 },
 {
  $project: {
   "data": {
    $filter: {
     input: "$data",
     as: "data",
     cond: {
      $gte: [
       "$$data.Mdate",
       10
      ]
     },

    }
   }
  }
 },
 {
  $project: {
   "data": {
    $filter: {
     input: "$data",
     as: "data",
     cond: {
      $eq: [
       "$$data.Month",
       "November"
      ]
     },

    }
   }
  }
 }
])
```

Observations for time taken just when data is loaded :

Queries using index :

	Test 1 (ms)	**Test 2(ms)**	**Test 3(ms)**
Query 1	463	389	406
Query 2	1242	1375	1024

Queries not using index :

	Test 1 (ms)	**Test 2 (ms)**	**Test 3 (ms)**
Query 3	2740	2890	3610
Query 4	3849	4281	4026

Observations for time taken after 1 hour :

Queries using index :

	Test 1 (ms)	**Test 2(ms)**	**Test 3(ms)**
Query 1	362	269	398
Query 2	1034	1108	1179

Queries not using index :

	Test 1 (ms)	**Test 2 (ms)**	**Test 3 (ms)**
Query 3	2453	2512	2945
Query 4	3607	3821	3872

It is observed that the queries using an index and others without using an index at the time of data creation show vast differences in the time taken for execution which is the similar case as of Derby. Though, as compared to another time frame, when the observations were recorded after an hour, the time difference between the queries using and index and others without using an index was almost similar to the time recorded earlier which is one of the major difference observed as compared to derby, as the time difference reduced significantly for derby whereas there was not much difference recorded for the *MongoDb* database.

CONCLUSION

Even though the concept of indexing is similar for both *Derby* (SQL) and *MongoDb*(NoSQL) some minor differences are still observed, be it the difference in the data retrieval time after certain hours of running the application or the time taken to retrieve data just when the index is created. This could be explained by different structure of the two databases even though both the system uses B-Tree for their indexes. In Relational databases like Derby, indexes can be used for faster data retrieval operations where a missing index can lead to a full scan of table to fulfil a query. Whereas *MongoDb* uses pointers to the mapped files as far as indexes are concerned. This memory mapping assigns a file

to a block of virtual memory and this relation between the file and memory lets MongoDB to interact with the data for faster retrievals. Another major difference is the maximum number of indexes that could be created, where SQL databases like Derby can support up to 1000 indexes per tables whereas NoSQL database like *MongoDb* can support only 64 per collection. Even though both the database uses similar supports, the major difference lies between the type of indexes and limitations in sizes. Though, as queries are much easier in *MongoDb* and doesn't require us to know the full structure of database including merging (like joins in Derby) and even indexing, and with not much difference observed in the time apart from stability that Derby observes after some time of running, the efficiency of a system is still increased using MongoDB for a real world application that require frequent changes in data, as indexed database in Derby will drastically increase the data retrieval time, it can be concluded that generally when larger databases are considered, MongoDB is a better choice as compared to Derby Database.

TASK 3 : B+ Tree

DISCUSSION

The basic idea of a B+ tree is to utilize it for dynamic indexing on multiple levels. Apart from that, B+ Tree is based on the idea that the data can only be stored at the leaf nodes of the tree, which in my implementation would be known as "*ChildNode*" . This data storing at the leaf node level thus improves the search process by making it much faster and accurate. Apart from that, the internal nodes(known as "*NeighbourNode*" for my implementation) of the B+ Tree are used to store the leaf nodes. The basic idea is to insert in each leaf node (which is the "*ChildNode*" in this case) in case there is no overflow encountered and otherwise if there is an overflow in the leaf node, split the leaf node into two or if there is an overflow in the inner node (*NeighbourNode*), split the inner node into two. The main reason for using a B+ Tree is that numerous keys or indexes can be easily added on the page of memory as they do not have any data associated with the inner nodes providing quick access to data on the leaf node. Also, as all the leaf nodes are linked together, only one linear pass would be required for a full scan of elements. The implementation would be explained in the next section.

IMPLEMENTATION

To implement the B+ Tree, an abstract class known as *TreeNode* is created to provide code reusability, as it consists of basic methods involved in the process of creating a heap file and running certain queries on it. Now to insert a leaf node, which in our case is the *ChildNode*, the first step involves traversing to the required leaf node and using binary search if the key is present already. So, it is required to go to the appropriate leaf node and *Collections.binarySearch()* is used. Now, the key is inserted into the leaf node (*ChildNode*) only when there is no overflow. If an overflow is noted, in case of an overflow of leaf node (*ChidlNode*), the leaf node is split from half into two nodes. In the first node, place half of the values in a new placement in the tree node, such that the new *ChildNode* contains (m/2)-1 elements, (here m is the size of List key, given by *totalNodes* in my implementation). Now, the second node created after split will contain the remaining values. The next step is to copy the smallest key value from the second node consisting of other half of elements to the parent node such that a right hierarchy of tree is created and the minimum key node is associated with the top

level node. Though, if an overflow in the inner node, which is the *NeighbourNode* in our case is observed, this node is again split into two different nodes. Now, the first node of the split will contain 50% of the values as above and the smallest among the remaining ones are moved to parent node such that the second node of the split contains the rest of the values. The above steps are continued such that a top level node that doesn't need any splits is found and the top level node is updated . This is how insertion is completed. The next part is the search operation on B+ Tree. The search operation is fairly simple and uses a recursive approach on the inner nodes (in our case *NeighbourNode*). Each inner node is marked visited and the leaf nodes associated with the given inner node are explored. Then a simple iteration to all the leaf nodes is used, similar to a linear scan to search for the given key in the tree. If the search parameters match the key, as *String.contains()* is used, even if the search key is a substring of the key value in the B+ Tree, the results are outputted on the console. A more efficient way could be to use binary search on the records in the B+ Tree instead of a recursive approach due to heap memory constraints. As the leaf nodes are connected to each other, (in this case part of the same list) searches are comparatively faster. As far as range searches are concerned, a similar approach to search is used such that a recursive call is made on all the inner nodes and the leaf nodes for each inner node are explored. The key or index of leaf node is then matched with key1 and key2 provided in the query. *String.compareTo()* is used, such that if a value greater than or equal to 0 is returned for key1, it means that the value the leaf node has is greater than the key1 and thus should lie in the range. Also, if a value less than or equal to 0 is returned for key2, it means that the value of the leaf node is less than key2 and thus should also lie in the range. Another condition is checked if key2 is the substring of the value for given leaf node as this requirement is not met by using *String.compareTo()*. The final results are outputted on the console for the user to get an insight on the data the B+ Tree has and the time for both search and range search are recorded for further analysis.

OBSERVATIONS

TIME TAKEN TO LOAD THE DATA IN B+ TREE :-

	Test 1 (ms)	**Test 2 (ms)**	**Test 3 (ms)**
2048 page size	46324	42945	43102
4096 page size	18294	22658	21762
8192 page size	16234	15932	15428

TIME TAKEN TO QUERY B+ TREE :-

Query 1 : *"java dbquery 2018 4096 -b"*

Query 2 : *"java dbquery 3701/02/2021 4096 -b"*

Query 3 : *"java dbquery 2008-2018 4096 -b"*

Query 4 : *"java dbquery 1007/27/2021----1010/27/2021 4096 -b"*

	Test 1 (ms)	Test 2 (ms)	Test 3 (ms)
Query 1	962	819	770
Query 2	77	84	65
Query 3	1213	970	1158
Query 4	155	130	115

CONCLUSION

As compared to the heapfile, B+ Tree takes more time to load data for smaller sizes like 2048. This is because of the presence of large number of pages in heap file and thus more I/O operations as compared to slightly bigger page size like 4096 or 8192 as the heap file with such sizes will have lower number of pages reducing the overall I/O operations.

As far as the queries are concerned, as the index is based on "*SDT_NAME*", both the range queries and the other queries should be based on the index. From the four queries ran above, when a index which might have a wider range of records matching or which is more general, say "*2018*" in our case, the time taken is around 10 times more than to that with a lot more specific value like "*3701/02/2021*". This is because large number of leaf nodes are supposed to be traversed when finding a value based on the index 2018, as wider number of leaf nodes will indexes having "*2018*" as a substring in them whereas, the other value being more specific would be relevant to only a few leaf nodes and relevant to even fewer inner nodes thus making the query time a lot faster. As far as the range queries are concerned, they are expected to take longer as compared to for a specific string as large number of inner nodes will be explored which in turn increases the number of leaf nodes to be explored. It is visible that our generalization is correct as range queries over year took slightly more time than the query ran on a single year whereas the range query on certain period of dates or *SDT_NAME* took slightly more time as compared to a single date or *SDT_NAME*.

Thus, it can be concluded that searches are much faster In B+ Tree as compared to B tree or even heap files because keys are the indexes stores in the internal nodes (in our case *NeighbourNode*) whereas the actual values are stored on the leaf nodes (in our case *ChildNode*) whereas in B tree all the key values and the record value are stored in both of the nodes and in the heap file large number of pages are to be searched to find a particular value increasing the search time. As, leaf nodes are interconnected to each other (Each node has a pointer to its *nextNode* as in our implementation), sequential access is provided and searching is even quick and efficient because all the data stored is in leaf node.

APPENDIX

GitHub Repository Link : https://github.com/kaushal-gawri9899/Database-Systems-Assignment-2.git