Indexing Techniques in Main Memory Databases

Mukund Manikarnike
Master of Computer Science
Arizona State University
mmanikar@asu.edu; mukunm@gmail.com

Nitish Bhatia
Master of Computer Science
Arizona State University
nbhatia3@asu.edu; bhatia.nitish2905@gmail.com

ABSTRACT

As more and more applications move to the cloud, the number of users interacting with any application is set to increase. This increase will require applications to promise faster query response times which can be fulfilled by porting databases to the main memory. In order to do this, certain optimizations that have been made to databases keeping in mind that they would fit into secondary storage would have to be tweaked so that they do well on the main memory as well. This paper describes a framework that we implemented[1] to allow the database to be ported to the main memory and have indexes use the database. It also describes the several already existing techniques that have been explored in this area, provides a summary of the current research that has already been carried out, how the survey has affected the key aspects that our project aims to explore and measures the performance of a few index structures on the main memory database framework.

CCS Concepts

• Information systems → Database management system engines • Computing methodologies → Modeling and Simulation.

Keywords

Main Memory Databases, Indexes, Slow Storage, Cheap Memory

1. INTRODUCTION

In accordance with Moore's law, we have seen that the size of memory that can fit into one chip has increased by the year and as a result of that we've seen a drop in prices in main memory as well. This has led to an increased amount of research in fitting databases in the main memory instead of secondary storage as it has been done traditionally.

One could argue that main memory databases are similar to disk resident databases which provide caching mechanism to store frequently used data on the cache instead of the disk. However, the key difference here is that the indexing mechanism is optimized for data stored on the disk rather than that on the cache. If the data required isn't on the cache, it still requires fetches to be carried out from the disk.

It makes sense that keeping the data in the main memory instead of maintaining in the disk, will definitely improve the underline application's performance. We have been studying that disk access is the most time consuming process in running a database query. On the software side, disk access involves a system call operation which is expensive in terms of performance. Thus to improve this performance factor, main memory databases proves to be a groundbreaking factor [100].

Often the question is raised to why not improve the caching mechanisms by employing the entire database in the cache to realize the desired performance gains? There are certain RAM driven utilities that exist to create file system in main memory. But the studies done in [100] presents the comparison and explain how caching, data transfer and other overhead sources inherent in a disk-based database (even on a RAM-drive) cause the performance disparity.

Although main memory databases definitely promise faster query retrieval times, there are newer problems in fitting an entire database into main memory in aspects of access times, access patterns, stability and security.

A brief summary of upcoming sections in this paper is given below

- The rest of the **INTRODUCTION** section talks about
  - Main Memory v/s Secondary Storage.
  - Motivation for the work in this paper.
  - Related work in this area.
  - Key Contributions from the work described in the paper.
- The **SYSTEM OVERVIEW** section provides an overview of how the system is designed.
- The **IMPLEMENTATION** section talks about details of how the system was implemented.
- The **RESULTS** section presents the results that were obtained as part of the experiments that were carried out, a theoretical analysis of the paper.
- The **CONCLUSION** section talks about the key takeaway points that have resulted from the work in this paper.

Main Memory v/s Secondary Storage

The key differences and how each of them affect the design of a main memory database are described in this section.

The following are the key aspects in which main memory and secondary storage differ

1. Access Time
   - Since secondary storage is a spindle of magnetic discs, it requires physical movement of the track head to go to a particular track where the data

[1] Source code for the project is available at https://goo.gl/xIIuX8

resides. Hence, the access time with secondary storage is high.
    - However, since main memory is a Random access memory, each address can be randomly accessed and hence, the access time is very low.
2. Access Pattern
    - Since random access memory allows address based access to data, it can be carried out with the use of pointers which isn't possible in the case of secondary storage because of how the disk is organized.
3. Stability
    - Since the RAM is volatile, fault tolerance to back-up data on the RAM becomes much more important than if it were on the secondary storage.
    - However, with increase in availability of Non-Volatile RAMs, fault tolerance on main memory might lose importance.
4. Security
    - If the data resides on the RAM, it becomes much easier for applications to access the data and there could be issues like corruption or data theft in case of buggy or malicious code present in other applications.

Motivation

The motivation for this project stems from the fact that the key differences between main memory and secondary storage have certain implications on any database design.

The implications of the key differences are

1. Concurrency Control
    - This stems from the fact that access times are low, thereby allowing more transactions per second and hence concurrency control becomes more important on main memory.
2. Access Methods (Indexes)
    - This stems from the fact that there are no disk accesses in main memory databases and the indexing mechanisms for the same would need to be altered.
3. Data Representation
    - This stems from the fact that any data on the RAM can be uniquely addressed which isn't possible on disks.
4. Query Processing
    - Query processing on main memory databases focusses more on reducing computation time than on minimizing disk accesses as it is on secondary storage. This stems from the fact that the entire data resides on the main memory.
5. Commit Processing and Recovery
    - Since RAMs are volatile in nature, the usage of logs of transactions and how they are consistently maintained becomes more important than in secondary storage.
    - Since RAMs are volatile, power failures will result in complete loss of data and recovery as well is more important than on disk resident databases.

Each of the implications stated above are a reason for the mechanisms stated to be fine-tuned so that they perform better on main memory databases. Although there are several implications that arise from the key differences stated previously, this paper only focusses on the issues related to indexing on main memory.

Related Work

There has been research work related to creating index structures to efficiently use the query processing in main memory. The initial work started in 1986 with [1] talking about creating a new index structure called T-Trees which worked very efficiently with then available RAM size and computing technology. But later Rao and Ross in [4] showed that T-trees experiences poor cache behavior and works slower than B+ Trees on modern hardware. Other works done in [2], [3] also defines new tree index structures as well as compare existing structures like TPR tress and Adaptive Radix Tree to overcome the bottlenecks while using B-tree for indexing in main memory. Another significant work was done by Boehm in [5] where he proposed Generalized Prefix Tree as a general purpose indexing structure. His work was one of the finalists in SIGMOD programming context in 2009. Kiss-Tree was studied by T. Kissinger and his team in [6] which is an efficient radix tree with three levels but it failed soon as it could only store 32 bit keys. The research is still going on to analyze and build a faster data structure to support indexing while taking other factors like concurrency control, database backup etc.

Key Contributions

The key contributions of this paper are

- As seen in the related work, there has been a lot of work in the area of measuring index performance. However, these indexes have all been measured in an environment where there are a lot of software artefacts and a lot of things needed to be tweaked to keep the environment constant for the purposes of measuring the performance of indexes. In light of this issue, creation of a simple data-store which has the ability to persist data on the main memory without fault tolerance or concurrency is one of our major contributions.
    - ❖ Such a simple system would be useful in cases of experimentation because the item of interest such as an index can be purely tested without interference from other software artifacts which would be inherent in a commercial database with many features.
- Demonstration of how indexes can be built on this data-store to carry out a performance comparison.
- Exploiting the advantages of data stored in the memory by using pointers to the tuples in the data-store in the indexes.

2. SYSTEM OVERVIEW

The architecture of the system is as illustrated in Figure 2. The lifecycle of how the system operates is as shown in Figure 2

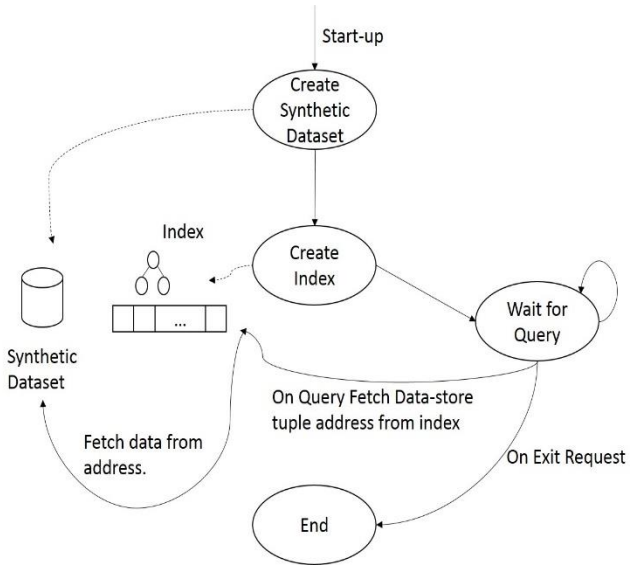


Figure 1 Operation lifecycle

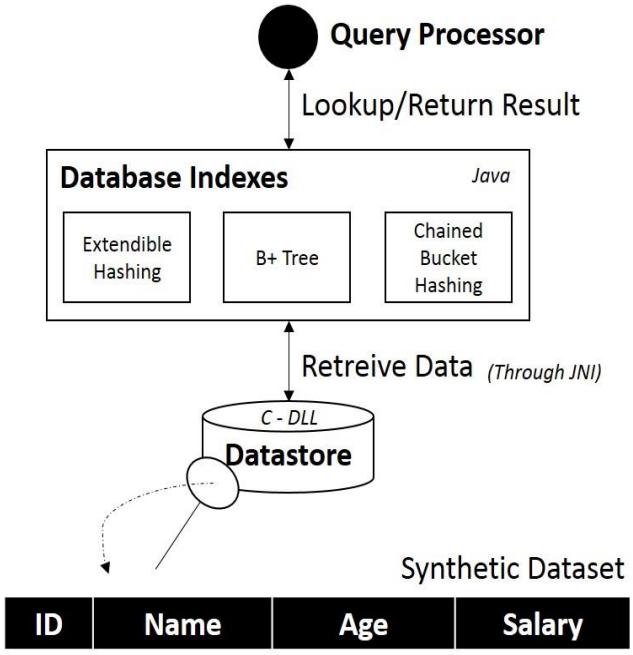


Figure 2 System Architecture

The following sub-section describe the design of the data-store and the indexes that were used as part of the work in this paper.

2.1 Data-store

The data-store allocates a large chunk of memory on start-up and makes this memory region available through the use of some APIs to read and write data to/from it. The data-store provides for the storage of a single table with the parameters.

- ID
- Name
- Age
- Salary

The main functionalities of the data-store are as listed below

- Allocation of the required region of memory on start-up
- Deallocation or Destruction of the memory allocated
- Reading and writing into the exact regions in memory where the data is stored.

The data-store does not provide concurrency or fault tolerance because the main focus was to support the features required indexes.

2.2 Indexes

2.2.1 B+ Tree

An n-ary tree with a large number of children per node, B+ tree is similar to its parent B tree but with a distinction that each node contains only keys (and not Key-Value pairs). Because of a very high fan out (which is the number of pointers to child nodes in a node), B+ trees are very efficient data structure for block-oriented storage context mainly file system as it reduces the number of I/O operations required to find an element in the tree [11].

Multiple main memory systems have incorporated B+ Tree as their primary index structure. Systems like H-Store, Mongo-DB, SAP Hana (uses CSB+ Tree) have been using B+ tree and research is still going to improve the systems with respect to concurrency control, fault tolerance, etc.

The leaves which are the bottom most index blocks of the B+ tree are linked to each other using linked lists. This is done to make the iteration and the range queries to be simpler and more efficient. Although it does not increase the space consumption or maintenance on the tree. Because of all the properties mentioned, B+ tree forms a very efficient index structure and an efficient structure for housing the data itself [11]. The only concern while using B+ trees for in memory data indexing is that the reasonable choice for the block size would be the size of processor's cache line. This does not affect us as far as our project is concerned because of the way we constructed our dataset.

One concern with the B+ tree is that it stores all the data in the bottom leaves of the tree. It is a waste of space if we already have the database in the main memory. Otherwise, B+ trees are good for memory use as the storage utilization is good because the pointer to data ratio is small. Also with good data representation techniques like using pointer instead of actual data can make B+ trees more effective in main memory. In our project we have replaced the data with the address of the data in the main memory which minimizes the storage needed to generate the B+ tree index structure.

2.2.2 Chained Bucket Hashing

CBH consists of a hashing function, a hash table and a linked list of record identifiers. The hash function works by transforming the key-value pair into an address-value pair whose range is the cardinality of the hash table. There is a time when the all the keys of the hash table are exhausted or the expected chain length exceeds a threshold value, but it is possible to expand the hash table by just increasing (usually doubling) the hash table. But the expansion of the hash table requires a complete reorganization of the hash table which involves re-linking of the RIDs which will definitely cause a dip in the performance of the whole system [14].

Instead of storing the data item directly in the hash table as done in the linear hashing, each hash table entry contains a reference to

a data structure, e.g. a linked list (as in our project), or a balanced search tree [16]. It is a static hashing structure which is very efficiently used both in the in memory and on disk. The advantage of this index structure being static is that it is very fast and it never reorganizes its data. But it is also a disadvantage when it comes to space consideration. Also, it is not advantageous in dynamic environments because the amount of space needed to fill in the hash table is needed before the table is filled. Guessing the space limit is not easy as if the estimated size is too small, it can directly affect the performance of the whole system where as if the estimated size is too large, then a lot of space is wasted [14]. In our project, we have used linked list as the data structure for chaining. Every node in the list stores an object which contains a key and the data which is the address to the database tuple for the key. This saves the space in situations where the data tuple is very large.

2.2.3 Extendible Hashing

The problem of guessing the space in the above hashing technique is solved by the extendible hashing system where the table grows with the incoming data, therefore the table size does not need to be known in advance. Extendible hashing represents as a dynamic hashing which work by the hash node splitting into two whenever an overflow occurs. The directory grows in the power of 2 which means that the node doubles every time there is an overflow and has reached the maximum depth for a particular size [10]. The hash system is treated as a bit string which uses a tree for the bucket lookup.

Variety of main memory database systems are using different versions of hashing techniques. Systems including H-Store, Hekaton HyPer, MemepiC, RAMCloud, Redis, Memcached, and MemC3 are using various hashing indexes like Linear hashing, Extendible hashing etc.

For a successful implementation for the extendible hashing system, the hashing function needs to be very efficient as any node can cause the directory to split which can in turn cause the directory to grow very large.

The research done in [14] proposes a new hashing technique called Extendible Chained Bucket Hashing (ECBH) which is a seamless combination of Chained Bucket Hashing and Extendible Hashing. The basic structure of the ECBH replaces the leaf nodes in the Extendible hashing with chained bucket hash table and RID chains. With the search time still being O(1), the ECBH effectively inherits the high performance attributes of the Chained Bucket Hashing with the gradual extendibility powers of the Extendible Hashing. Due to time constraint, we could not implement this index structure.

2.3 Query Processing

The query processing as indicated in the operation lifecycle in Figure 1 is one that touches both the indexes and the data-store. Given a query that requires a certain search to be performed, the query is processed as follows

- The required data is looked-up in the index to retrieve the address of the tuple where the data resides.
- This address is then used to fetch the data from the data-store.

This process makes the retrieval much faster because it only needs one search on the index and that will directly lead to the data in the data-store.

2.4 Data-store – Query Processing Interface

As and when an insert to the data-store is done, the data-store returns the address at which the tuple is stored. In order to exploit the fact that the data-store also resides in main memory, the tuple address is stored in the index. When the user fires a query that requires one to search the database, this feature comes to use in the following way.

- The search result from the index directly results in the address of the tuple where the data is stored.
- This address can directly be used to fetch data from the data-store.

An illustration of how this is carried out is shown in Figure 3.

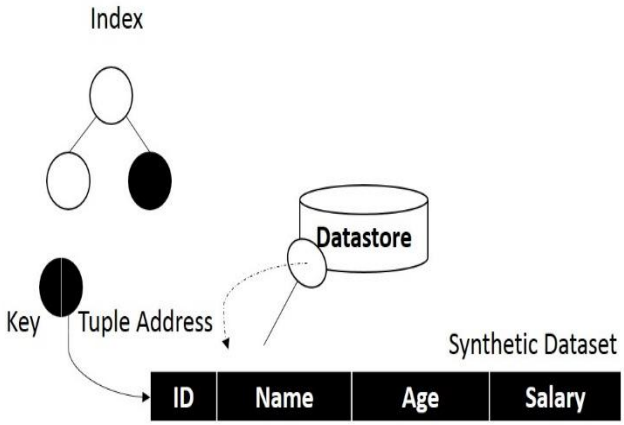


Figure 3 Data-store – Query Processing Interface

3. IMPLEMENTATION

This section talks about how each of the tasks mentioned in the **SYSTEM OVERVIEW** section are achieved in the implementation. The Data-store is implemented in C and the Indexes and queries are implemented in Java. The reasons behind this are two-fold

- Implementation of the data-store in C makes it easier to allocate and de-allocate memory at will
- Implementation of indexes and query processor in Java was primarily to make the system extensible for the future in the case of requirement of a demonstrable application.

3.1 Data-store

The data-store implemented in C is built as a DLL to enable smooth communication with the indexes in Java.

3.1.1 Memory Organization

The data-store allocates 418MB of memory on the RAM and organizes this entire region into several tuples starting from the start address where the memory is allocated. The organization look as shown in Table 1.

Table 1 Data-store Memory Organization

Tuple 1	Tuple 2	…	Tuple N
32 Bytes	32 Bytes		32 Bytes

Each tuple in the memory contains the fields as mentioned in the **SYSTEM OVERVIEW** section. The amount of memory allocated for each attribute in a tuple is as shown in Table 2

Table 2 Tuple Attribute Memory Allocation

ID	Name	Age	Salary
4 Bytes	23 Bytes	1 Byte	4 Bytes

The reasons for this allocation are stated below

- The ID is supposed to identify each tuple uniquely and act like a primary key which means that it needs to support as many values as possible. In this case, it supports $2^{31} - 1$ values which is good enough for experimentation purposes in this paper.
- Name supports a string of length 23 which is again good enough for the purposes of experimentation.
- The Age cannot go beyond 255 in realistic scenarios and hence allocating 1 byte would be sufficient for this field.
- The Salary field also doesn't go beyond the allocated maximum range for a 4 byte value. Hence, this number was chosen.

3.1.2 Read-Write Abstraction

The data-store provides the following APIs which the indexes can make use of through the Java native interface.

- init_data_store
  - Allocates memory using malloc, resets it to zero and returns the pointer to the start of the memory region.
- write_data
  - Takes the ID, Name, Age and Salary as inputs
  - Writes these values to the current available region in memory.
  - Returns the address of the tuple.
- update_data
  - Takes the address of the tuple to update, the ID, Name, Age and Salary to be updated.
  - Writes the new values to the mentioned address.
  - Returns the address of the tuple
- read_data
  - Takes the address of the tuple to read from, a DataTuple object in which the result needs to be populated
  - Reads the data from the provided address. Updates the DataTuple object with the data obtained and returns.
- destroy_data_store
  - Destroys the allocated memory region by calling the free function on the start address of the allocated memory region.

3.2 Indexes

Each data structure was implemented in Java programming language on Eclipse Juno Service Release 2.

3.2.1 B+ Tree

The following are the classes in which the B+ tree implementation resides

- BTree Class
  - This implements the basic functions including insert, delete, and search functions.
- BTreeNode Class
  - It class helps in creating the nodes when a request of new node comes in.
- BTreeTest Class
  - Employs the mainExecution function which takes an IntegerBTree object and creates a BTree with the given input keys and data

3.2.2 Chained Bucket Hashing

The following are the classes in which the Chained Bucket Hashing implementation resides

- HashChain Class
  - This includes the main logic for the implementation of the chained bucket hash. It uses an object class called Link.
  - The mainExecution method in this class takes a HashChain object and creates the index structure and further populates populates it with the given data. This java class also includes helper functions like insert, delete, search, and display.
- Link Class
  - It creates objects with key and data attributes.

3.2.3 Extendible Hashing

The implementation of extendible hashing includes splitting a full page according to the (local depth) bit and then reloading all the key-values after hashing each key and using the d bit of each hash to see which page to allocate to [17]. The page size is static right now and can be hardcoded according to the need of the experiment scenario.

3.3 Synthetic Data Creation

The mainExecution() function in the index structures calls the GenerateData.java class which generates random data to populate the index structures. The random data includes picking up a random name from a given set of ten names. Random class of java is used to implement this. Also, random age between 22 and 60 years is generated using the random function and similarly the salary is generated between 20,000 and 500,000. The mainExecution function also calls the write_data() function of the DatastoreInterface.c class which returns the address in the main memory where the generated tuple is stored. The key and the address (which serves as the value) are then stored in the index structure. Similarly while reading the data from the data store, read_data() function of DatastoreInterface.c class is called with the address and DataTuple object as input which in turn stores the data in the DataTuple object.

3.4 Query Processing

Query processing is entirely carried out in the Java implementation within one class IndexOps. The procedure for each query is simple

- It uses the get method from the index that is currently being used. Once this is done, the address to the required tuple obtained.
- It then performs a read from the data-store using the read_data API that the data-store provides.
- It obtains a DataTuple object on performing this call which contains the data of interest.

3.5 Data-store – Index Interface

Certain challenges in the implementation of the interface between the index and the data-store are described in this section.

Representation of Pointers

The data-store often returns pointers when a write or an update is performed which need to be stored in the index. This becomes a challenge because of lack of pointers in Java. In order to solve this problem, the address returned from the data-store is treated as an integer and stored in the index.

Passing Objects from the Data-store

Since C doesn't have an object representation returning a large collection of data such as a tuple in the form of an object becomes a problem. Such an issue has been handled in this system by using the Java Native Interface's C APIs to store different attributes obtained from memory in the attribute of the object passed.

4. RESULTS

In order to perform our experiments, the system used had the following configuration

- OS – Windows 8.1 64-bit
- Processor – Intel® Core™ i7-4510U CPU @ 2.00 GHz (4 CPUs), ~ 2.6 GHz
- RAM - 8192 MB

The following subsections provide theoretical and experimental analysis of the results obtained from the work in this paper. The results are largely measuring the query time for different queries. The queries that were run were insert query, point query, sweep query, range query on datasets of different sizes and the factors measured were index creation time, point query time, sweep query time and range query time.

4.1 Experimental results

A summary of the results obtained is provided in the following subsections. The actual numbers obtained and their graphical representation are provided in Figure 5, Figure 5and Table 3.

4.1.1 Index Creation

This experiment includes creating a variable number of nodes using the different index structures. The cost of a series of insert operation depends on the search time and various other operations depending on the index structure for example number of tree rebalancing operations called, number of memory allocation operations, amount of data moved etc. The results of this test are shown in Graph 1.

The x-axis includes the number of nodes which is the number of records in the dataset and it is set from 0 to 10,000 records. The y-axis shows the time taken to create the number of records. B+ tree takes the most time while creating the nodes because it makes more memory allocation calls and uses a costly rotation operation and some intra node data movement to rebalance the tree after some insertion.

Chained bucket hashing and extendible hashing performs very similar to each other but still we can see that extendible hashing performs the best in our results which can be explained from the movement caused because of the linked list chaining in chained hashing. Also, it might depend on the hashing functions used.

4.1.2 Search Operation

The search operation is applicable for the range query because a search needs to be done. The search operation is divided in three queries which includes a point query, sweep query and a range query. Point query aims at getting a result given an input. For example in our experimental scenario, the point query is - Find the salary of the employee with a given ID.

Extendible hashing works the fastest as the hashing techniques have a fixed cost of linear search of the node and any associated overflow buckets. For small node size the index structures perform almost equivalent.

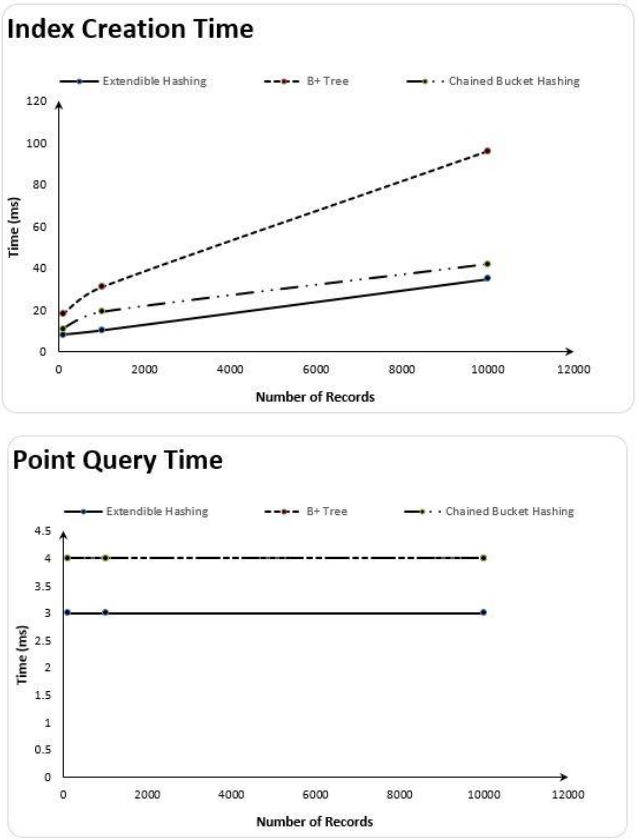
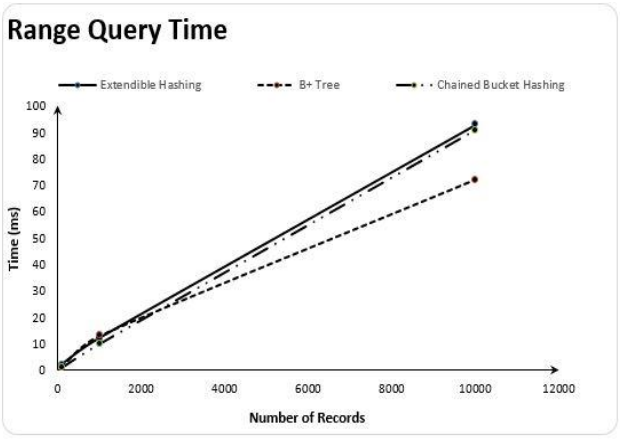


Figure 4 Query Time Graphs - 1

Sweep queries include traversing through the whole dataset of get a result. Sweep query in our experiment - Find the average salary of all the employees in the dataset. As we can see in the Table, the sweep query time of every index structure increases as the number of records increase which is expected. B+ trees shows the worse performance among the three index structures because it requires several binary searches to perform the operation.

Range queries includes doing an operation over a certain range of data in the dataset. Range query in our experiment - Find the average of the salaries of all the employees between age 20 and 40.

A range query usually consists of two parts: the search for the lower bound value which is the age 20 years in our case, and then the scan through the data structure until the upper bound value of the range is found or surpassed which is age 40 years in our case. B+ Trees performs better than the hashing because the data is indexed and sorted in the tree which makes it easier to sweep through the data between a set ranges. It just has to find the lower bound and the upper bound which takes lesser time than the hashing techniques.



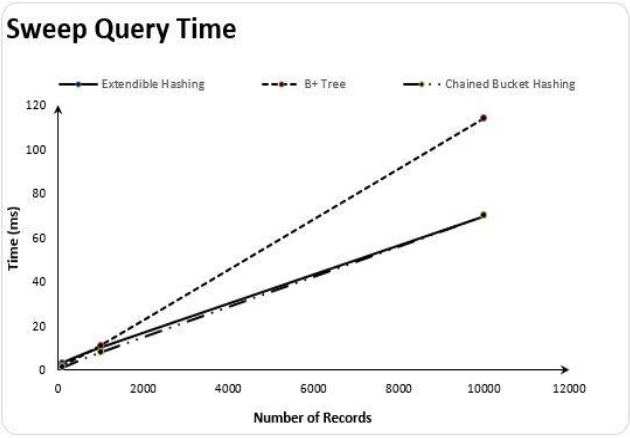


Figure 5 Query Time Graphs – 2

Table 3 Query Times

Index Type	No. of records	Index Creation Time (ms)	Point Query Time(ms)
Extendible Hashing	100	8	3
	1000	10	3
	10000	35	3
B+ Tree	100	18	4
	1000	31	4
	10000	96	4
Chained Bucket Hashing	100	11	4
	1000	19	4
	10000	42	4

Index Type	No. of records	Sweep Query Time (ms)	Range Query Time(ms)
Extendible Hashing	100	3	2
	1000	10	12
	10000	70	93
B+ Tree	100	2	1
	1000	11	13
	10000	114	72
Chained Bucket Hashing	100	1	1
	1000	8	10
	10000	70	91

5. CONCLUSION

This section talks about the key learnings from the research that was carried out as part of work for this paper and the learnings from the implementation and experimentation carried out.

5.1 Conclusions from research

This section summarizes all ideas explored as part of the research for the project and how they've been used to formalize the approach that this project is going to take.

5.1.1 Summary of Existing Systems

Existing systems and researches have considered aspects similar to the aspects we aim to explore as described in this survey paper. A summary of what main memory databases have been analyzed as part of research in this area and what aspects about these have been considered is provided in Table 4.

Table 4 Summary of Existing Systems

Systems Explored	Aspects Considered
H- Store	Access Methods
Hekaton	Data Representation
HyPer/ScyPer	Query Processing
SAP HANA	Concurrency Control
	Fault Tolerance

5.1.2 Key Learnings from the Research

The main point that this research highlighted was what key aspects need to be evaluated while porting a database to the main memory. Having summarized the key aspects, the paper also talks about what aspects could be considered for experimentation given a time constraint in light of this project. Table 5 gives a summary of what aspects are experimented with as part of the project and the reasons for the same.

Table 5 Summary of Project Choices

Aspect	Yes/No	Reason
Access Methods	Yes	Because it is a key aspect to data insertion and look-up in any database.
Data Representation	Yes	Because it is key to illustrate data access on successful look-ups in an index data structure.
Query Processing	Yes	Simple query processing is required to show the difference in performance of difference indexes. However, complex queries like joins are not experimented with in this project.
Concurrency Control	No	Due to lack of time, this feature isn't experimented with.
Fault Tolerance	No	Due to growth in non-volatile RAMs, this might not be an area that promises a lot of growth in the future.

5.2 Conclusions from Experiments

In this paper we compared and analyzed different index structures including B+ Trees, Extended hashing and Chained Bucket Hashing in main memory environment in terms of performance. We created our own data store instead of using a commercially available data store. The main reason behind doing this was to test the index structures for their performance without interference from other software artifacts. For our experiments, we created a data store that includes read and write APIs to store employee data. We also created three index structures and tested their performance based on index creation time, point queries, sweep queries and range queries. We found that B+ Trees perform the worse while creating indexes and that hashing structures takes the least time.

For sweep queries, B+ tree performs the worst among the three index structures because it requires several binary searches to perform the operation, whereas in case of the range queries, B+ Trees performs better than the hashing because the data is indexed and sorted in the tree which makes it easier to sweep through the data between a set ranges.

5.2.1 Key Learnings from the implementation

The key learnings from implementing this project were

1. We learnt how a full-fledged database system has to be designed if designed from scratch for the purpose of porting it to main memory.
2. Having a lower layer to the database implemented in a lower level language and having the other functionalities in higher level language like Java was something that we had initially thought out and it turned out really advantageous because a lot of experimentation activities become a lot easier. By ensuring this split between languages, it also doesn't overuse memory because those functionalities have anyway been pushed to a lower level language.
3. The results obtained have been in line with the expected outputs of implemented data structures which is satisfying because the implemented data-store is able to produce previously proven results.

6. FUTURE WORK

The possible activities that can be carried out on top of the work that is done as part of this paper can be split into 2 separate areas.

1. Indexes and Query Processing
2. Data-store

6.1 Indexes and Query Processing

The data-store has been implemented completely and it is scalable to large data sizes subject to the main memory size available. Currently simple queries have been supported since the focus of the work has been to create a data-store that can write/read to/from the main memory and demonstrate the usage of certain indexes on main memory. Since, the framework is well tested for small data sizes, complex indexes and queries can be implemented on the existing data-store to measure their performance. This will provide trustworthy results because of the fact that the data-store is simple and won't have any interference from other software artefacts.

The queries are now generated through programs written which are not very flexible and will need changes to the implementation to support new queries. Future work in this area could include incorporating an SQL interface to the main memory database in order to allow the user to make more generic queries.

6.2 Data-store

The following are the possible activities that can be carried out with respect to the data-store in the future

Issues with data-store

- The data-store currently has a size limitation issue. It can only allocate 418 MB on the RAM after which a write to it results in a segmentation fault. This leads to a data-store with very less size.
- The data-store also crashes when more than ~14,000 records are written to it. This has something to do with the access through Java Native interface.

If the above issues are resolved, then the data-store is perfectly scalable and experiments with large data-sets can be performed.

Additions to the Data-store

- Currently, the data-store is fixed to create a static table which cannot be changed.
- The data-store also only supports a simple read/write.

If dynamic table creations can be done through the user, this will make the data-store more flexible to test different datasets. If more features like concurrency and fault tolerance can be added to the data-store, it can work like a full-fledged main memory database which retrieves data from the storage on start-up, runs as a background process throughout its lifetime and backs up the data to the storage when the database stops.

7. ACKNOWLEDGMENTS

We would like to express our sincere thanks to Dr. Mohamed Sarwat for his efforts in discussing interesting topics in class that motivated us to do well in this project. He introduced us to several papers in the database area which were truly thought provoking. The discussions in class were very engaging and we really did learn a lot.

8. REFERENCES

[1] T.J. Lehman, et al., Query Processing in Main Memory Database Management Systems, in SIGMOD, 1987

[2] Viktor Leis, Alfons Kemper, Thomas Neumann, The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases

[3] Laurynas Biveinis, Simonas Saltenis, Towards Efficient Main-Memory Use For Optimum Tree Index Update

[4] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," in VLDB, 1999

[5] M. Bohm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner, ¨ "Efficient in-memory indexing with generalized prefix trees," in BTW, 2011

[6] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner, "KISS-Tree: Smart latch-free in-memory indexing on modern architectures," in DaMoN workshop, 2012

[7] Per-Åke Larson, Spyros Blanas, Cristian Diaconu Craig Freedman, Jignesh M. Patel, Mike Zwilling, High-Performance Concurrency Control Mechanisms for Main-Memory Databases, Microsoft, University of Wisconsin – Madison

[8] Hector Garcia Molina, Member, IEEE and Kenneth Salem, Member, IEEE, Main Memory Database Systems: An Overview

[9] Key Differences between Main Memory and Hard Disk http://pc.net/helpcenter/answers/memory_and_hard_disk_space

[10] Tobin J. Lehman, Michael J. Carey, A Study of Index Structures for Main Memory Database Management Systems, SIGMOD, 1987

[11] B+ Tree, https://en.wikipedia.org/wiki/B%2B_tree

[12] Radix Tree, http://kukuruku.co/hub/algorithms/radix-trees

[13] Viktor Leis, Alfons Kemper, Thomas Neumann, The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases

[14] Pyung-Chul Kim, Kee-Wook Rim, Jin-Pyo Hong, Extendible Chained Bucket Hashing from Main Memory Databases

[15] Kong-Rim Choi and Kyung-Chang Kim, T*-tree: a main memory database index structure for real time applications.

[16] Hashing – Explained

http://www.itu.dk/research/theory/SearchEngine/E2004/mirrors/wiscdocs/notes/HASHING.html

[17] An Extendible Hashing implementation

https://en.wikibooks.org/wiki/Data_Structures/Hash_Tables#A_Java_implementation_of_extendible_hashing

[18] Main Memory v/s RAM Disk Databases

http://www.bloke.com/linux/ramdisk/memorybenchmark.html

[19] B+ Tree Implementation in Java

http://jxlilin.blogspot.in/2013/11/b-tree-implementation-in-java.html