Analysis of MySQL DBMS with InnoDB Engine

Database Systems
FCT NOVA

22nd of may, 2020

Contents

1 Introduction

For this project, we chose to study MySQL 8.0 based on its popularity and due to the fact that we are using it as our main database in other Computer Science courses.

As we will explain in detail in Section 2.1, MySQL supports several storage engines. However, for the purpose of this project, we will focus our analysis on the InnoDB storage engine, as it currently is the default for MySQL.

1.1 Report Structure

What follows is a very short summary of the contents discussed in each section of this report:

In Section 2, we talk about the storage and file structure used in MySQL. We start by specifying the various storage engines, and then go into detail only with InnoDB. We discuss MySQL's file system, tuple organization, buffer management and partitioning.

In Section 3, we discuss how MySQL implements indexing and hashing. We talk about the index structures it implements and focus our explanation mainly on clustered and secondary indexes.

In Section 4, we talk about the whole pipeline of how MySQL processes, optimizes and executes SQL queries. We focus mainly on the optimization aspect, by describing the several ways in which the query optimizer can optimize queries, namely optimizing through indexes and optimizing simple and complex operations. We also explain how the Query optimizer can be controlled, how MySQL estimates table and index statistics and how the user can see information about the execution plan.

In Section 5, we talk about MySQL transactions and how they adhere to the ACID model, explaining how each property is implemented in detail. We also describe how MySQL controls concurrency by the using locks.

Seen as the InnoDB storage engine does not offer support for distributed databases, that subject is not covered in this report.

Additionally, in Section 6, we describe some interesting characteristics of MySQL. We talk about MySQL's support for XML and XPath, and its implementation of stored routines and triggers. We also discuss how to use JDBC to connect to a MySQL database, and present a very useful graphical tool, MySQL Workbench.

Please note that all sections, except Section 6, end with a comparison of MySQL to Oracle regarding the subjects studied in that section.

1.2 History of MySQL

MySQL is the world's most popular open source Database Management System (DBMS) and it is currently developed, distributed and supported by Oracle Corporation [80]. It has great performance, reliability, ease-of-use and has become the leading database choice for web-based applications. It is used by high profile companies such as Facebook, Twitter and YouTube.

MySQL was born in Sweden, in the 90's, at a company founded by David Axmark, Allan Larsson and Michael Widenius. The creators felt the need to connect their tables to a database, mSQL, using Indexed Sequential Access Method (ISAM) routines. However, mSQL was not fast nor flexible enough for their needs, so they created a new SQL interface.

MySQL was an open source project from the beginning but, when its popularity and number of users increased within the database community, it was bought by Sun Microsystems (2008). Later, this company was acquired by Oracle Corporation (2009).

Nowadays, MySQL still belongs to Oracle Corporation and it comes in several editions [49]. Some of these editions are free of charge and open source, and others are paid and come with additional functionalities.

2 Storage and File Structure

As we mentioned in Section 1, this project will focus on the InnoDB storage engine. However, in Section 2.1, we will give some details about the various storage engines supported. Then, we will discuss MySQL's file system, tuple organization and, finally, we will detail how MySQL manages caching by using the buffer pool, and the various ways of partitioning it allows.

2.1 Storage Engines

MySQL is known for supporting multiple storage engines [9]. These storage engines are software components that a DBMS uses to handle SQL operations like create, read, update and delete (CRUD) for different table types. In other words, the DBMS uses storage engines as a way to manage the data in the underlying memory and storage systems [1], providing an abstraction layer that prevents contact between the users and the native file system.

Below, we present the storage engines supported by MySQL 8.0:

- **InnoDB:** This engine is the default in MySQL 5.5 and later. It is ACID compliant, having commit, rollback and crash-recovery capabilities. It supports row-level locking and multi-version concurrency control and it is the only engine that provides foreign key referential-integrity constraints;
- **MyISAM:** This engine uses table-level locking and, as such, it has very limited performance when it comes to read/write workloads, so it is mostly used in read-only or read-mostly workloads;
- **Memory:** This engine, formerly known as HEAP engine, stores all data in RAM to improve access speeds. As of late, it is not being used frequently because other engines have evolved in ways that match the access speed requirements;
- **CSV:** This engine has tables (text files with comma-separated values) that let you import or dump data in CSV format;
- **Archive:** This engine has compact unindexed tables used for storing and retrieving large amounts of rarely used information;
- **Blackhole:** This engine accepts but does not store data so, queries always return an empty set. It is often used in distributed environments for replicating data with the master-slave model;
- **NDB aka NDBCLUSTER:** This clustered database engine is mostly used in distributed environments that have high uptime and availability needs;

- **Merge:** This engines allows the grouping of identical MyISAM tables;
- **Federated:** This engine is commonly used in distributed environments, because it allows the creation of a logical database by linking several physical MySQL servers;
- **Example:** This engine, present in the source code for MySQL, is an example that shows how to begin writing new storage engines.

The choice of storage engine greatly impacts the performance of the application, so choosing the right one is very important. The users should take into account the requirements of the application and the characteristics of each storage engine, in order to choose the one that better suits their needs. The code below shows how to set the default storage engine for a session [66]:

```
SET default_storage_engine = INNODB;
```

It is important to note that the user is not restricted to using the same storage engine for an entire server or schema. The users can specify the storage engine for any table like so:

```
CREATE TABLE t1 (i INT) ENGINE = CSV;
```

When the `ENGINE` option is omitted, the default storage engine is used. If the user did not set the default engine, InnoDB will be used. It is also possible to convert a table from one storage engine to another:

```
ALTER TABLE t1 ENGINE = INNODB;
```

2.2 File System

InnoDB implements a storage system that has physical and logical characteristics to enhance performance, scalability and backup.

The System tablespace is the storage area for the change buffer. This buffer caches changes to secondary index pages when those pages are not in the buffer pool [11].

By default, InnoDB stores both table and index data in a File-Per-Table tablespace, by creating a .ibd file per table [29]. If the user does not wish to store table and index data in a File-Per-Table tablespace, InnoDB allows him to choose other methods of storage, such as [23]:

- **Storage in the System tablespace:** Keeping all InnoDB table and index data in one file, often causing it to become very large. The System tablespace never shrinks and, because of this, storage problems can arise if large amounts of temporary data are loaded and then deleted;
- **Storage in a General tablespace:** Keeping all InnoDB table and index data in a shared tablespace that supports all row formats and can be created like so:

```
CREATE TABLESPACE ts ADD DATAFILE 'ts.ibd' ENGINE = INNODB;
```

As can be seen in Figure 1, each InnoDB tablespace stores rows inside pages. These pages have a default size of 16KB, but you can reduce it down to 4 or 8KB, or increase it to 32 or 64KB. The pages are then grouped into extents of size 1MB per 16KB in page size (e.g. If pages have 32KB, the extent size is 2MB) [30].

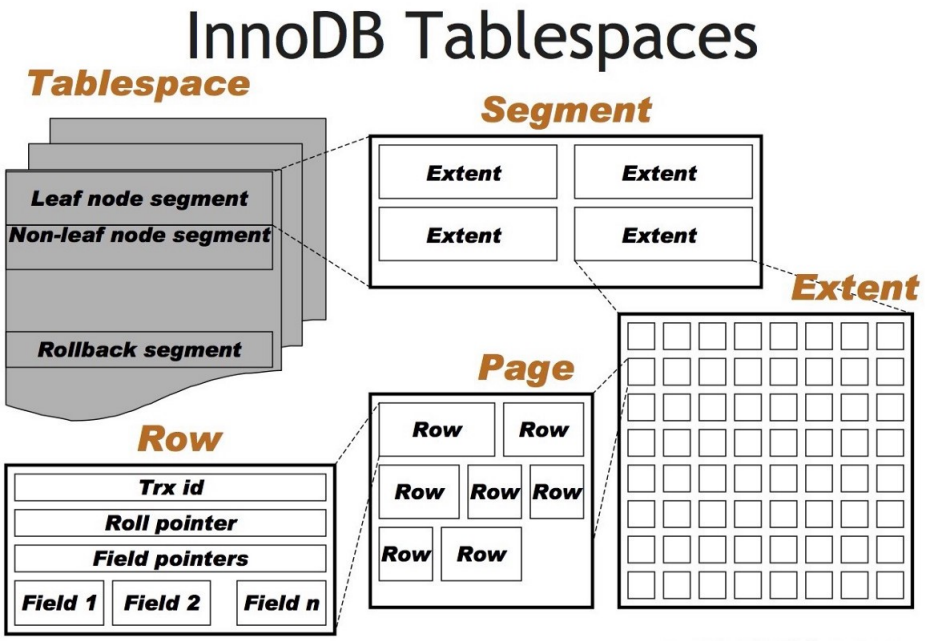

Figure 1: Visual representation of InnoDB tablespaces. Adapted from [4].

In turn, the extents are grouped into "files" inside a tablespace called segments. When these segments grow, InnoDB allocates the first 32 pages to it one at a time. After that, it starts allocating whole extents to the segment. This can go up to a maximum of 4 extents at a time.

Each index is allocated two segments:

1. A segment for non-leaf nodes of the B+-tree;
2. A segment for the leaf nodes of the B+-tree.

If the leaf nodes are kept in contiguous memory on disk, the sequential I/O operations will be better because they contain the actual table data.

When you delete data from a table, InnoDB shrinks the corresponding B+-tree indexes. The availability of the freed space depends on the pattern of deletes freeing individual pages or extents. On the other hand, when you delete all rows from a table or drop it, the space is guaranteed to be freed.

It is important to note that, deleted rows are physically removed by the purge operation when they are no longer needed for transactions, rollbacks or consistent reads.

2.3 Buffer Management

The InnoDB engine reserves space in main memory for the buffer pool, a memory area where table and index data is cached when it is accessed. The buffer pool significantly speeds up the processing of data that is frequently used, so knowing how to manage it is extremely relevant for MySQL Database Tuning [10].

The buffer pool is implemented as a linked-list of pages and, in order to manage the cache efficiently, a variation of the Least Recently Used (LRU) algorithm is used. As such, pages that are not frequently used tend to be removed from the buffer pool.

2.3.1 Buffer Pool LRU Algorithm

As we mentioned before, the buffer pool is managed by a variation of the LRU algorithm. When the buffer is full and someone accesses a page that is not in memory, that page needs to be inserted.

The defining aspect of this variation is the midpoint insertion strategy - when inserting a new page, it is inserted in the middle of the list. This strategy divides the buffer pool in two sublists:

1. At the head of the buffer pool is a sublist of new pages that were accessed recently;

2. At the tail of the buffer pool is a sublist of old pages that were accessed less recently. These pages are the candidates for eviction.

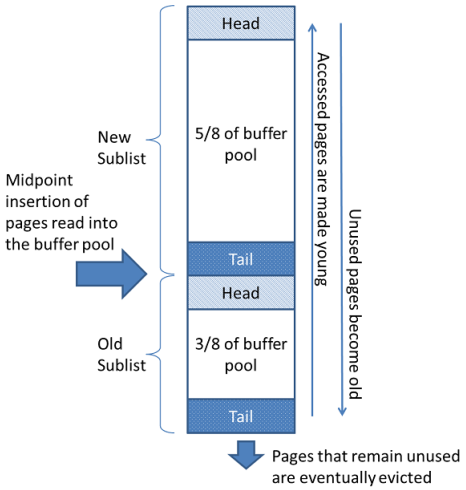

Figure 2: Visual representation of the Buffer Pool List and its division. Taken from MySQL Documentation page [10].

The LRU algorithm reserves 3/8 of the buffer pool for the old sublist. When a page is read into the buffer, it is inserted at the midpoint. As can be

seen in Figure 2, this midpoint is not in the middle of the list. It is located on the boundary between the new page sublist and the old page sublist. When a page in the old sublist is accessed, it is moved to the head of the new sublist.

As time passes, the pages that are not accessed keep moving toward the tail of the list. Pages in the new sublist "age" when other pages are made new. Pages in the old sublist "age" when other pages are made new and when pages are inserted at the midpoint. If a page is not used for a certain period of time, it tends to reach the tail of the old sublist, being evicted when a new page is inserted.

It is important to note that a page can be read because of a user-initiated operation or due to a read-ahead operation.

2.3.2 Buffer Pool Configuration

The buffer pool can be configured in order to improve performance. There are several parameters that can be tuned, such as:

- **Size of the buffer pool:** The size of the buffer pool should be set to as large a value as possible while still leaving enough memory for other processes. Obviously, if the buffer pool is very big, InnoBD reads data from disk only once and then is able to access it from memory. The size of the buffer pool can specified using the variable `innodb_buffer_pool_size`;

- **Buffer pool instances:** If the `innodb_buffer_pool_size` is set to a value larger than 1GB, it is possible to divide the buffer pool into various instances. This can significantly improve performance when concurrent threads are trying to read or write the cached pages. The number of buffer pool instances is set with `innodb_buffer_pool_instances` and can vary between 1 and 64;

- **Resistance to scan:** When a table scan is performed, a large amount of data is brought into the buffer pool and a lot of older data is evicted, even if the new data is never used again. This can cause serious issues, so you can keep frequently accessed data in memory regardless of sudden spikes of activity. By setting `innodb_old_blocks_time`, the time window (in milliseconds) after the first access to a page during which it can be accessed without being moved to the front of the LRU list, to a high value, our buffer will become table scan resistant;

- **Buffer pool prefetching:** Read-ahead requests prefetch pages into the buffer pool in anticipation that the pages will be needed soon. The `innodb_read_ahead_threshold` and `innodb_random_read_ahead` variables can control, respectively, when linear read-aheads and random read-aheads are performed;

- **Buffer pool flushing:** InnoDB performs the flushing of dirty pages from the buffer pool in the background. There are several parameters that can be controlled, such as the number of cleaner threads,

`innodb_page_cleaners`, and the low water mark for the percentage of dirty pages, `innodb_max_dirty_pages_pct_lwm`;

- **Buffer pool state:** To avoid a large warm up period after restarting the server, a percentage of the most recently used pages for each buffer pool is saved at shutdown and restored at startup. This percentage can be set with the `innodb_buffer_pool_dump_pct` variable.

2.4 Tuple Organization

In MySQL, the table data is divided into several pages. These pages are organized in a tree data structure called a B+-tree. This structure is used for storing both table data and secondary indexes [41].

The B+-tree that represents a table is called the clustered index and is organized by the corresponding primary key columns. The nodes of the tree structure contain the values of all columns in the row. If the B+-tree is a secondary index, its nodes only contain the values of index columns and primary key columns.

If columns are of variable-length, also known as off-page columns, and are too long to fit on a B+-tree page, they are stored on separately allocated disk pages called overflow pages. Each column has a singly-linked list of one or more overflow pages. However, depending on column length, some column values may be stored in the B+-tree to avoid wasting storage and having to read a separate page. MySQL variable length types are `VARCHAR`, `VARBINARY`, `BLOB` and `TEXT`.

InnoDB supports four row formats and we will talk about them in the following sections.

2.4.1 REDUNDANT Row Format

- Provides compatibility with older versions of MySQL;
- Tables store the first 768 bytes of variable-length column values in the index record within the B+-tree node. The rest is stored on overflow pages;
- Fixed-length columns greater than or equal to 768 bytes are encoded as variable-length columns and are also stored off-page;
- For columns with values of 768 bytes or less, an overflow page is not used.

2.4.2 COMPACT Row Format

- Reduces row storage space by about 20% when compared to the REDUNDANT row format. However, it increases CPU usage for some operations;

- Tables store variable-length and fixed-length column values in the same way as the REDUNDANT row format.

2.4.3 DYNAMIC Row Format

- Offers the same benefits as the COMPACT row format but enhances storage for long variable-length columns and supports large index key prefixes;
- Stores long variable-length column values completely off-page. The clustered index record contains only a 20-byte pointer to the overflow page;
- Fixed-length fields greater than or equal to 768 bytes are encoded as variable-length fields;
- The off-page storage depends on the page size and the total size of the row. The longest columns are chosen for off-page storage until the clustered index record fits on the B+-tree page;
- Tables can be stored in the System tablespace, File-per-Table tablespaces and General tablespaces.

2.4.4 COMPRESSED Row Format

- Offers the same benefits as the DYNAMIC row format but supports table and index data compression;
- The `KEY_BLOCK_SIZE` variable controls how much column data is stored in the clustered index, and how much is placed off-page;
- Tables can be stored in File-per-Table tablespaces and General tablespaces.

2.5 Partitioning

Not all storage engines support the partitioning of tables. However, InnoDB is one of the engines that does [61].

Partitioning allows the users to distribute the tables in portions across a file system, storing them as separate tables in different locations. This is done according to a rule chosen by the user, the partitioning function. This function relies on the partitioning type, which is also specified by the user.

Before delving into the partitioning types, it is important to note that:

- The user cannot disable partitioning support in InnoDB;
- The same storage engine must be used for all partitions of the same partitioned table [60];
- The user must partition both table and indexes. It is also not possible to partition only a portion of a table.

There are two kinds of partitioning: the **horizontal partitioning**, where different rows of a table are assigned to different physical partitions, and **vertical partitioning**, where different columns of a table are assigned to different physical partitions. With this said, note that MySQL 8.0 does not support vertical partitioning.

Let us now look at the different horizontal partitioning types available in MySQL 8.0 [62].

2.5.1 RANGE Partitioning

When a table is partitioned by `RANGE`, each partition contains rows for which the partitioning expression value lies within a given range. This is defined using the `VALUES LESS THAN` operator. To avoid getting an error when adding a row whose `store_id` value is greater than the highest value defined, we use the `MAXVALUE` operator [63]:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

2.5.2 LIST Partitioning

This type of partitioning is similar to partitioning by `RANGE` in the way that each partition must be explicitly defined. The big difference lies in the fact that, in `LIST` partitioning, each partition is defined and selected based on if a column value belongs to a list of values and not to a contiguous range of values. To specify the list that will define each partition we use the `VALUES IN` operator [44]:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
```

```
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY LIST(store_id) (
    PARTITION pNorth VALUES IN (3,5,6,9,17),
    PARTITION pEast VALUES IN (1,2,10,11,19,20),
    PARTITION pWest VALUES IN (4,12,13,14,18),
    PARTITION pCentral VALUES IN (7,8,15,16)
);
```

2.5.3 COLUMNS Partitioning

This partitioning is a variant on `RANGE` and `LIST` partitioning. `COLUMNS` partitioning allows the use of multiple columns in partitioning keys, so when dividing the tables in partitions and putting rows into those partitions, all of these column values will be taken into account. Additionally, both `RANGE COLUMNS` partitioning and `LIST COLUMNS` partitioning support the use of non-integer columns for defining value ranges or list members [13]:

RANGE COLUMNS partitioning:

```
CREATE TABLE rcx (
    a INT,
    b INT,
    c CHAR(3),
    d INT
)
PARTITION BY RANGE COLUMNS(a,d,c) (
  PARTITION p0 VALUES LESS THAN (5,10,'ggg'),
  PARTITION p1 VALUES LESS THAN (10,20,'mmm'),
  PARTITION p2 VALUES LESS THAN (15,30,'sss'),
  PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE,MAXVALUE)
);
```

LIST COLUMNS partitioning:

```
CREATE TABLE customers_1 (
    first_name VARCHAR(25),
    last_name VARCHAR(25),
    street_1 VARCHAR(30),
    street_2 VARCHAR(30),
    city VARCHAR(15),
    renewal DATE
)
PARTITION BY LIST COLUMNS(city) (
```

```
    PARTITION p1 VALUES IN('Oskarshamn','Högsby','Mönsterås'),
    PARTITION p2 VALUES IN('Vimmerby','Hultsfred','Västervik'),
    PARTITION p3 VALUES IN('Nässjö','Eksjö','Vetlanda'),
    PARTITION p4 VALUES IN('Uppvidinge','Alvesta','Växjo')
);
```

2.5.4 HASH Partitioning

We use `HASH` partitioning to ensure an even distribution of data among a predetermined number of partitions. This differs from `RANGE` or `LIST` partitioning because with those types of partitioning, you had to specify which partition a given column value or set of column values would be stored in. On the other hand, with `HASH` partitioning there is no need to specify that. All you need to do is specify the column value or expression based on a column value to be hashed, and the number of partitions you want your table to be divided in [35]:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY HASH(store_id)
PARTITIONS 4;
```

2.5.5 KEY Partitioning

This type of partitioning is similar to partitioning by `HASH`, except that where the hashing for `HASH` partitioning uses an expression defined by the user, the hashing function for `KEY` partitioning is supplied by the MySQL server [43]. `KEY` takes a list of zero or more column names and does the following:

- If `KEY` has zero column names, then the table's primary key is used, if there is one;
- If `KEY` has any column names, then they must comprise part or all of the table's primary key, if the table has one.

```
CREATE TABLE k1 (
    id INT NOT NULL PRIMARY KEY,
    name VARCHAR(20)
)
PARTITION BY KEY()
PARTITIONS 2;
```

Unlike some partitioning types, in KEY partitioning the columns used can have values besides integer or NULL, since the hashing function is supplied by MySQL and guarantees an integer result regardless of the column data type.

```
CREATE TABLE tm1 (
    s1 CHAR(32) PRIMARY KEY
)
PARTITION BY KEY(s1)
PARTITIONS 10;
```

2.5.6 Subpartitioning

Subpartitioning, also known as composite partitioning, consists of subdividing each partition in a partitioned table [70]. This partitioning can be done with RANGE or LIST partitioning. The subpartitions may use either HASH or KEY partitioning.

The following table has three partitions (p0, p1, p2), each one divided in two subpartitions:

```
CREATE TABLE ts (
    id INT,
    purchased DATE
)
PARTITION BY RANGE( YEAR(purchased) )
SUBPARTITION BY HASH( TO_DAYS(purchased) )
SUBPARTITIONS 2 (
    PARTITION p0 VALUES LESS THAN (1990),
    PARTITION p1 VALUES LESS THAN (2000),
    PARTITION p2 VALUES LESS THAN MAXVALUE
);
```

The subpartitions can also be explicitly defined using SUBPARTITION. It is important to note that:

- Each partition must have the same number of subpartitions;
- If you explicitly define any subpartitions using SUBPARTITION, you must define them all.

```
CREATE TABLE ts (
id INT,
purchased DATE
)
PARTITION BY RANGE( YEAR(purchased) )
    SUBPARTITION BY HASH( TO_DAYS(purchased) ) (
        PARTITION p0 VALUES LESS THAN (1990) (
            SUBPARTITION s0,
```

```
        SUBPARTITION s1
    ),
    PARTITION p1 VALUES LESS THAN (2000) (
        SUBPARTITION s2,
        SUBPARTITION s3
    ),
    PARTITION p2 VALUES LESS THAN MAXVALUE (
        SUBPARTITION s4,
        SUBPARTITION s5
    )
);
```

2.6 Comparison with Oracle 18c

In Oracle, the standard table is organized in a heap structure. However, it is possible to organize the table data by using index structures, similarly to MySQL with the InnoDB engine. It is worth mentioning that, unlike in MySQL with InnoDB, in Oracle multitable clustering is possible.

Along the lines of MySQL, in Oracle the data is organized in tablespaces. Each tablespace is divided into segments and each segment is divided into extents - sets of contiguous data blocks.

Oracle has its own buffer and buffer manager and, just like with InnoDB, it uses the LRU algorithm to manage the cached pages.

Much like MySQL, Oracle allows the partitioning of tables. However, some of the partitioning types are different. MySQL and Oracle both support: Range, List, Composite and Hash partitioning. Oracle does not support Key partitioning but it allows the following extra types [86]: Auto-List Partitioning, Interval Partitioning, Reference Partitioning Partitions, Virtual Column Based Partitioning and Interval Reference Partitioning.

3 Indexing and Hashing

In MySQL there are few sophisticated search algorithms. As such, the correct use of index files makes a huge difference on the performance of query executions. In this section, we will talk about what structures MySQL uses to stores indexes, and which types of indexes it supports.

3.1 InnoDB Index Structures

As we mentioned in Section 2.4, MySQL uses B+-trees not only for storing indexes, but also for storing tuples. The exception to the use of this structure are spatial indexes, which are stored in R-trees. R-trees are data structures used for indexing multi dimensional data.

Independently of the structure used to store the index, index records are stored in the leaf pages of trees.

To improve spatial organization, InnoDB performs a sorted index build when creating or rebuilding an index structure, with the exception of spatial indexes [67]. Additionally, if the occupation of an index page drops below the `MERGE_THRESHOLD`, which is 50% by default, InnoDB tries to shrink the tree to free the page [72].

3.2 Clustered Indexes

For every table, InnoDB creates a special index called clustered index. A clustered index is where the data for rows is stored. So, when trying to access a row, the data will be found very quickly because the index in question stores the row data in the leaves of its B+-tree [12].

The clustered index is associated with the primary key of a table, as to achieve the best performance several in operations, such as queries and inserts. The most important things to note when talking about clustered indexes are:

- When a table has a `PRIMARY KEY`, InnoDB uses it as the clustered index;
- If a table doesn't have a `PRIMARY KEY` defined, MySQL will find the first `UNIQUE` index that doesn't allow `NULL` values and uses it as the clustered index;
- Furthermore, if the table has no `UNIQUE` index, a hidden clustered index named `GEN_CLUST_INDEX` is generated on a synthetic column with row ID values.

By default, the table below will have a clustered index on the `id` attribute:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    lname VARCHAR(30),
    store_id INT
) ENGINE = InnoDB;
```

3.3 Secondary Indexes

All other indexes in MySQL are known as secondary indexes [72]. In the leaves of the secondary index trees, InnoDB stores not only the columns specified for the index, but also the primary key columns. This way, if a query that uses the specified secondary index needs other data that isn't in that index, InnoDB uses the primary key value to quickly fetch the row data from the clustered index.

The table below will have a secondary index on the `store_id` attribute:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    lname VARCHAR(30),
    store_id INT,
    INDEX (store_id)
) ENGINE = InnoDB;
```

3.3.1 Unique Indexes

A `UNIQUE` index imposes the condition that, all the values in said index are different from each other. If you use a prefix value for a column in a `UNIQUE` index, the column values must also be unique within the prefix length. Since all the values in the key column are unique, you can't add a new row with an existing key value [75].

The code below shows one of the ways to create a unique index on the `store_id` attribute:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    lname VARCHAR(30),
    store_id INT,
    UNIQUE KEY unique_store_id (store_id)
) ENGINE = InnoDB;
```

3.3.2 Full-Text Indexes

InnoDB supports the creation of `FULLTEXT` indexes for text-based columns like `CHAR`, `VARCHAR` and `TEXT`. This type of indexing does not support prefixing. These indexes have an inverted index design. This means that they store a list of words, and for each of them, a list of documents that the word appears in [32].

The code below shows one of the ways to create a full-text index on the `lname` attribute:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
```

```
    lname VARCHAR(30),
    store_id INT,
    FULLTEXT text_index_lname (lname)
) ENGINE = InnoDB;
```

3.3.3 Multi-Valued Indexes

With MySQL 8.0.17 or above, InnoDB supports multi-valued indexes. These are secondary indexes defined on a column that stores an array of values. Multi-valued indexes can have multiple index records for a single data record and are used for indexing JSON arrays [46].

Assuming we have a JSON file, `empinfo`, with the following information,

```
empinfo = {
    "user":"Bob",
    "zipcode":[94477,94536]
}
```

the code below shows one of the ways to create a multi-valued index on the `zipcode` portion:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    lname VARCHAR(30),
    store_id INT,
    empinfo JSON,
    INDEX zips( (CAST(empinfo->'$.zipcode' AS UNSIGNED ARRAY)) )
) ENGINE = InnoDB;
```

3.3.4 Spatial Indexes

The InnoDB storage engine supports spatial columns, such as `POINT` and `GEOMETRY`, and the creation of `SPACIAL` indexes [68]. There are several rules for creating `SPACIAL` indexes, such as:

- They cannot be created over multiple spatial columns;
- The indexed columns must be `NOT NULL` and no prefix indexing is allowed;
- These indexes cannot be primary keys or unique indexes.

The code below shows one of the ways to create a spacial index on the `g` attribute:

```
CREATE TABLE geom (
    g GEOMETRY NOT NULL SRID 4326,
    SPATIAL INDEX(g)
);
```

3.4 Hash Indexes

InnoDB does not completely support hash indexes. However, in some very particular cases, it may use the Adaptative Hash Index described in the next section.

3.4.1 Adaptive Hash Index

InnoDB can perform an Adaptive Hash Index. This is an in-memory hash index that is used only when a table fits entirely in main memory [7]. This feature is enabled by the `innodb_adaptive_hash_index` variable and turned off at server startup by `--skip-innodb-adaptive-hash-index`.

InnoDB builds an hash index, automatically, for the pages of the index that are accessed often. It uses a prefix of the index key, which may lead to only having some values of the B+-tree appear in the hash index. This index speeds up queries by making the direct lookup of any element possible, because it uses the index value as a sort of pointer.

3.5 Inconsistent Structures

In InnoDB, all user activity occurs inside a transaction [38]. Due to this, for a structure to be temporarily inconsistent the `autocommit` variable as to be set to zero, like so: `SET autocommit = 0`. This makes it so that the session always has an open transaction and allows the data structures to be inconsistent, until the changes are explicitly committed by the user.

3.6 Comparison with Oracle 18c

Even though Oracle allows the organization of files to be made with clustered indexes, the default organization is with a heap structure.

Both Oracle and MySQL support R-tree indexing for spatial indexes, B+-tree indexing and create one B+-tree index for each primary key. However, contrary to MySQL, Oracle automatically creates B+-tree indexes for each `UNIQUE` attribute and also supports Bitmap indexing [84]. They both allow the creation of multiple indexes with the same columns as keys.

4 Query Processing and Optimization

In order to get the best performance out of MySQL queries, we need to thoroughly understand how MySQL processes, optimizes and executes them.

As you can see in Figure 3, when a client queries the server, there are several steps until he receives the results of said query [5]:

1. First, the server checks the query cache. If there is a hit, then it returns the stored results; otherwise, it passes the SQL statement to the next step;
2. Secondly, the server parses, preprocesses and optimizes the SQL query into a query execution plan;
3. Then, the query execution engine executes the plan;
4. Finally, the server sends the results back to the client.

In the following sections, we will explain these steps in more detail.

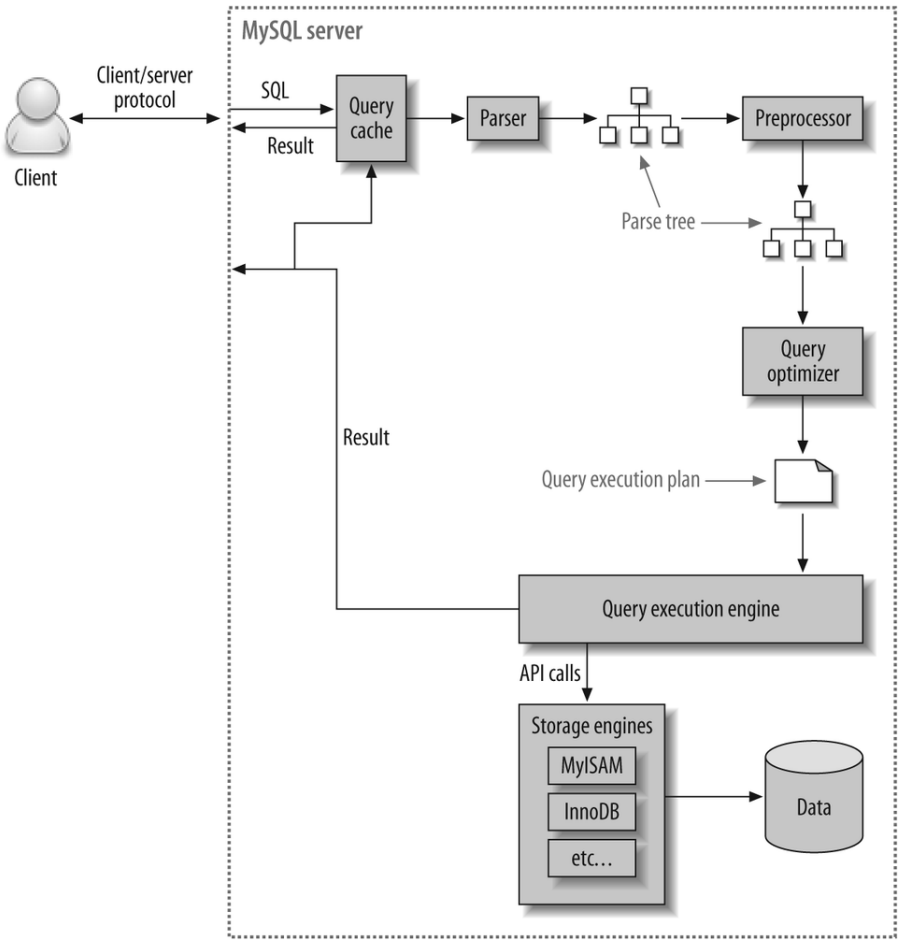

Figure 3: Execution path of a query in MySQL. Taken from [5].

4.1 The Parser and the Preprocessor

The first step after a query cache miss, is that the parser breaks the SQL query into several parts and validates them. With theses parts, it builds a "parse tree" according to the syntactical scheme of relational algebra [3].

Then, the preprocessor checks the resulting tree for other semantic errors that the parser may not have been able to detect and solve. It checks that the tables and columns used in the query exist, and resolves their names and aliases to guarantee that they aren't ambiguous. Finally, the preprocessor checks the user's privileges to make sure that he can perform said query [5].

4.2 The Query Optimizer

Now that there is a valid parse tree, the query optimizer will turn it into a query execution plan [5]. Seen as there can be several different ways of executing the same query, there are multiple execution plans and it is the optimizer's job to find the best one, according to the estimated execution performance.

First, the optimizer expands the parse tree, forms multiple alternative execution plans and then tries to predict their execution cost and choose the least expensive one. The estimation of the execution costs is based on statistics, such as:

- The number of pages per table or index;
- The cardinality of indexes;
- The length of rows and keys;
- The key distribution.

Even though MySQL caches recently used pages in the buffer pool, the optimizer does not include the effects of any type of caching in its estimates, assuming every read will result in a disk I/O operation.

It is important to note that the chosen plan, though it might not always be optimal, will at least be near optimal. There are several reasons for this:

- The statistics could be wrong. For example, the InnoDB storage engine doesn't maintain accurate statistics about the number of rows in a table;
- The cost metric is not exactly equivalent to the true cost of running the query, so even when the statistics are accurate, the query may be more or less expensive than the estimate;
- MySQL's idea of optimal might not match the user's. Sometimes, MySQL doesn't use cost-base optimization and uses predefined rules;
- MySQL doesn't take concurrent running queries into account;

- The optimizer doesn't consider the cost of operations that are not under its control, such as executing stored functions or user-defined functions;
- A single query can have an infinite number of possibilities of execution plans and the optimizer can't always estimate every possible one, so it may miss an optimal plan.

MySQL performs two types of optimization:

1. **Static Optimizations:** Can be performed simply by inspecting the parse tree. For example, the optimizer can transform clauses into equivalent forms by applying algebraic rules. They can be performed once and will always be valid, even when the query is re-executed with different values. They are a kind of "compile-time optimization";
2. **Dynamic Optimizations:** Are based on context and can depend on many factors, such as table and index statistics and which value is in a clause. They must be reevaluated each time the query is executed. They are a kind of "run-time optimization".

4.3 Optimizing through Indexes

As we have seen in Section 3, MySQL automatically creates clustered indexes, indexes for the primary key of each table that contains all the row data. These are very important in time-sensitive queries, so the user should always define a set of primary key columns for each table [54].

The user should also remember that the primary key columns are duplicated in all secondary indexes. As such, they should not specify too many or too long columns in the primary key, because it will reduce the performance.

The user should not create a separate secondary index for each column, because each query can only make use of one index. If the user creates indexes on columns that are rarely tested or columns with very few distinct values, they might not be helpful.

If many queries are made to the same table, it is smart to test indexes with different combinations of columns or experiment with a small number of concatenated indexes instead of a large number of single-column indexes. If an index contains all the columns needed for the query result, the query might not even read the table data.

The user should always specify if a column cannot contain NULL values using `NOT NULL`. This helps the optimizer determine which index is most effective to use for a query.

4.4 Optimizing Simple Operations

4.4.1 The WHERE clause

The optimization of the `WHERE` clause is done using a series of transformations and operations, such as [81]:

- **Removal of unnecessary parentheses:**

```
((a AND b) AND c OR (((a AND b) AND (c AND d))))

RESULT: (a AND b AND c) OR (a AND b AND c AND d)
```

- **Replacement of constants for values:**

```
(a<b AND b=c) AND a=5

RESULT: b>5 AND b=c AND a=5
```

- **Removal of constant conditions:**

```
(b>=5 AND b=5) OR (b=6 AND 5=5) OR (b=7 AND 5=6)

 RESULT: b=5 OR b=6
```

There are many other optimizations that can be done for the `WHERE` clause. However, we will not detail them in this report.

4.4.2 The ORDER BY clause

There are two strategies for optimizing the `ORDER BY` clause [57]:

Using Indexes

First, MySQL checks if it can use an index. This method is the best if the index contains all columns needed for the result. However, if the query contains columns which are not in the index, it may require the look up of table rows that are not indexed, making it more expensive than scanning the table and sorting the results.

Using Filesort

If an index cannot be used to satisfy an `ORDER BY` clause, an operation that reads table rows and sorts them is used - the `filesort` operation.

The optimizer uses the `sort_buffer_size` variable to allocate a memory buffer, but it increases it as needed. Because of this, this variable can be set to larger values in order to improve large sorts, without consuming excessive memory in smaller sorts. If a large amount of space that doesn't fit in memory is needed, the optimizer uses a temporary disk file.

4.4.3 The GROUP BY clause

The most basic way to satisfy a `GROUP BY` clause is to perform a Full Table Scan and create a new temporary table where all rows from each group are consecutive [33]. Then this temporary table is used to find groups and apply aggregate functions, if there are any to apply.

However, in some cases, MySQL is able to perform much better and avoid the creation of temporary tables altogether by using indexes. The most important conditions for using indexes are that all `GROUP BY` columns reference attributes from the same index and that the index stores its keys in order. There are two ways to execute a `GROUP BY` query through index access:

Loose Index Scan

The most efficient way to do a `GROUP BY` is when an index is used to directly retrieve the grouping columns. Because the keys are ordered in B+-tree indexes, this enables the use of lookup groups without having to take into account all keys in the index that satisfy all `WHERE` conditions.

When there is no `WHERE` clause, a Loose Index Scan reads as many keys as the number of groups, which may be a much smaller number than that of all keys. If the `WHERE` clause contains range predicates, a Loose Index Scan looks up the first key of each group that satisfies the range conditions and reads the smallest possible number of keys. This is only possible if the following conditions are met:

- The query is performed on a single table;
- The `GROUP BY` only has columns that form a leftmost prefix of the index and no other columns. If a table has an index on `(c1, c2, c3)`, this algorithm is only applicable if we have `GROUP BY c1, c2`. It is not applicable if we have `GROUP BY c2, c3` or `GROUP BY c1, c2, c4`;
- The aggregate functions used in the select are `MIN()` or `MAX()`, and all of them refer to the same column. This column must be in the index and must immediately follow the columns in the `GROUP BY`;
- All other parts of the index that are not in the `GROUP BY` must be referenced in equalities with constants;
- Full columns values must be indexed, not just prefixes.

Tight Index Scan

When we can't use a Loose Index Scan, it may still be possible to avoid the creation of temporary tables. If there are range conditions in the `WHERE` clause, the Tight Index Scan reads only the keys that satisfy these conditions. Otherwise, it performs an Full Index Scan. The grouping operation is performed after all needed keys have been found.

4.4.4 The JOIN clause

When executing joins between tables, MySQL uses the Nested-Loop join algorithm or the Block Nested-Loop join algorithm [52]. However, since version 8.0.18, MySQL began employing the Hash join algorithm for any query in which each join has an equi-join condition and uses no indexes [34].

- **Nested-Loop Join Algorithm:** This algorithm reads rows from the outer table of a loop one at a time, passing each row to an inner loop that processes the next table in the join.
- **Block Nested-Loop Join Algorithm:** This algorithm reads blocks of rows in the outer loop to reduce the number of times that tables in inner loops must be read.
- **Hash Join Algorithm:** This algorithm uses a hash function to partition the tuples of both tables and then performs the join in each of the partitions. This is usually faster than the Block Nested-Loop algorithm when there are equi-joins or natural joins.

These algorithms were studied in the lectures, so we will not go into more detail about their implementations.

In the next sections, we will talk about the optimizations and simplifications done by MySQL when processing the `JOIN` clause.

4.4.4.1 OUTER and INNER JOIN

MySQL outer joins include the `LEFT JOIN` and the `RIGHT JOIN`, but internally the optimizer converts any `RIGHT JOIN` into the equivalent `LEFT JOIN`, by reversing its roles, like so [59]:

```
T1 RIGHT JOIN T2 ON P(T1, T2)

RESULT: T2 LEFT JOIN T1 ON P(T1, T2)
```

All inner join expressions of the form `T1 INNER JOIN T2 ON P(T1,T2)` are replaced by the list `T1,T2, P(T1,T2)` and joined with the `WHERE` condition as well.

If a query with a `LEFT JOIN` has a `WHERE` condition that is always false for the generated NULL row, the `LEFT JOIN` is changed to an inner join, like so [58]:

```
SELECT * FROM t1 LEFT JOIN t2 ON (column1) WHERE t2.column2=5;

RESULT: SELECT * FROM t1, t2 WHERE t2.column2=5
        AND t1.column1=t2.column1;
```

4.5 Optimizing Complex Operations

4.5.1 Materialization

The optimizer uses materialization as an strategy to make the processing of subqueries more efficient. Materialization consists of storing subquery results in a temporary table and, when the results are needed again, MySQL simply reads them [56].

If possible, this method uses an in-memory temporary table. If the table is too large, it uses on-disk storage.

This optimization strategy is used by default [71]. However, if the user wants to disable it they can simply do:

```
SET [GLOBAL|SESSION] optimizer_switch='materialization=off';
```

4.6 Controlling the Query Optimizer

MySQL has many ways of controlling the query optimizer. It can use system variables, which are used in switchable optimizations and also affect how query plans are chosen and evaluated. Besides that, it can use optimizer hints [17].

4.6.1 Number of Evaluated Plans

Regarding the number of plans that the optimizer evaluates, the user can control them by using two system variables [16]:

1. The `optimizer_prune_level` variable tells the optimizer to skip certain plans based on estimates of the number of rows accessed for each table. This usually reduces query compilation times dramatically and is turned on by default - `optimizer_prune_level=1`;

2. The `optimizer_search_depth` variable tells the optimizer how deeply he should look into each incomplete plan to know if it should be expanded any further. This variable should be used very carefully because it can make the compile time vary from less than a minute to several hours. If the user is unsure of what a reasonable value is, this variable should be set to 0 to tell the optimizer to determine the value automatically.

4.6.2 Switchable Optimizations

The `optimizer_switch` system variable is used to gain control over optimizer behavior. This variable is set by using flags that specify the wanted optimizer behavior and it can be enabled or disabled by using `on` and `off`. This variable also has global and session values and can be changed at runtime [71].

It is important to note that, any changes to the `optimizer_switch` variables affect execution of all subsequent queries and, to affect one query differently from another it is necessary to change the variables before each one.

An example of the use of these variables has been shown before with materialization, in the Section 4.5.1.

To switch off the use of Block Nested-Loops you could use the following instruction:

```
SET [GLOBAL|SESSION] optimizer_switch='block_nested_loop=off';
```

4.6.3 Optimizer Hints

A way of controlling the optimizer is to specify optimizer hints inside individual statements. Hints within a statement take precedence over switching the `optimizer_switch` flags [53].

To force a specific join order you could use the following hint:

```
SELECT/*+ JOIN_ORDER(t1,t2)*/ *
FROM t1 INNER JOIN t2
WHERE ...
```

To force or ignore the use of an index you could use the following hints [36]:

```
SELECT *
FROM t1 USE INDEX(col1Index,col2Index)
WHERE ...

SELECT *
FROM t1 IGNORE INDEX(col3Index)
WHERE ...
```

To force the use of a specific join algorithm you could use the following hint [53]:

```
SELECT /*+ NO_BNL() HASH_JOIN(t1,t2)*/*
FROM t1 INNER JOIN t2 INNER JOIN t3
```

To set system variables for the duration of the statement you could use the following hint [53]:

```
INSERT /*+ SET_VAR(foreign_key_checks=OFF) */
INTO t2 VALUES(2);
```

Concluding Section 4.6, it is best not to try to outsmart the optimizer. By doing this, you may end up defeating its purpose.

However, it is important to note that sometimes you may know something about the data that the optimizer doesn't. This frequently happens with database administrators or users that know a database inside out. If you know for sure that the optimizer isn't giving you good results, and you know why, you can help it by using the tools described in this section.

4.7 Estimating Table and Index Statistics

The estimation of query performance is made by counting the number of disk seeks. When using B+-tree indexes, you need this many seeks to find a row [27]:

$$\frac{log(row\_count)}{log(\frac{index\_block\_length}{3} * \frac{2}{(index\_length+data\_pointer\_length)}} + 1 \quad (1)$$

Optimizer statistics are persisted to disk by default. If you turn this behaviour off by using `innodb_stats_persistent=OFF` or create tables with `STATS_PERSISTENT=0`, statistics are not persisted to disk [14]. Instead, they are stored in memory, and are lost when the server shuts down. Statistics are also updated periodically by certain operations and under certain conditions.

To update non-persistent optimizer statistics you could:

- Run `ANALYZE TABLE`;
- Run `SHOW TABLE`;
- Run `SHOW INDEX`;
- Query `INFORMATION_SCHEMA.TABLES` or `INFORMATION_SCHEMA.STATISTICS` with the `innodb_stats_on_metadata` option enabled.

4.8 The Execution Plan

After the parsing and optimizing stages, a query execution plan is outputted. This plan is used by the query execution engine to process the query. Contrary to many other databases, in MySQL this plan is not byte-code but a tree of instructions [5].

After executing the plan, the results are sent back to the client. It is important to note that the server generates and sends results incrementally. If the query is cacheable, MySQL will also place the results into the query cache.

To display detailed information from the optimizer about a statement execution plan, we use `EXPLAIN` with an explainable statement: `SELECT`, `DELETE`, `INSERT`, `REPLACE` or `UPDATE`. MySQL explains how it would process the statement, including information about how tables are joined and in which order [55].

The syntax to use the EXPLAIN statement is the following [28]:

```
EXPLAIN
SELECT t1.a, t1.a IN (SELECT t2.a FROM t2) FROM t1;
```

4.9 Comparison with Oracle 18c

Generally speaking, the Oracle optimizer is very similar to the MySQL optimizer, in the sense that it performs query transformations and optimizations in order to obtain better performance. Just like in MySQL, Oracle stores estimations of table and index statistics to use when evaluating the cost of an execution plan.

However, when you delve into the details of Oracle's optimization, you can clearly see that it is more complete. Besides performing materialization, as MySQL does, it also allows pipelining. Furthermore, besides implementing the Full Table and Index Scans, Oracle implements several others, such as [88]:

- Table access by row id;
- Sample Table Scan;
- Index Fast Full Scan;
- Index Skip Scans;
- Index Join Scans;
- In-Memory Dynamic Scans;
- Etc.

As for the join algorithms available, Oracle provides one more than MySQL, the Sort Merge join [87].

Both MySQL and Oracle allow the user to specify which indexes and algorithms to use through hinting within statements.

Lastly, just like MySQL does, Oracle has an instruction that allows the user to see the details of the execution plan.

5 Transaction Management and Concurrency Control

5.1 Transactions

A transaction is a unit of program execution that accesses and possibly updates various data items in a database [2]. To preserve the integrity of data, the InnoDB storage engine adheres closely to the ACID model. Its properties guarantee that data is not corrupted and results are not distorted by exceptional conditions, such as software crashes and hardware malfunctions [37].

Below, we present an example of how to start a transaction in MySQL, then update a table and commit the changes [73]:

```
START TRANSACTION;
UPDATE t SET b = 5 WHERE b = 3;
COMMIT;
```

If, instead of committing the changes, the user wanted to cancel all modifications made by the current transaction, he could just use the `ROLLBACK;` statement.

In MySQL, transactions cannot be nested. This happens due to the implicit commit performed for any current transaction when you use the `START TRANSACTION` statement [69].

However, it is possible for a user to "mimic" this functionality by using savepoints [6].

As we said before, the transactions implemented by InnoDB have ACID properties. Long Duration Transactions are transactions that take a lot of time, so it is unlikely that isolation can be guaranteed. Without isolation, the atomicity may be compromised and, as such, this kind of transaction isn't implemented by InnoDB.

5.2 The ACID Model

The following sections discuss how the InnoDB storage engine deals with the categories of the ACID model.

5.2.1 Atomicity

In the ACID model, the A stands for Atomicity. Atomicity ensures that, within a transaction, either all operations are properly reflected in the database or none are [2]. MySQL with InnoDB engine guarantees this property by using the following features [37]:

- **Autocommit Setting:** By default, MySQL has the autocommit setting turned on. This setting treats each statement as atomic and commits the work after said statement has finished. If an error occurs during the statement execution a rollback statement is issued;

- **Commit Statement:** MySQL with InnoDB engine implements the `COMMIT` statement, which makes the current transaction's changes permanent. If you have autocommit enabled, it automatically commits after each statement. Because of this, the user should only use the `COMMIT` statement when a `START TRANSACTION` was issued or when autocommit is turned off;

- **Rollback Statement:** MySQL with InnoDB engine implements the `ROLLBACK` statement, which cancels the current transaction's changes. As explained above, this statement is used when a `START TRANSACTION` is issued or when autocommit is turned off;

- **Information Schema table:** With the `INFORMATION_SCHEMA` table, `INNODB_TRX`, we can check information about every transaction that is executing inside InnoDB. When using this table, we can see the transaction state, when the transaction started and the particular SQL statement the transaction is executing [39].

5.2.2 Consistency

In the ACID model, the C stands for Consistency. Consistency ensures that when the execution of a transaction ends, it preserves the state of the database [2]. Guaranteeing the consistency property essentially involves internal InnoDB processing to protect data from crashes. MySQL with InnoDB engine guarantees this property by using the following features [37]:

- **Doublewrite Technique:** This technique allows InnoBD to recover a page in case of a system crash in the middle of a page write, because it keeps a copy of said page in the doublewrite buffer. In a nutshell, InnoDB first writes and flushes pages to the doublewrite buffer. Finally, after it has completed the write and flush, InnoDB writes the pages to their correct positions in the data file [25];

- **Crash Recovery:** After a crash, when MySQL is re-started, it starts cleanup activities. These activities consists of scanning tables for incomplete transactions and then, using the redo log, starting them again. It checks for changes that were committed before the crash, but weren't written into the data files and uses the doublewrite buffer to reconstruct them. However, if the database is shut down normally, this type of activity is performed during shutdown by the purge operation [18].

In an SQL statement that inserts, deletes, or updates many rows, foreign key constraints are checked row-by-row. InnoDB sets shared row-level locks on child or parent records that it must examine and checks the constraints immediately. The check cannot be deferred to a later moment, for example to the transaction commit [31].

5.2.3 Isolation

In the ACID model, the I stands for Isolation. Isolation ensures that, even though multiple transactions may execute concurrently, each one must be unaware of other concurrently executing transactions and mustn't see their intermediate results [2].

Transaction isolation is at the foundation of database processing. Isolation levels allow the user to balance performance, reliability, consistency and reproducibility of results, when multiple transactions run concurrently [73].

InnoDB uses a combination of multi-versioning properties with traditional two-phase locking [42]. Multi-version concurrency control allows InnoDB transactions with certain isolation levels to query rows that are being updated by other transactions and see their values before those updates occurred [47].

5.2.3.1 Isolation Levels

InnoDB offers all four standard SQL transaction isolation levels [73]:

1. `READ UNCOMMITTED`: Uncommitted records can be read. `SELECT` statements are performed without using locks. When using this isolation level, reads are not consistent and are also called a dirty reads;

2. `READ COMMITTED`: Only committed records can be read. However, each consistent read, even within the same transaction, sets and reads its own fresh snapshot. This means that successive reads of a record may return different committed values;

3. `REPEATABLE READ`: This is the default isolation level for InnoDB. Only committed records can be read. However, consistent reads within the same transaction read the snapshot established by the first read. This means that the record values read within each transaction will always be consistent with each other;

4. `SERIALIZABLE`: Completely isolates the effects of one transaction from all others. Tries to ensure that transactions run in such a way that they appear to be executed one at a time, rather than concurrently.

InnoDB supports each of the transaction isolation levels described using different locking strategies.

If the user wants to perform operations that need a high degree of consistency, the `REPEATABLE READ` isolation level can be used. On the other hand,

if the user wants to perform operations with a lower the degree of consistency, he can do so by using `READ COMMITTED` or `READ UNCOMMITTED`. For stricter rules than the repeatable read, the user may choose the `SERIALIZABLE` level.

The user can change the isolation level for a single session or for all subsequent connections like so [65]:

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

5.2.3.2 Savepoints

With the InnoDB engine, the user can make use of savepoints [64]. The following code creates a transaction savepoint with a name of identifier:

```
SAVEPOINT transaction_id1;
```

It is important to note that, if the current transaction has a savepoint with the same name, the old savepoint is deleted and a new one is set.

To undo the modifications that the current transaction made to rows after the savepoint was set, the user can use the following code:

```
ROLLBACK TO SAVEPOINT transaction_id1;
```

Note that savepoints that were created at a later time than the named savepoint are deleted.

To remove a named savepoint from the set of savepoints of the current transaction without performing a commit or a rollback, the user can use the following code:

```
RELEASE SAVEPOINT transaction_id1;
```

If the user issues a `COMMIT` or a `ROLLBACK` statement, all savepoints of the current transaction are deleted.

5.2.4 Durability

In the ACID model, the D stands for Durability. Durability ensures that after a transaction completes successfully, the changes it has made to the database persist, even if there are system failures [2].

Guaranteeing durability with MySQL involves software features interacting with particular hardware configurations. There are many possibilities that depend on the capabilities of the CPU, network and storage devices. We will not go into detail about these specifications because it is very hard to provide guidelines for them [37].

However, as we mentioned in Section 5.2.2, InnoDB uses the doublewrite buffer as a storage area to write pages flushed from the buffer pool, before writing them to their corresponding positions in the data files [24]. This

guarantees that, if there is a system crash, InnoDB can find good copies of the pages in the doublewrite buffer during crash recovery.

Even though data is written twice, this doesn't require twice as much I/O overhead or operations, because the data is written to the doublewrite buffer in a large sequential chunk, with a single `fsync()` call to the operating system.

5.3 Concurrency Control

5.3.1 Lock Granularity

InnoDB implements standard row-level locking where there are two types of locks [40]:

1. **Shared (S) Locks:** Allow the transaction that holds the lock to read a row;
2. **Exclusive (X) Locks:** Allow the transaction that holds the lock to update or delete a row.

If transaction `T1` holds an S lock on `row r` and another transaction `T2` requests a lock on the same row, then:

- If `T2` requested an S lock, it can be granted immediately. This means that both `T1` and `T2` now hold an S lock on `row r`;
- If `T2` requested an X lock, it cannot be granted immediately.

If transaction `T1` holds an X lock on `row r` and another transaction `T2` requests either type of lock on the same row, then it cannot be granted immediately, and `T2` has to wait for the lock to be released.

Because InnoDB supports multiple granularity locking, it allows for the coexistence of row locks and table locks.

So, InnoDB implements intention locks, table-level locks that specify which type of lock a transaction requires later for a row in a table. There are two types of intention locks:

1. **Intention Shared (IS) Locks:** Specify that a transaction intends to set an S lock on individual rows in a table;
2. **Intention Exclusive (IX) Locks:** Specify that a transaction intends to set an X lock on individual rows in a table.

With intention locking, before a transaction can acquire an S lock on a table row, it must first acquire an IS lock or stronger on the table.

Additionally, before a transaction can acquire an X lock on a table row, it must first acquire an IX lock on the table.

To obtain an IS lock, use the following code:

```
SELECT * FROM table1
WHERE id = 1
FOR SHARE;
```

To obtain an IX lock, use the following code:

```
SELECT * FROM table1
WHERE id = 1
FOR UPDATE;
```

Table 1 represents the compatibility of Table-level lock types:

Table 1: Table-Level Lock Type Compatibility

	X	IX	S	IS
X	Conflict	Conflict	Conflict	Conflict
IX	Conflict	Compatible	Conflict	Compatible
S	Conflict	Conflict	Compatible	Compatible
IS	Conflict	Conflict	Compatible	Compatible

A lock is only granted to a transaction if it is compatible with all other existing locks. Otherwise, a transaction waits until the conflicting existing lock is released. However, if there is a conflict with an existing lock that cannot be granted because it would cause a deadlock, an error occurs. Details on how InnoDB deals with deadlocks will be given in Section 5.3.3.

The goal of intention locks is to show that someone is locking a row, or going to lock a row in a table. As such, intention locks do not block anything except full table requests:

```
LOCK TABLE table1 WRITE;
```

5.3.2 Row-Level Locks

- **Record Lock:** A lock on an index record. This type of lock prevents any other transaction from inserting, updating or deleting rows from an index record. They lock the index record even if there is no index on such table, due to the existence of the clustered index;

  The following code prevents any other transaction from inserting, updating, or deleting rows where the value of `t.c1` is 10:

  ```
  SELECT c1 FROM t WHERE c1 = 10 FOR UPDATE;
  ```

- **Gap Lock:** Locks all index records within a range, or the gap before the first or after the last index record. In InnoDB, the purpose of a Gap Lock is to prevent other transactions from inserting to the gap so, multiple Gap Locks can co-exist;

  The following code prevents any other transactions from inserting values between 10 and 20 into column `t.c1`:

  ```
  SELECT c1 FROM t WHERE c1 BETWEEN 10 and 20 FOR UPDATE;
  ```

- **Next-Key Lock:** A lock that is a combination of a Record lock on an index record and a Gap lock on the gap before that index record. A Next-Key lock on an index record also affects the gap between that record and the one before it. For example, suppose that an index contains the values 10, 11, 13, and 20. If a Next-Key lock is created for record 20 in the index, another session cannot insert a new index record in the gap between 13 and 20;

- **Insert Intention Locks:** A type of Gap lock that is set for `INSERT` operations. This lock allows multiple transactions to insert into the same index gap without waiting for each other, if they are not inserting at the exact same position within the gap.

5.3.3 Deadlocks

A deadlock is a situation in which different transactions are unable to proceed because each of them holds a lock that the other needs. This happens because both transactions are waiting for a resource to become available, while holding a resource that the other transaction needs. This situation can occur, for example, when transactions lock rows in multiple tables in the opposite order[21].

Below, we illustrate how a deadlock can occur in MySQL. The following example involves two clients, A and B [22]:

```
/* Client A inserts one row into a table, then
starts a transaction and obtains
an S lock on the row */

INSERT INTO t (id) VALUES(1);
START TRANSACTION;
SELECT * FROM t WHERE id = 1 FOR SHARE;

/* Client B starts a transaction and tries to
delete the same row from the table */

START TRANSACTION;
DELETE FROM t WHERE id = 1;
```

The delete operation requires an X lock that cannot be granted because it is incompatible with the S lock that client A has. So, client B is blocked while waiting for client A to release his S lock.

```
/* Client A also attempts to delete the
same row from the table */

DELETE FROM t WHERE id = 1;
```

Deadlock occurs because client A needs an X lock to delete the row. That lock cannot be granted because client B already requested an X lock and is waiting for client A to release its S lock. The S lock held by A cannot be upgraded to an X lock because of the prior request by B for an X lock. So, InnoDB generates an error for one of the clients, releases its locks and grants the other clients lock.

5.3.3.1 Deadlock Detection and Rollback

In MySQL, deadlock detection is enabled by default. When InnoDB detects a deadlock, it chooses a transaction to roll back, the victim. When choosing the victim, InnoDB tries to pick small transactions to roll back, with the size of a transaction being determined by the number of rows inserted, updated, or deleted [20].

When deadlock detection is disabled by using the `innodb_deadlock_detect` configuration option, InnoDB relies on the `innodb_lock_wait_timeout` setting to roll back transactions in case of a deadlock.

5.3.3.2 Minimizing Deadlocks

To diminish the possibility of deadlocks, MySQL recommends:

- Using transactions instead of locking tables;
- Keeping transactions as small and short as possible in order to avoid collisions;
- When modifying multiple tables within a transaction, doing those operations in a consistent order each time;
- Creating indexes on columns used by the `SELECT ... FOR UPDATE` and the `UPDATE ... WHERE` statements;
- Trying to re-issue a transaction if it fails due to deadlock.

5.4 Comparison with Oracle 18c

Oracle and MySQL are very alike when it comes to transactions, as both follow the ACID model. Regarding isolation levels, both implement the standard ones but have different defaults: Oracle uses the `READ COMMITTED` and MySQL uses the `REPEATABLE READ`. They both support the use of savepoints with similar syntax.

When it comes to locking, they are both very similar, with the exception that Oracle allows the deferral of constraint checking, and MySQL does not [83].

As for deadlocks, they use the similar tactics for handling them, by choosing a victim. However, they differ on the method of choosing such victim: Oracle's victim is the transaction that detected the deadlock [85]; MySQL's victim is the transaction with the least amount of rows inserted, updated or deleted.

6 Other characteristics of MySQL

MySQL has a lot of functionalities that do not align directly with the subjects lectured in the Database Systems course. Of those functionalities, we will present some of the ones that we found to be more interesting.

6.1 Web Support

MySQL provides Web support through XML and XPath.

6.1.1 XML Support

6.1.1.1 Exporting XML

MySQL provides a way to obtain XML-formatted output by starting the `mysql` or `mysqldump` clients with the `--xml` option [82]. We get the same output using either client.

Using `mysql`, the command will be as follows [48]:

```
mysql --xml -uroot -e "SHOW VARIABLES LIKE 'version%'"
```

To write the output to an XML file, do as follows:

```
mysql --xml -e 'SELECT * FROM mydb.table1' > fileName.xml
```

Using `mysqldump`, the command will be as follows [51]:

```
mysqldump --xml -u root world City
```

6.1.1.2 Importing XML

MySQL also allows us to import data from an XML file into a table using the `LOAD XML` statement [45]. This is complimentary to running the `mysql` client in XML output mode.

The following example shows how you could import data from an XML file into `table1`:

```
LOAD XML LOCAL INFILE "/dirPath/fileName.xml"
INTO TABLE table1
```

6.1.2 XPath Support

MySQL manipulates XML data using XPath, a language for navigating and querying XML documents. For this, if offers two functions:

1. `ExtractValue(xml_far, xpath_expr);`
2. `UpdateXML(xml_target, xpath_expr, new_xml).`

The `ExtractValue` function receives an XML markup fragment and an XPath expression and returns the text of the first text node which is a child of the element or elements matched by the XPath expression. In case of multiple matches, they are returned as a single space-delimited string. If there are no matches, an empty string is returned.

The `UpdateXML` function receives two XML markup fragments and an XPath expression. It replaces a portion of an XML fragment with the new XML fragment and returns the changed XML. The portion replaced is the one that matches the XPath expression. If no expression matching the given one is found, or if multiple matches are found, the function returns the original XML [82].

6.2 Stored Routines

MySQL supports two types of stored routines: procedures and functions. A stored routine is a set of SQL statements that can be stored in the server. Then, the clients don't need to reissue the individual statements but can make a reference to the stored routine [77].

These stored routines can be very useful:

- When multiple client applications use different languages or platforms, but need to perform the same operations;
- If security is very important, applications and users could have no access to the database tables themselves and only be able to execute specific stored routines.

In the following sections, we will demonstrate how to manage procedures.

6.2.1 Create Procedure

The following code shows how to create a procedure that counts the number of cities in a country [19]:

```
CREATE PROCEDURE citycount (IN country CHAR(3), OUT cities INT)
       BEGIN
         SELECT COUNT(*) INTO cities FROM world.city
         WHERE CountryCode = country;
       END;
```

6.2.2 Alter Procedure

The following code shows how to alter the previously defined procedure [8]:

```
ALTER PROCEDURE citycount [characteristic ...]

characteristic: {
```

```
    COMMENT 'string'
  | LANGUAGE SQL
  | {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}
  | SQL SECURITY {DEFINER | INVOKER}
}
```

The `ALTER PROCEDURE` statement can be used to change the characteristics of a stored procedure. However, it is not possible to change the parameters or body by using this statement. In order to do that, the user must `DROP` the procedure and re-create it.

6.2.3 Drop Procedure

The following code shows how to drop the previously defined procedure [26]:

```
DROP PROCEDURE [IF EXISTS] citycount;
```

6.3 Triggers

A trigger is a named database object that is associated with a table. This object is activated when a particular event for that table occurs. Triggers are used to perform checks of values to be inserted into a table or to perform calculations on values involved in an update [78].

In the following sections, we will demonstrate how to manage triggers.

6.3.1 Create Trigger

The following code shows how to create a trigger that accumulates the values inserted into one of the columns of the table [74]:

```
CREATE TRIGGER ins_sum BEFORE INSERT ON account
    FOR EACH ROW SET @sum = @sum + NEW.amount;
```

6.3.2 Drop Trigger

The following code shows how to drop the previously defined trigger:

```
DROP TRIGGER schema.ins_num;
```

With the `DROP TRIGGER` statement, the user must specify the schema name if the trigger is not in the default schema. Trigger names exist in the schema namespace. This means that all triggers must have unique names within a given schema.

It is important to note that when you drop a table, all the existing triggers related to that table are also dropped [74].

6.4 JDBC DriverManager Interface

With MySQL, it is possible to use Java to send instructions to a database by using Java Database Connectivity (JDBC). To establish a connection, the user can import the `DriverManager` class [15]. The following code is an example of how to create a connection and use it to query a database [76]:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;

Connection conn = null;
Statement stmt = null;
ResultSet rs = null;

try {
     conn =
       DriverManager.getConnection(
       "jdbc:mysql://localhost/test?"
        + "user=minty&password=greatsqldb");

    stmt = conn.createStatement();
    rs = stmt.executeQuery("SELECT foo FROM bar");
}
catch (SQLException ex){
    // Handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
finally {
    // Release resources in this block
}
```

6.5 MySQL Workbench

MySQL Workbench is a graphical tool for working with MySQL servers and databases. It supports MySQL version 5.5 and above, and has five main areas of functionality [50]:

1. **SQL Development:** Allows the creation and management of connections to database servers. It provides the capability to execute SQL queries on the database connections using the Visual SQL Editor, discussed in Section 6.5.1;

2. **Data Modeling:** Allows the creation of models of database schemas graphically and the edition of all aspects of the database;

3. **Server Administration:** Allows the creation and administration of server instances;

4. **Data Migration:** Allows migrations from Microsoft SQL Server, Sybase ASE, SQLite, SQL Anywhere, PostreSQL and other RDBMS tables, objects and data to MySQL;

5. **MySQL Enterprise Support:** Provides support for Enterprise products.

6.5.1 Visual SQL Editor

The Visual SQL Editor is a set of specialized editors (query, schema, table, etc) and panels that, together, enable you to [79]:

- Build, edit and run queries;
- Create and edit data;
- View and export results;
- Perform basic RDBMS administrative tasks.

In Figure 4, you can see the appearance of the Visual SQL Editor tool.

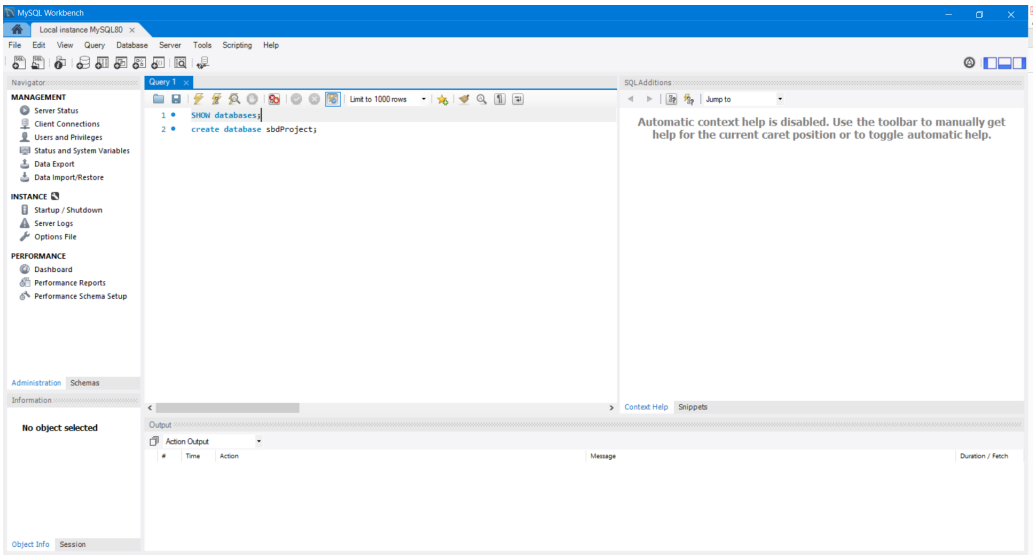

Figure 4: Screenshot of Visual SQL Editor.

7 Conclusion

Because we are Computer Science students, getting to know a new Database Management System was really important and useful to us.

Usually, the biggest issue when starting a new project is to define which technologies to use, namely the database system. So, we found the experience of researching the "ins and outs" of MySQL with the InnoDB storage engine very enriching and helpful in understanding two of the most popular DBMS, MySQL and Oracle.

With the knowledge we gathered in this extensive study, we will be able to use MySQL databases with more ease in future projects and applications. We believe that, the next time we have to choose DBMS for a new project, we will be more aware of the relevant characteristics to bear in mind.

Besides getting to know a new DBMS, we found that this project was also very effective in making us understand Oracle. It helped us with solidifying the subjects lectured in class and prepared us for the exam in a way that "regular" studying couldn't.

Overall, we found the making of this project to be a very positive experience and we believe that it was very adequate for the purpose of this course.

References

[1] *A Busy Developer's Guide to Database Storage Engines — The Basics*. Accessed: 2020-05-13. URL: https://blog.yugabyte.com/a-busy-developers-guide-to-database-storage-engines-the-basics/.

[2] *Alferes. J., Aula 9 - Chapter 17: Transactions. Lecture Slides. Database Systems, FCT-UNL. Lecture delivered on May 8th 2020*. Accessed: 2020-05-18. URL: https://drive.google.com/file/d/1Mzcrp1N8kUl6mZe8RevjUWbWoGhBYcu-/view.

[3] *Database Journal, Understanding Query Execution in Relational Database System Architecture*. Accessed: 2020-05-16. URL: https://www.databasejournal.com/features/mysql/understanding-query-execution-in-relational-database-system-architecture.html.

[4] *Hackernoon, MySQL 8 vs PostgreSQL 10*. Accessed: 2020-05-18. URL: https://hackernoon.com/showdown-mysql-8-vs-postgresql-10-3fe23be5c19e.

[5] *High Performance MySQL, 2nd Edition, Chapter 4. Query Performance Optimization*. Accessed: 2020-05-15. URL: https://www.oreilly.com/library/view/high-performance-mysql/9780596101718/ch04.html.

[6] *How can we use nested transactions in MySQL?* Accessed: 2020-05-17. URL: https://www.tutorialspoint.com/how-can-we-use-nested-transactions-in-mysql.

[7] *MySQL Documentation, Adaptative Hash Index*. Accessed: 2020-05-15. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-adaptive-hash.html.

[8] *MySQL Documentation, ALTER PROCEDURE Statement*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/alter-procedure.html.

[9] *MySQL Documentation, Alternative Storage Engines*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/storage-engines.html.

[10] *MySQL Documentation, Buffer Pool*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html.

[11] *MySQL Documentation, Change buffer*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-change-buffer.html.

[12] *MySQL Documentation, Clustered and Secondary Indexes*. Accessed: 2020-05-15. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-index-types.html.

[13] *MySQL Documentation, COLUMNS Partitioning*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/partitioning-columns.html.

[14] *MySQL Documentation, Configuring Non-Persistent Optimizer Statistics Parameters*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-statistics-estimation.html.

[15] *MySQL Documentation, Connecting to MySQL Using the JDBC DriverManager Interface*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-usagenotes-connect-drivermanager.html.

[16] *MySQL Documentation, Controlling Query Plan Evaluation*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/controlling-query-plan-evaluation.html.

[17] *MySQL Documentation, Controlling the Query Optimizer*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/controlling-optimizer.html.

[18] *MySQL Documentation, Crash Recovery from Glossary*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_crash_recovery.

[19] *MySQL Documentation, CREATE PROCEDURE and CREATE FUNCTION Statements*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html.

[20] *MySQL Documentation, Deadlock Detection and Rollback*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlock-detection.html.

[21] *MySQL Documentation, Deadlocks in InnoDB*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlocks.html.

[22] *MySQL Documentation, Deadlocks in InnoDB*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlock-example.html.

[23] *MySQL Documentation, Different Tablespaces choices while creating tables*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/using-innodb-tables.html.

[24] *MySQL Documentation, Doublewrite Buffer*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-doublewrite-buffer.html.

[25] *MySQL Documentation, Doublewrite Buffer from Glossary*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_doublewrite_buffer.

[26] *MySQL Documentation, DROP PROCEDURE and DROP FUNCTION Statements*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/drop-procedure.html.

[27] *MySQL Documentation, Estimating Query Performance*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/estimating-performance.html.

[28] *MySQL Documentation, EXPLAIN Output format*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/explain-output.html.

[29] *MySQL Documentation, File per Table Tablespaces*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-file-per-table-tablespaces.html.

[30] *MySQL Documentation, File Space Management*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-file-space.html.

[31] *MySQL Documentation, FOREIGN KEY Constraint Differences*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/ansi-diff-foreign-keys.html.

[32] *MySQL Documentation, FullText Indexes*. Accessed: 2020-05-15. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-fulltext-index.html.

[33] *MySQL Documentation, GROUP BY Optimization*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/group-by-optimization.html.

[34] *MySQL Documentation, Hash Join Optimization*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/hash-joins.html.

[35] *MySQL Documentation, HASH Partitioning*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/partitioning-hash.html.

[36] *MySQL Documentation, Index Hints*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/index-hints.html.

[37] *MySQL Documentation, InnoDB and the ACID Model*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/mysql-acid.html.

[38] *MySQL Documentation, InnoDB Autocommit, Commit, and Rollback*. Accessed: 2020-05-15. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-autocommit-commit-rollback.html.

[39] *MySQL Documentation, InnoDB INFORMATION_SCHEMA Transaction and Locking Information*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-information-schema-transactions.html.

[40] *MySQL Documentation, InnoDB Locking*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html.

[41] *MySQL Documentation, InnoDB Row Formats*. Accessed: 2020-05-14. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-row-format.html#innodb-row-format-compact.

[42] *MySQL Documentation, InnoDB Transaction Model*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-model.html.

[43] *MySQL Documentation, KEY Partitioning*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/partitioning-key.html.

[44] *MySQL Documentation, LIST Partitioning*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/partitioning-list.html.

[45] *MySQL Documentation, LOAD XML Statement*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/load-xml.html.

[46] *MySQL Documentation, Multi-Valued Indexes*. Accessed: 2020-05-15. URL: https://dev.mysql.com/doc/refman/8.0/en/create-index.html#create-index-multi-valued.

[47] *MySQL Documentation, MVCC*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_mvcc.

[48] *MySQL Documentation, mysql Client Options*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/mysql-command-options.html#option_mysql_xml.

[49] *MySQL Documentation, MySQL Editions*. Accessed: 2020-04-28. URL: https://www.mysql.com/products/.

[50] *MySQL Documentation, MySQL Workbench*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/workbench.html.

[51] *MySQL Documentation, mysqldump — A Database Backup Program*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/mysqldump.html#option_mysqldump_xml.

[52] *MySQL Documentation, Nested-Loop Join Algorithms*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/nested-loop-joins.html.

[53] *MySQL Documentation, Optimizer Hints*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html.

[54] *MySQL Documentation, Optimizing InnoDB Queries*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/optimizing-innodb-queries.html.

[55] *MySQL Documentation, Optimizing Queries with EXPLAIN*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/using-explain.html.

[56] *MySQL Documentation, Optimizing Subqueries with Materialization*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/subquery-materialization.html.

[57] *MySQL Documentation, ORDER BY Optimization*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/order-by-optimization.html.

[58] *MySQL Documentation, Outer Join Optimization*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/outer-join-optimization.html.

[59] *MySQL Documentation, Outer Join Simplification*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/outer-join-simplification.html.

[60] *MySQL Documentation, Overview of Partitioning in MySQL*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/partitioning-overview.html.

[61] *MySQL Documentation, Partitioning*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/partitioning.html.

[62] *MySQL Documentation, Partitioning Types*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/partitioning-types.html.

[63] *MySQL Documentation, RANGE Partitioning*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/partitioning-range.html.

[64] *MySQL Documentation, SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Statements*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/savepoint.html.

[65] *MySQL Documentation, SET TRANSACTION Statement*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html#set-transaction-scope.

[66] *MySQL Documentation, Setting the Storage Engine*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/storage-engine-setting.html.

[67] *MySQL Documentation, Sorted Index Builds*. Accessed: 2020-05-15. URL: https://dev.mysql.com/doc/refman/8.0/en/sorted-index-builds.html.

[68] *MySQL Documentation, Spatial Indexes*. Accessed: 2020-05-15. URL: https://dev.mysql.com/doc/refman/8.0/en/create-index.html#create-index-spatial.

[69] *MySQL Documentation, Statements That Cause an Implicit Commit*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/implicit-commit.html.

[70] *MySQL Documentation, Subpartitioning*. Accessed: 2020-05-13. URL: https://dev.mysql.com/doc/refman/8.0/en/partitioning-subpartitions.html.

[71] *MySQL Documentation, Switchable Optimizations*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/switchable-optimizations.html.

[72] *MySQL Documentation, The Physical Structure of an InnoDB Index*. Accessed: 2020-05-15. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-physical-structure.html.

[73] *MySQL Documentation, Transaction Isolation Levels*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html.

[74] *MySQL Documentation, Trigger syntax and examples*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/trigger-syntax.html.

[75] *MySQL Documentation, Unique Indexes*. Accessed: 2020-05-15. URL: https://dev.mysql.com/doc/refman/8.0/en/create-index.html#create-index-unique.

[76] *MySQL Documentation, Using JDBC Statement Objects to Execute SQL*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-usagenotes-statements.html.

[77] *MySQL Documentation, Using Stored Routines*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/stored-routines.html.

[78] *MySQL Documentation, Using Triggers*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/triggers.html.

[79] *MySQL Documentation, Visual SQL Editor*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/workbench/en/wb-sql-editor.html.

[80] *MySQL Documentation, What is MySQL?* Accessed: 2020-04-28. URL: https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html.

[81] *MySQL Documentation, WHERE Clause Optimization*. Accessed: 2020-05-16. URL: https://dev.mysql.com/doc/refman/8.0/en/where-optimization.html.

[82] *MySQL Documentation, XML Functions*. Accessed: 2020-05-17. URL: https://dev.mysql.com/doc/refman/8.0/en/xml-functions.html.

[83] *Oracle Database Administrator's Guide, Deferred Constraint Checking*. Accessed: 2020-05-17. URL: https://docs.oracle.com/en/database/oracle/oracle-database/18/admin/managing-schema-objects.html#GUID-FDF6445E-D1E0-4305-B1D4-C5D4310CA1FB.

[84] *Oracle Database Administrator's Guide, Managing Indexes*. Accessed: 2020-05-15. URL: https://docs.oracle.com/en/database/oracle/oracle-database/18/admin/managing-indexes.html.

[85] *Oracle Database Concepts, Locks and Deadlocks*. Accessed: 2020-05-17. URL: https://docs.oracle.com/en/database/oracle/oracle-database/18/cncpt/data-concurrency-and-consistency.html#GUID-C1971E9B-849A-4634-9575-4F8FAD697750.

[86] *Oracle Partitioning*. Accessed: 2020-05-14. URL: https://www.oracle.com/database/technologies/partitioning.html.

[87] *SQL Tuning Guide, Join Methods*. Accessed: 2020-05-16. URL: https://docs.oracle.com/en/database/oracle/oracle-database/18/tgsql/joins.html#GUID-54F957FB-3568-499A-BCD2-B242BFFF913D.

[88] *SQL Tuning Guide, Optimizer Access Paths*. Accessed: 2020-05-16. URL: https://docs.oracle.com/en/database/oracle/oracle-database/18/tgsql/optimizer-access-paths.html#GUID-00711237-35D3-4CFC-A234-59B3EC53DCD1.