REPORT

COSC2406 – Database SystemsAssignment #2: MongoDB, Arpache Derby, Java

S3588773
KUNGA KARTUNG

Overview

The experiment entailed the testing of four queries on a Derby and Mongo database. The queries on these two types of databases would be tested with and without a secondary index. Two queries would use the secondary index when available and two would not. This report will contain the methodology used and the results obtained.

Part 1 – Derby

The four queries test on the Derby Database were

1. SELECT BUSINESS_NAME FROM BUSINESS WHERE BUSINESS_NAME LIKE 'Agostini Mechanical Repair'
2. SELECT BUSINESS_NAME FROM BUSINESS WHERE BUSINESS_NAME >= 'M' AND BUSINESS_NAME <= 'P'
3. SELECT COUNT(STATUS) FROM BUSINESS WHERE STATUS = 'Registered'
4. SELECT BUSINESS_NAME FROM BUSINESS WHERE ABN = '98846385136'

To create the secondary index in the database

```
CREATE INDEX B_NAME_ASC ON BUSINESS(BUSINESS_NAME ASC)
```

I chose to create the secondary index on Business name field because for a secondary index to be viable it should be unclustered meaning that the field cannot be a null. It also had to be field that would be frequently searched. As this database is based on businesses, the business name would most likely be the most searched.

Creating the secondary index on the most frequently searched field will help to decrease the number of rows that Derby would have to scan thus improving the search functionality. The downside to this fast search is that it slows the down inserts and updates as now the secondary index has to also be taken into consideration.

The secondary index improving search functionality is evident from comparing the table1.1. and table1.2. which show the results of not using a secondary index and using a secondary index respectively. There is a dramatic increase in searching for business name and also an increase in the range search.

As my secondary index is sorted in ascending order on the business name field. Any queries specific on field name would likely see improvements in searching as the sorted secondary index is a structure in the database which points to its respective in the Business table. This is the reason why finding the count of businesses whose status is Registered and business with the ABN 98846385136 do not see any change, as they are not utilising the secondary index because the database does see it as optimum to use the secondary index.

No Index Derby

Query	Test1 (ms)	Test2 (ms)	Test3 (ms)	Average (ms)
Find Name "Agostini Mechanical Repair"	3873	3797	3732	**3801**
Find Names between "M" and "P" inclusive	1246	1316	1295	**1286**
Find Count of Status "Registered"	6465	6687	6522	**6558**
Find Name with ABN "98846385136"	3924	3799	3751	**3825**

Table 1.1. Table representing the time taken (in milliseconds) to run specified queries in a Derby Database without a secondary index.

Index Derby

Query	Test1 (ms)	Test2 (ms)	Test3 (ms)	Average (ms)
Find Name "Agostini Mechanical Repair"	1	1	1	**1**
Find Names between "M" and "P" inclusive	389	369	370	**376**
Find Count of Status "Registered"	6876	6458	6564	**6633**
Find Name with ABN "98846385136"	3814	3792	3805	**3804**

Table 1.2. Table representing the time taken (in milliseconds) to run specified queries in a Derby Database with a secondary index on Business Name field in ascending order.

Part 2 – MongoDB

The results for the experiment on MongoDB concluded with similar results to part 1.

The queries:

1. db.Business.find ({ Name: “Agostini Mechanical Repairs” })
2. db.Business.find ({ Name: { $gte: “M”, $lte: “P” } })
3. db.Business.count ({ Status: “Registered” })
4. db.Business.find ({ ABN: “98846385136” })

To create index:

```
db.Business.createIndex( { Name: 1 } )
```

Overall MongoDB yields faster results than Derby which was to be expected as MongoDB is a No-SQL database compared to derby which is a relational database. This meaning that MongoDB rely on denormalization so it doesn’t have to join different tables in the database to get the information thus faster.

However it is not always the case as seen when using an index (in this case a secondary index) in an SQL database, it can be just as fast or even faster than mongoDB. This can be seen comparing the first query in table 1.2. and 2.2. The derby secondary index query is faster.

No Index MongoDB

Query	Test1 (ms)	Test2 (ms)	Test3 (ms)	Average (ms)
Find Name “Agostini Mechanical Repair”	2210	1630	1720	**1853**
Find Names between “M” and “P” inclusive	46	112	98	**85**
Find Count of Status “Registered”	1670	1650	1648	**1656**
Find Name with ABN “98846385136”	1890	1980	1970	**1947**

Table 2.1. Table representing the time taken (in milliseconds) to run specified queries in a MongoDB database without a secondary index.

Index MongoDB

Query	Test1 (ms)	Test2 (ms)	Test3 (ms)	Average (ms)
Find Name "Agostini Mechanical Repair"	26	21	22	**23**
Find Names between "M" and "P" inclusive	43	62	53	**53**
Find Count of Status "Registered"	1670	1990	1711	**1790**
Find Name with ABN "98846385136"	2110	2410	2320	**2280**

Table 2.2. Table representing the time taken (in milliseconds) to run specified queries in a MongoDB database with a secondary index on Business Name field in acceding order.

Part 3 – Hash Index

Part 3 consisted of creating a Hash Index to compare search speeds with assignment 1 query search which did not have a hash index.

The initial step taken was to choose an appropriate hash size to avoid excessive collision. As there are about 2.6 million records, the hash size chosen was 3.7 million records. From here the a hash file was initialised and 3.7 million -1 were written onto it, each representing one free slot.
From here the heap file heap.4096 created from assignment 1 was read and from each record the business name was extracted. A variable was kept to keep track of where the current read was in the heap.4096. This was done by incrementing 297 after each record read as that was the start of the new record.
The name obtained was hashed and mod by 3581791. This number was chosen as it was a prime number thus gives a better distribution of the keys. Once the value obtained this was the hash index and in that location in the hash file was where the variable that kept track of where the current read was in the heap file written.
If there was a collision at that point the hash index incremented by 4 (size of int in bytes) which keeping incrementing until the next -1 and here it will be placed. If at the end of the hash file then points back to the beginning of the file. This loops until all records in the heap file read, hashed, seeked that position in the hash file and the record offset written there.

To search for the Business Name, it was first hashed the same way done as above. This resulted in obtaining the hash index which was used to go to that point in the hash file and read that variable stored at that position. This would give us the location of the start of the record in the heap file. After reading that record we compare the names and if they match we found the correct record, otherwise the hash index increments by 4 to go to read the next int which will point to another record and check if that is what we searched for. This keeps happening until we find the record.
As there is no guarantee of a one to one ratio there is bound to be collisions thus to counteract it linear probing has been done. If the number of collisions is too high then the search time will be increased.

Part 3

Business Name to Find	Using A1 search (Non-Hash Index) (ms)	Using A2 part 3 search (Hash Index) (ms)
Warby Wares	**11519**	**3**
The Happy Yak	**11559**	**3**
ASI Outdoors	**11531**	**4**

Table 3.1. Table representing the time taken (in milliseconds) to search for the specified Business Names in search implementations from Assignment 1 part 3 and Part 3 of this Assignment.

From the results obtained from running the two programs, the hash index is far superior. The times for A1 is not 100% accurate as the sample code provided only terminates after searching the whole heap instead of after finding the wanted business name.
However ASI Outdoors is the very last record thus the read time for that will be accurate and that compared to the hash index time shows how much more efficient the hash index is in terms of search speed. This is why hashing is used when dealing with lots of data given the hash function is easy to compute and avoids excessive collisions.