





# Indexing techniques and structured queries for relational databases management systems

Isah Charles Saidu <sup>a,\*</sup>, Musa Yusuf <sup>b</sup>, Florence Chukwuemeka Nemariyi<sup>c</sup>, Ayenopwa Comfort George<sup>c</sup>

<sup>a</sup>Department of Computer Science, Baze University, Abuja, Nigeria

<sup>b</sup>Department of Computer Science Bingham University, Nasarawa, Nigeria

<sup>c</sup>IT Unit, Bingham University, Nasarawa, Nigeria

## Abstract

Indexing has long been used to improve the speed of relational database systems, and choosing an adequate index at design time is critical to the database's efficiency. In this study, it was demonstrated empirically that data access time and data insertion time for moderately large datasets are influenced by the index chosen at design time. However, deletion time is approximately the same. As a result, regardless of the query optimization strategy utilized at runtime, record access/insertion time depends on the type of index employed at design time. This paper presents a comparison of BTree indexes with Hash indexes. It was demonstrated empirically that insertion is substantially faster with the Hash index than with the Btree index, at the expense of a larger Hash index file size. The Btree index is slower due to the rebuild time of Btree indexes during insertion. The empirical results of this study complement that of theoretical results for both Btree and Hash indexes. On the other hand, hash index files are large, restricting their use for applications with rapidly increasing dataset sizes; thus, a tradeoff employing Hash index or Btrees is required. In general, this study proposes Hash indexes for small dataset applications and Btree indexes for large dataset applications on systems with limited memory.

DOI:10.46481/jnsps.2024.2155

**Keywords:** Relational database systems, Btree index, Hash index, Structured Query Language

## Article History :

Received: 21 May 2024

Received in revised form: 14 July 2024

Accepted for publication: 10 September 2024

Published: 27 September 2024

© 2024 The Author(s). Published by the [Nigerian Society of Physical Sciences](#) under the terms of the [Creative Commons Attribution 4.0 International license](#). Further distribution of this work must maintain attribution to the author(s) and the published article's title, journal citation, and DOI.

Communicated by: Oluwatobi Akande

## 1. Introduction

Databases usually contain voluminous amount of data such that queries for a subset of the data doesn't have to require a scan of the entire volume of available data. According to Ref. [1], an Index for a file in a database system works in much same

way as the index of a textbook, if a particular topic (specified by a word or a phrase) in the textbook is to be searched, the index located at the back of the book is searched and the page number corresponding to the entries in the index is used to locate the particular topic searched.

In general, indexes can be categorized into two types: primary and secondary indexes. In the case of a primary index, the tuples in the relation are ordered according to the indexed column - which is the primary key column, as a result, primary indexes are created on the primary key of the table. However, this

\*Corresponding author: Tel.: +234-807-992-5958

Email address: [charles.isah@bazeuniversity.edu.ng](mailto:charles.isah@bazeuniversity.edu.ng) (Isah

Charles Saidu )

is not the case for a secondary index [2]. Instead, secondary indexes are created on attributes other than the primary attribute, which also implies that they can contain duplicate entries.

Creating indexes is essential as indexes are the most commonly used strategy for speeding up query response. It is therefore critical to appropriately select the columns to be indexed, as well as the type of indexing technique, as the quality of index affects the query response.

In relational databases, a query optimizer software is responsible for analyzing queries and choosing the most efficient way to access information [3]. The query optimizer selects the index record that will best retrieve records faster. Therefore, it is worth mentioning that the query optimizer assumes the existence of already indexed column records. The query optimizer makes query selection decisions by either scanning the entire table of records or using defined indexes within the table, as a result, the choice of the indexed column to use during query planning is essentially an optimization problem.

For example, consider the simple query “select \* from city where name = ‘Abuja’”. It was assumed that the table consists of attribute “name” and depicts the query plan (Figure 1). Once this query is submitted, the query optimizer has to choose between performing a full scan or using filtered pointers present in the index of attribute “name” of the table - if such an index exists. This step is known as the query execution plan step. The execution plan is computed from the expected response time of each index generated over time [4]. Now consider the case where the attribute “name” is properly indexed – meaning the tuples would have been ordered based on the entries of this attribute, then if the right choice of index is made involving the index entries for “name”, the response time would be much less compared to a full scan of the table. In Figure 1, the key “Abuja” would be found in the “name” index file and a pointer to the actual record in the table can be used to directly retrieve the record.

The example described in Figure 1 assumes columns are already indexed but it is worth mentioning that the performance of these indexing techniques varies, so the decision of what type of indexing technique to use has to be decided at database design time. An important aspect of software development is information storage design – one that is dynamic enough to withstand the growing volume of data and can also return results of queries over a large volume of data in record time. However, choosing a particular index structure based on the kind of data and size of data versus the available storage engine is non-trivial. Random investigations from these studies have shown that most developers or system analysts just design relational schema without factoring in the kind of indexes to apply on certain tables.

Therefore, the entirety of this work is focused on design time choice for indexes. An extensive empirical comparative study was conducted to suggest the right choices of indexes, based on the kind of data type and relationship imposed on the tables. Given that these database indexes vary in terms of time and space complexity, it is then obvious that a very good relational schema with a badly indexed table will lead to poor query performance over a long period, especially as the vol-

ume of data increases. The motivation for this research lies in the quest to provide a comparative framework upon which relational database design can be referenced during the design phase of relational databases.

In this paper, the performance of BTree vs Hash-based indexes was compared as a function of response time and space complexity, using MySQL relational engine. The specific contributions of this paper are in the presentation of empirical results quantifying the data access time, data insertion time, data deletion time, and space complexity between BTree and Hash indexes, on different datatypes of different sizes and different numbers of records. The goal of the study is to provide guidelines for database administrators when making decisions on the type of index structure to use, especially in MySQL databases.

This article is structured as follows: section 2 presents the index of the database while section 4 is the literature review of the key concepts discussed within the body of this work. This study narrowed the discussion to indexes supported by MySQL database in section 5 while section 6 discusses the methodology for this empirical study. Section 10 discusses the results of the experiments of this study. The summary, conclusions, and recommendations are presented in section 11.

## 2. Database index

An index is a structure that exists solely to speed up searches to its underlying dataset. According to Ref. [1], an index can be thought of as a collection of data entries with an efficient method for locating all data entries with search key value  $k$ . Each such data entry has sufficient information to allow retrieval of (one or more) data records with search key value  $k$ . According to Ref. [5] an index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to look through every row in a database table every time it is accessed. An index can be built utilizing one or more columns from a database table, allowing for both quick random lookups and efficient access to ordered items.

In designing the logical structure of a database – also known as the *schema*, care should be taken on the type of index structure that should be imposed on a column. Some databases extend the power of indexing by letting developers create indices on functions or expressions. For example, an index could be created on `upper(last_name)`, which would only store the uppercase versions of the `last_name` field in the index. Another option sometimes supported is the use of partial indices, where index entries are created only for those records that satisfy some conditional expression. A further aspect of flexibility is to permit indexing on user-defined functions, as well as expressions formed from an assortment of built-in functions. In general, the indexes can be categorized according to their architecture:

### 2.1. Non-clustered indexes

In a non-clustered index structure, the data is present in an arbitrary order but the logical ordering is specified by the index.

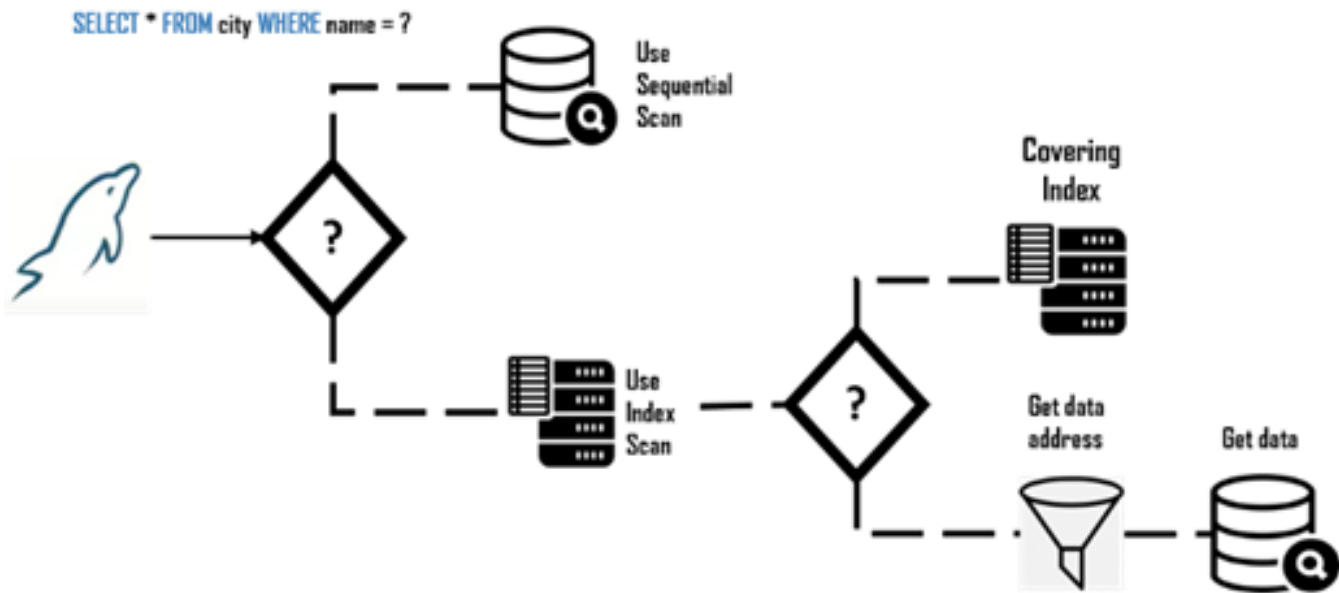


Figure 1. Query planning.

The data rows may be spread throughout the table regardless of the value of the indexed column or expression.

## 2.2. Clustered indexes

In a clustered index structure, each index is associated with a particular search key. Just like the index of a book or library catalog. It is an ordered index structure and it associates with each search key the records that contain it. According to Ref. [5], clustering alters the data block into a certain distinct order to match the index, resulting in the row data being stored in order. Therefore, only one clustered index can be created on a given database table. The overall speed of retrieval can greatly be increased using a clustered index but usually only where the data is accessed sequentially in the same or reverse order of the clustered index. In practice, files are rarely kept sorted since this is too expensive to maintain when data is updated [6].

## 3. Index structure architecture

Most index structures usually follow two well-known design structures. They can be either Hash-based or Tree-based index structures as shown in Figure 2 and Figure 3.

### 3.1. Hash-based index structures

In Hash-based index structures, records are organized using a technique called Hashing to quickly find records that have a given search key value. According to Ref. [1], addresses of disk blocks containing the desired record are obtained directly by computing a function on the search key value of the record. Formally, let  $K$  denote the set of all search key values, and let  $B$  denote the set of all bucket addresses. A Hash function  $h$  is a function from  $K$  to  $B$ . To insert a record with the search key  $K_i$ ,

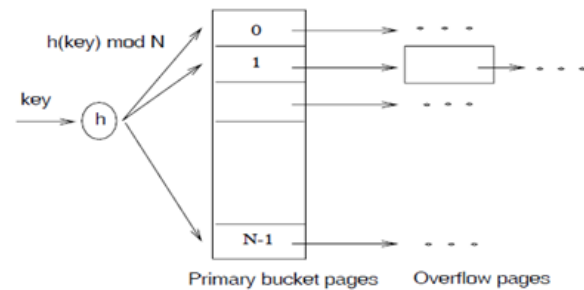


Figure 2. Concept of Hashing [6].

compute  $h(K_i)$  is executed, which gives the address of the disk block.

### 3.2. Tree-based index structures

A tree is a widely used data structure that simulates a hierarchical tree structure with a root node known as the parent node and sub-nodes known as child nodes. Generally, the tree data structure is represented as a set of linked nodes. A typical example of a tree-based index structure is the B-Tree, B+-Tree, R-Tree, R+ Tree [7], and the *Indexed Sequential Access Method* (ISAM) [8].

The ISAM [8], was originally developed by IBM for main-frame computers. It contains an index of tables whose elements are pointers allowing individual records to be retrieved without having to search the entire data set. This index is small and can be searched quickly usually using a binary search algorithm, thereby allowing the databases to access only the records it needs. When an ISAM file is created, index nodes are fixed

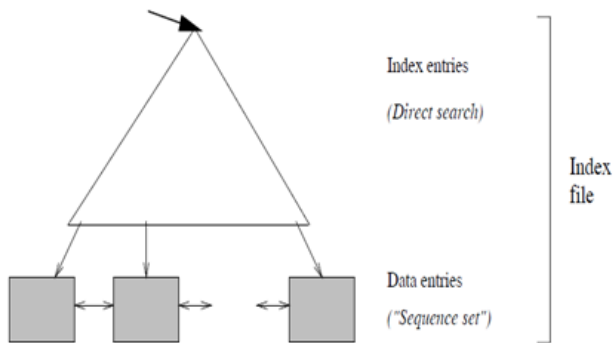


Figure 3. B+ Tree.

and their pointers do not change during insertion and deletion of records that occur later. This is the reason why ISAM are static structures unlike the B+ Trees (discussed in the next paragraph). As a consequence of this fixed or static index structure, if insertions to some leaf node exceed the node's capacity, new records are stored in overflow chains. If there are many more inserts than deletions from a table, these overflow chains can gradually become very large, and this affects the time required for retrieval of a record.

The B+ tree index structure is widely used. It is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries.

Since the tree structure grows and shrinks dynamically, it is not feasible to allocate the leaf pages sequentially as in ISAM, where the set of primary leaf pages are static. To retrieve all leaf pages efficiently, it links them using page pointers. By organizing them into a doubly linked list, it can easily traverse the sequence of leaf pages (sometimes called the sequence set) in either direction [1]. This structure is illustrated in Figure 3.

#### 4. Literature review

Index selection has been extensively studied in Refs. [2, 9–11], albeit from an optimization standpoint. The goal in most of these studies is to develop an efficient algorithm for selecting optimal indexes among the various columns already indexed at design time. By assuming each index response time as a dependent variable, the index selection problem is in effect cast as an optimization problem with objective/cost function being specifically crafted linear combination of index columns.

In the work by Ref. [2], they formulated a general cost model that incorporated dependency between primary and secondary index operational statistics. Due to the complexity of their cost function, they split the objective into two: one involving primary index selection and the other secondary index selection. Because these problems are mutually dependent, their formulation raised the question of “what order should primary and secondary index selection be based on?” – to which they discovered that primary index selection supersedes.

The work by Chaudhuri *et al.* [9] presented a formal statement for the index selection problem and showed that it is computationally “hard” to solve or even approximate. However, they developed a new algorithm based on reformulating the objective as a knapsack problem. The novelty of their approach lies in an LP (linear programming) based method that assigns benefits to individual indexes. For a slightly modified algorithm, that does more work, they proved specific guarantees about the quality of their solution (As these specific guarantees are not relevant to this research therefore readers are recommended to Ref. [9] for more details).

In the work by Schlosser *et al.* [10], they introduced a recursive strategy that does not exclude index candidates in advance and effectively accounts for index interaction. They used a large real-world workload to demonstrate the applicability of their approach for both large and small databases.

More recently Yadav *et al.* [11] introduced AIM (Automatic Index Manager), a configurable index management system that identifies impactful secondary indexes for SQL databases to efficiently use available resources such as CPU, I/O, and storage. It proposes a novel way of exploring the search space of candidate indexes such that the execution cost of the workload running on it can be minimized under certain constraints.

Works by Refs. [12, 13] surveyed the different index selection approaches. In [12], they discussed the big data requirements for indexing and offered a comparison study on different indexing selection techniques such as non-artificial intelligence-based selection techniques, artificial intelligence, and collaborative-based techniques.

The work of Shin *et al.* [14] presents a novel approach to optimizing keyword search in relational databases by leveraging an inverted index data structure and proposing a multi-way skip-merge join algorithm. The contribution of the paper relates to reducing search time delays by minimizing unnecessary comparisons during query execution. The utilization of the gallop search within the merge-join method demonstrates a significant improvement over the traditional method. However, the paper lacks of comprehensive empirical evaluation across diverse datasets and real-world applications, which could limit the generalizability of the results.

The work of He *et al.* [15] proposes a query execution time estimation in databases using graph neural networks to capture dependency and interaction relationships. They employed graph-based feature modeling and neural networks to enhance the accuracy of query execution time predictions. The three-stage process—workload generation, graph-based feature modeling, and training and estimation—provides a comprehensive framework for accurate time estimation, which is crucial for effective database management and monitoring. The experimental results demonstrating the superior accuracy of this approach compared to existing methods further solidify its contribution. However, the paper may have weaknesses related to the scalability of the proposed method in very large databases, as well as the potential overhead introduced by the graph modeling process.

Whairit *et al.* [16] introduce JINDEX, a JSON and index-based search system for plant germplasm databases, address-



ing the limitations of traditional data warehousing in terms of structural flexibility, scalability, and search performance. They proposed a hybrid key-value/document data model within a NoSQL framework, allowing for flexible and efficient storage and retrieval of plant germplasm data. The results highlight significant improvements in query response time compared to previous relational database implementations, underscoring the system's practical viability since its deployment in 2020. However, their method can have potential challenges in managing and updating the on-disk tree structure over time.

Most of the above works focused on indexing selection algorithms for already indexed columns. The main goal of techniques in the reviewed works is to minimize query access time by selecting index columns at runtime. However, to the best of our knowledge, there is little or no information about controlled empirical results showing the performances between different indexes selected at design time given the existence of these index selection algorithms.

The following section presents the empirical analysis by describing the index structures in the MySQL database. All empirical results are expected to be presented in graphical and graph.

## 5. MySQL index structures

The MySQL relational database system comes with options for several storage engines. Each table in MySQL comes with an option to specify the kind of storage engine that will be used [4]. Some of these storage engines are listed below:

- a) MyISAM: This default MySQL version 5 storage engine is used by most developers in web, data warehousing, and other application environments. MyISAM is supported in all MySQL configurations and is the default storage engine unless you have configured MySQL to use a different one by default [4].
- b) Archive: Provides the solution for storing and retrieving large amounts of seldom-referenced historical, archived, or security audit information [4].
- c) InnoDB: This is a transaction-safe (atomicity, consistency, isolation, and durability (ACID) compliant) storage engine for MySQL. It is built to include commit, rollback, and crash-recovery capabilities for the protection of user data. The InnoDB storage engine provides row-level locking (without escalation to coarser granularity locks). It also supports nonlocking reads with increased multi-user concurrency and performance [4]. The user data in InnoDB is stored in clustered indexes to reduce I/O for common queries based on primary keys. To maintain data integrity, it also supports FOREIGN KEY referential-integrity constraints.

The MySQL 5.0 comes with two types of Indexes namely BTree and Hash. Table 1 shows the various storage engines and the support index.

## 6. Methodology

This study focuses on empirical analysis of BTrees and Hash indexes using real data and a relational data schema of an existing student record database. The name of the student record management system is withheld for privacy concerns. The database containing multiple related tables was designed with several tables each indexed based on a target research question.

Therefore, several SQL queries were executed on these database tables and a performance log was obtained from the execution. A client program written in java was used to test various queries on MySQL and the execution time for these queries were logged.

In subsequent sections, these logs are presented in a graph showing the performance of these queries on different index structures. The relational schemas designed for test in this research were carefully picked such that it contains some tables having a lot of numeric data and some containing characters. The following section describes the logical schema used for this analysis.

## 7. Logical schema

A real-time database of student records and their results in a higher institution was used in this study. In particular, the schemas were scaled because the full schema containing all the tables is not required for the study. Therefore, only tables required for analyses were picked and analyzed. These tables are:

- a) CourseRegistration table: The purpose of this table is to capture student's registration as well as hold the student's continuous assessment scores and exam scores. This table is as described by the SQL description in Table 2. This table contains a composite primary key composed of columns (semester, students\_student\_id, sessions\_session\_id, courses\_course\_id). The default index structure is shown in Table 3.
- b) Students table: The students' table is a biodata table. The students' table is described in Table 4 and its default index structure is defined in Table 5.
- c) Courses table: The courses table contains course records. It is described in Table 6 and its default index structure is in Table 7.
- d) Programs table: The Program table contains list of all the programs. It is described in Table 8.
- e) Session table: The table contains all academic sessions records. A special point to note about this table is the presence of a self-referential column between session\_id, current\_session and next\_session column. The table is described in Table 9 and its index structure is defined in Table 10.

Table 1. Comparison of comparison of a few MySQL storage engines [4].

Feature	MyISAM	InnoDB	Archive
Storage limits	256TB	64TB	None
Transactions	No	Yes	No
Locking granularity	Table	Row	Table
MVCC	No	Yes	No
Geospatial data type support	Yes	Yes	Yes
Geospatial indexing support	Yes	Yes[a]	No
B-tree indexes	Yes	Yes	No
T-tree indexes	No	No	No
Hash indexes	No	No[b]	No
Full-text search indexes	Yes	Yes[c]	No
Clustered indexes	No	Yes	No
Data caches	No	Yes	No
Index caches	Yes	Yes	No
Compressed data	Yes[d]	Yes[e]	Yes
Encrypted data[f]	Yes	Yes	Yes
Cluster database support	No	No	No
Replication support[g]	Yes	Yes	Yes
Foreign key support	No	Yes	No
Backup / point-in-time recovery[h]	Yes	Yes	Yes
Query cache support	Yes	Yes	Yes
Update statistics for the data dictionary	Yes	Yes	Yes

Table 2. Index structure for Courses\_Registration table.

Field	Type	Null
Ca	Double	NO
Exam	Double	NO
approval_status	enum('Lecturer','HOD','Dean','Senate')	YES
Semester	enum('1','2','3','4')	NO
students_student_id	varchar(20)	NO
sessions_session_id	varchar(20)	NO
courses_course_id	varchar(20)	NO

Table 3. Index structure for Courses\_Registration table.

Column_name	Index_type
Semester	BTREE
students_student_id	BTREE
sessions_session_id	BTREE
courses_course_id	BTREE
students_student_id	BTREE
sessions_session_id	BTREE
courses_course_id	BTREE

The complete logical schema is presented in Figure 4. A point to note is that the initial index for all the tables is BTree. For the actual experiments, modifications were done on the index of these tables and performance based on select/access, insertion, and deletion SQL statements logged and graphed. More specifically, the column index was changed to Hash at each execution run, subsequently producing logged data that were eventually analyzed.

## 8. Setup for Structured Query Language (SQL)

The core goal in this research is to compare the performance of BTree, and Hash Index structure against a given database of records. Therefore, varied SQL statements were executed. These statements are categorized based on the goals to be achieved namely;

- Access time: For the analysis of access times on records, SELECT SQL statements were executed across different tables to measure access time. Regarding the physical storage of these indexes, the sizes of the index files—corresponding to a specified number of records per table—were obtained experimentally and recorded (refer to code snippet I).

### Code snippet I: select statement

```
"SELECT * FROM 'experiment_\n
course_registration'\n
r limit "+limit.
```

- Insertion time: To evaluate the empirical insertion time complexity, INSERT SQL statements were executed on

Table 4. Students table.

Field	Type	Null	Key
student_id	varchar(20)	NO	PRI
first_name	varchar(45)	NO	
middle_name	varchar(45)	YES	
last_name	varchar(45)	NO	
current_level	int(11)	NO	
lga_lga_id	varchar(20)	YES	MUL
student_pwd	varchar(255)	YES	
date_of_birth	Date	YES	
phone_no1	varchar(45)	YES	
phone_no2	varchar(45)	YES	
Email	varchar(255)	YES	
home_address	varchar(255)	YES	
contact_address	varchar(255)	YES	
nok_name	varchar(45)	YES	
nok_address	varchar(255)	YES	
nok_phone	varchar(255)	YES	
nok_rel	varchar(255)	YES	
sponsor_name	varchar(45)	YES	
sponsor_address	varchar(255)	YES	
co_activities	varchar(255)	YES	
graduate_status	enum('G','C','S','E','R','N')	YES	
student_program	varchar(20)	NO	MUL
done_registration	tinyint(1)	YES	
admission_session	varchar(45)	NO	MUL
graduation_year	Date	YES	
Sex	enum('M','F')	YES	
marrital_status	enum('S','M')	YES	
email_2	varchar(45)	YES	
Nationality	varchar(45)	YES	
Passport	varchar(45)	YES	
Disability	varchar(45)	YES	
details_disability	varchar(45)	YES	
medical_problem	varchar(255)	YES	
Religion	varchar(255)	YES	
entry_level	int(11)	YES	
last_reg_session	varchar(20)	NO	
last_reg_semester	varchar(20)	NO	
country_of_origin	varchar(45)	YES	
city_of_origin	varchar(45)	YES	
records_updated	tinyint(1)	YES	
sandwich_reg	tinyint(1)	YES	
summer_reg	tinyint(1)	YES	
sandwich_reg_com	tinyint(1)	YES	
summer_reg_compl.	tinyint(1)	YES	
spil_student	tinyint(4)	NO	
student_group	enum('A','B')	NO	

various tables. However, since the "course\_registration" table contains foreign key references to the "students" and "session" tables, with each referenced key indexed by either a Hash table or B-tree, we employed a composite query to assess the empirical insertion time complexity (see code snippet II).

#### Code Snippet II: insert statement

```
insert into experiment_course_registration\
(select * from course_registration r\n"+\
"where not exists (select * from \
experiment_course_registration e where \
r.semester=e.semester\n" + " and \
```

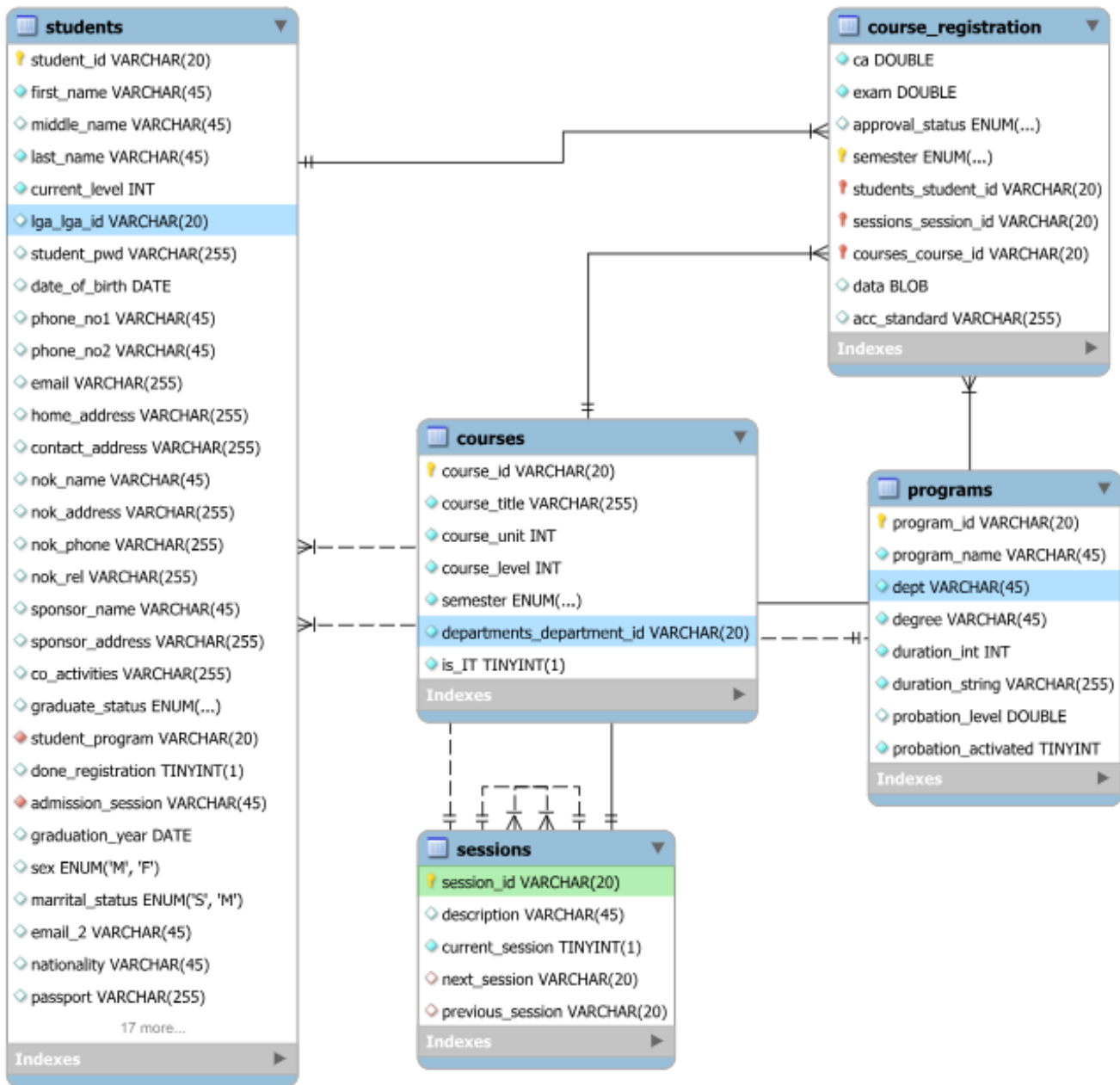


Figure 4. Logical schema design for experimental DB.

Table 5. Default index students.

Key_name	Seq_in_index	Column_name	Index_type
PRIMARY	1	student_id	BTREE
fk_students_lg	1	lga_lga_id	BTREE
FK_students_program	1	_program	BTREE
FK_students_admission_year	1	admission_session	BTREE

Table 6. Courses.

Field	Type	Null
course_id	varchar(20)	NO
course_title	varchar(255)	NO
course_unit	int(11)	NO
course_level	int(20)	NO
Semester	enum('1','2','3','4')	NO
departments_department_id	varchar(20)	NO
is_IT	tinyint(1)	NO



Table 7. Index structure for courses table.

Seq_in_index	Column_name	Index_type
1	Semester	BTREE
2	students_student_id	BTREE
3	sessions_session_id	BTREE
4	courses_course_id	BTREE
1	students_student_id	BTREE
1	sessions_session_id	BTREE
1	courses_course_id	BTREE

Table 8. Programs table.

Field	Type	Null	Key
program_id	varchar(20)	NO	PRI
program_name	varchar(45)	NO	
Dept	varchar(45)	NO	MUL
Degree	varchar(45)	NO	
duration_int	int(11)	NO	
duration_string	varchar(255)	NO	

Table 9. Session table.

Field	Type	Null	Key
session_id	varchar(20)	NO	PRI
Description	varchar(45)	YES	
current_session	tinyint(1)	NO	
next_session	varchar(20)	YES	MUL
previous_session	varchar(20)	YES	MUL

Table 10. Index structure for session table.

Key_name	Seq_in_index	Column_name	Index_type
PRIMARY	1	session_id	BTREE
FK_sessions_next_Session	1	next_session	BTREE
FK_sessions_previous_session	1	previous_session	BTREE

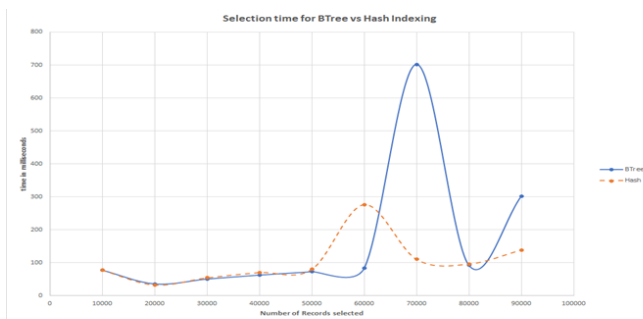


Figure 5. Selection time for BTree and Hash indexes.

```
r.students_student_id = \
e.students_student_id and\n"+ " \
r.sessions_session_id = \
e.sessions_session_id and \
r.courses_course_id = \
e.courses_course_id) limit 10000);
```

- c) Deletion time To evaluate the empirical deletion time, a DELETE statement was constructed and executed on the "course\_registration" table, as demonstrated in code snippet III.

### Code snippet III: delete statement

```
delete FROM 'experiment_db'.\
'course_registration'\
r limit "+limit;
```

## 9. Experiment setup

In this section, the details of the actual experiments. The client program that generates execution time log was written in Java and a simple algorithm was used to obtain the execution time for each SQL statement *s* seen in code snippet IV.

### Code snippet IV: query execution

```
public long executeQuery(String sql)
{
    try {
        //Connect to the database
        connect ();
        //Prepare the sql statement object
        stmt = con.createStatement();
        // time b4 query execution
        long beforeTime = System.nanoTime();
        //Execute query
        stmt.executeQuery(sql);
        // time after query execution
        long timeAfter=System.nanoTime();
        //Find the different in nanoseconds
        long diff=timeAfter-beforeTime;
        //close connection
        closeConnection();
        //return the time difference
        return diff;
    } catch (SQLException ex) {
        closeConnection();
        return -1; }
}
```

- a) System configuration: The experiment was executed on Dv6 HP Envy machine. The system configuration can be seen in Table 11.
- b) Test client tools and database configuration: This experiment was performed on MySQL Server version 5.5.40 running on localhost port number 3306. A program written in Java was used to obtain the execution time for each

Table 11. Hardware configuration.

OS Name	Microsoft Windows 8
OS Version	6.2.9200 N/A Build 9200
OS Manufacturer	Microsoft Corporation
OS Configuration	Standalone Workstation
OS Build Type	Multiprocessor Free
Registered Organization	Hewlett-Packard
System Manufacturer	Hewlett-Packard
System Model	HP ENVY dv6 Notebook PC
System Type	x64-based PC
Processor(s)	1 Processor(s) Installed.
[01]	Intel64 Family 6 Model 58 Stepping 9 GenuineIntel ~2401 Mhz
BIOS Version	Insyde F.28, 25/07/2013
System Locale	en-gb;English (United Kingdom)
Input Locale	en-gb;English (United Kingdom)
Total Physical Memory	16,273 MB
Available Physical Memory	11,403 MB
Virtual Memory Max Size	32,657 MB
Virtual Memory Available	21,737 MB
Virtual Memory In Use	10,920 MB

of the queries that was executed. The Java program runs on Java virtual machine version 1.7.

- c) Experiment layout: The experiment is categorized into two parts; analysis in terms of speed of execution of SQL queries and in terms of storage capacity.

Among the six tables selected for the study, the course\_registration table had the highest number of live activities and, consequently, the largest volume of records. By design, the number of records in this table grows arithmetically with each passing semester. At the time of the study, it contained approximately 149,377 records without any secondary index structure. However, after applying indexing to certain columns (semester, students\_student\_id, sessions\_session\_id, and courses\_course\_id) using the Hash index, the table could no longer accommodate the full 149,377 records. As a result, the total number of records had to be scaled down to 90,171.

Conversely, applying the B-tree index to the same columns (semester, student\_student\_id, sessions\_session\_id, and courses\_course\_id) did not impose any limitation on the number of records the table could accommodate.

For the speed analysis, three distinct experiments were conducted: access time (using the SELECT statement), insertion time (using the INSERT statement), and deletion time (using the DELETE statement). The SQL statements employed in these experiments are provided in Code Snippets I, II, and III, respectively.

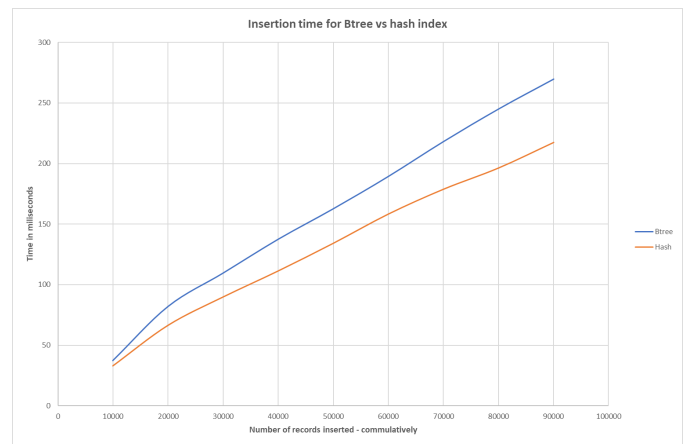


Figure 6. Insertion time for BTree and Hash indexes.

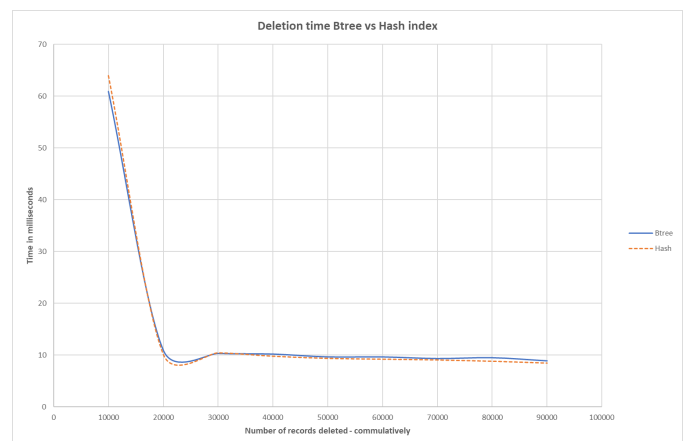


Figure 7. Insertion time for BTree and Hash indexes.

## 10. Result

- a) Analysis: query selection time: Given the code snippet I, where for each *limit* parameter, a total of 5 runs were conducted and the average obtained. Output for each Index structure is presented in Figure 5. As expected, it was observed that as the number of records selected increased, so did the time required to return the records. This trend is evident in both plots. However, superior performance was noted in the case of the hash index. This aligns with the theoretical guarantees of the hash table data structure, which offers an average search complexity of  $O(1)$  plus the cost of handling overflow pages, compared to the B-tree's search complexity of  $O(\log n)$ .

Additionally, in both plots in Figure 5, peaks were observed at 6,000 and 7,000 records for the Hash and B-tree indexes, respectively. This behavior is attributed to the query planning phase of the query optimization algorithm during runtime. Since the performance of a query optimization algorithm is influenced by how efficiently lookups are performed on either index, it was observed that the query planning stage is significantly faster in the case of the Hash index.

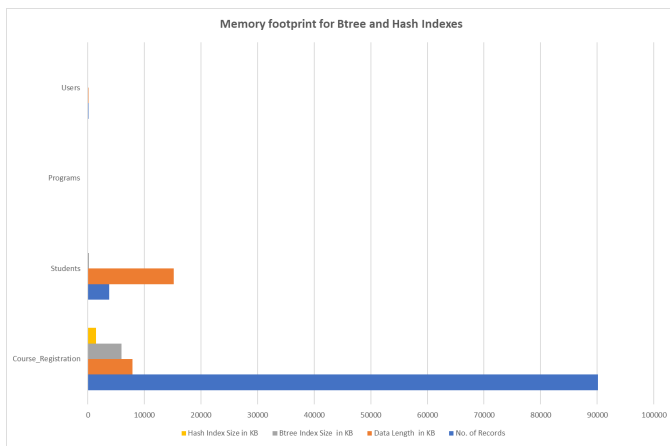


Figure 8. Memory footprint - BTree and Hash indexes.

- b) Analysis: insertion time: A composite SQL statement was provided to insert batches of records from another relational database into the experimental datasets. As shown in the SQL statement in Code Snippet II, the query fully utilizes the relationships defined within the experimental database schema.

For each execution run, 10,000 records were fetched, with each run repeated five times, and the average performance was computed. Figure 6 presents the performance plots. As seen in Figure 6, both plots exhibit a linear relationship between the number of records processed and execution time. However, the hash index demonstrates slightly better performance compared to the B-tree index. This observation is consistent with theoretical expectations, where insertion in a hash table operates in  $O(1)$  time, while insertion in a B-tree operates in  $O(\log n)$ .

- c) Analysis: deletion time: In the deletion time analysis, SQL statements are based on code snippet III. The limit variable specifies the number of records deleted in each iteration, which is set at 10,000 per batch. Similar to previous experiments, each run was executed five times, with the average deletion time recorded.

In the experiment, deletion times for both Hash and Btree indexes were relatively similar, although the Hash index was slightly faster. This result aligns with theoretical expectations, as deletion in a Hash index operates in  $O(1)$  time, while in a Btree index, it operates in  $O(\log n)$  time. The deletion results are illustrated in Figure 7.

- d) Comparative analysis based on memory size. Figure 8 shows the memory footprint distribution for each table and their respective index file sizes. Among all the index files, that of the hash table is the largest. Essentially, it points to the fact that as the number of records increases so is the size of the index size. Consequently, the hash index file will quickly become too large as the number of records increases.

## 11. Conclusion

This paper presents an empirical comparison of BTree and Hash indexes using the MySQL storage engine. A sample relational database, sourced from an online student learning management system, was used for the analysis. The comparison focuses on access time (lookup), insertion time, deletion time, and memory footprint for columns indexed with BTree and Hash structures.

In the study, a client-side program written in java was used to send sql statements to the relational database, while the time of execution is recorded and stored in a text file. The data from the logs form the input to the plots that showed the performance of these index structures.

It was observed that regardless of the query selection algorithm used, deciding on what type of index to use in a relational dataset design is pivotal. This study showed that it is relatively faster to perform selection, insertion and deletion on a table indexed using hash index, compared to the same table with Btree index and these results corresponds to theoretical time complexity for both hash and Btree indexes respectively.

In terms of memory consumption, the hash index has much more memory footprint than BTree index as the size of record increases. Therefore, a limitation of hash indexes is in the size of the index file. This obviously limits its usage for large tables.

In general, there is a tradeoff between achieving high speed performance and high memory footprint so the nature of the application should determine the choice of index. For system requiring high volume of selection and insertion, the Btree index should be desirable while for a system with not so large volume of data, the Hash index is most suited.

## References

- [1] A. Silberschatz, H. F. Korth & S. Sudarshan, "Indexing and Hashing", in *Database system concepts*, 5th Edition ed., Singapore, McGraw-Hill, 2006, pp. 1–134. <https://www.db-book.com/Previous-editions/db5/slide-dir/index.html>.
- [2] S. Choenni, H. Blanken & T. Chang, "Index selection in relational databases", in Proceedings of ICCI'93: 5th International Conference on Computing and Information, Sudbury, ON, Canada, 1993, pp. 491-496. <https://doi.org/10.1109/ICCI.1993.315323>.
- [3] A. Arteta, N. G. Blas & L. F. López, "Intelligent indexing—boosting performance in database applications by recognizing index patterns", *Electronics* **9** (2020) 2079. <https://doi.org/10.3390/electronics9091348>.
- [4] MySQL 2015. [Online]. <https://dev.mysql.com/doc/refman/5.0/en/history.html>. [Accessed 21 May 2015].
- [5] T. Lahdenmäki & M. Leach, "SQL Processing", in *Relational Database Index Design and the Optimizers*, T. Lahdenmäki and M. Leach (Eds.), John Wiley & Sons, Inc., Hoboken, New Jersey, U.S., 2005, pp. 29–45. <https://doi.org/10.1002/0471721379>.
- [6] R. Ramakrishnan & J. Gehrke, "Database Management Systems", 3rd Edition ed., Singapore McGraw-Hill, 2003. <https://raw.githubusercontent.com/pforpallav/school/master/CPSC404/Ramakrishnan%20-%20Database%20Management%20Systems%203rd%20Edition.pdf>
- [7] A. Gani, A. Siddiq, S. Shamshirband & F. Hanum, "A survey on indexing techniques for big data: taxonomy and performance evaluation", *Knowledge and Information Systems* **46** (2016) 0219. <https://doi.org/10.1007/s10115-015-0830-y>.
- [8] R. Ramakrishnan & J. Gehrke, *Database Management Systems*, McGraw-Hill Inc., Ithaca, New York USA, 2002, pp. 25–29. <https://raw.githubusercontent.com/>

- [9] S. Chaudhuri, M. Datar & Narasayya, “V.Index selection for databases: A hardness study and a principled heuristic solution”, *IEEE transactions on knowledge and data engineering* **16** (2004) 1313. <https://doi.org/10.1109/TKDE.2004.75>.
- [10] R. Schlosser, J.Kossmann & M. Boissier, “Efficient scalable multi-attribute index selection using recursive strategies”, in 2019 IEEE 35th International Conference on Data Engineering (ICDE), Macao, China, 2019, pp. 1238–1249. <https://doi.org/10.1109/ICDE.2019.00113>.
- [11] R. Yadav, S. Valluri & M. Zait, “AIM: A practical approach to automated index management for SQL databases”, in 2023 IEEE 39th International Conference on Data Engineering (ICDE), Anaheim, California, USA, 2023, pp. 3349–3362. <https://doi.org/10.1109/ICDE55515.2023.00257>.
- [12] P. K. Sadineni, “Comparative study on query processing and indexing techniques in big data”, in 2020 3rd International Conference on Intelligent Sustainable Systems (ICISS), Coimbatore, India, 2020, PP. 933–939. <https://doi.org/10.1109/ICISS49785.2020.9315935>.
- [13] M. Desai, R. G. Mehta & D. P. Rana, “A Survey on Techniques for Indexing and Hashing in Big Data”, in 2018 4th International Conference on Computing Communication and Automation (ICCCA), Noida, India, 2018, pp. 1–6. <https://doi.org/10.1109/CCAA.2018.8777454>.
- [14] Y. Shin, J. Ahn & D. H. Im, “Join optimization for inverted index technique on relational database management systems”, *Expert Systems with Applications* **198** (2022) 116956. <https://doi.org/10.1016/j.eswa.2022.116956>.
- [15] Z. He, J. Yu, T. Gu & D. Yang, “Query execution time estimation in graph databases based on graph neural networks”, *Journal of King Saud University-Computer and Information Sciences* **36** (2024) 102018. <https://doi.org/10.1016/j.jksuci.2024.102018>.
- [16] T. Whairit, B. Phadermrod & V. Attasena, “JINDEX: JSON and index search system for plant germplasm database”, *Journal of King Saud University-Computer and Information Sciences* **35** (2023) 101701. <https://doi.org/10.1016/j.jksuci.2023.101701>.