Show Submission Credentials

P3. MySQL Primer MySQL Primer

44 days 1 hours left

✔ Introduction to MySQL

✔ MySQL Tutorial

Introduction to MySQL

Introduction to MySQL

The Basics

The following video introduces the basic aspects of MySQL:

Let's take a closer look at databases and what a database schema is. A **database** holds the data as records organized in fields and cells. A **database schema** describes how the fields and cells are structured and organized and what types of relationships are mapped between these entities.

A database can have multiple tables that are related to each other. The schema defines the structure of the tables and the relations between them.

We will use a phone book as an example to demonstrate. A CMU phone book contains all the phone numbers of every CMU member, it includes last name, first name, and phone number.

MySQL provides the SELECT command to list the data, you can list the first 10 records of the table using the following command:

```
SELECT * FROM phone_book LIMIT 10;
```

To make the output clearer with fewer columns, you can specify the column names to select.

```
SELECT last_name, first_name, phone_number FROM phone_book LIMIT 10;
```

MySQL allows you to make queries from the database using keywords such as SELECT, WHERE and LIKE. For example, the SQL command to find the number of records that contain "Andrew" is:

```
SELECT * FROM phone_book WHERE first_name LIKE '%Andrew%';
```

You need to investigate the usage of `%` before and after "Andrew", and figure out whether the query above is case sensitive or not.

Data Definition Language (DDL) and Data Manipulation Language (DML)

Two major categories of SQL languages are data definition language (DDL) and data manipulation language (DML).

DDL statements are used to **define the structure** of the tables, e.g.

- `CREATE`
- `ALTER`
- `DROP`

DML statements are used to **manipulate the actual data** in tables. The four basic functions of persistent storage, Create, Read, Update, and Delete (a.k.a CRUD) are all DML statements with the following mapping:

- Create: `INSERT`
- Read: `SELECT`
- Update: `UPDATE`
- Delete: `DELETE`

Aggregate Functions

Aggregate functions (https://dev.mysql.com/doc/refman/5.7/en/aggregate-functions.html) allow you to perform calculations on a set of records and return a single value. The most common aggregate functions include `SUM`, `AVG`, `MAX`, `MIN`, and `COUNT`.

For example, the query to get the total number of entries in the phone book is:

```
SELECT COUNT(*) FROM phone_book;
```

Aggregate functions are often used with the MySQL `GROUP BY` keyword to perform calculations on each subgroup and return a single value for each subgroup. The MySQL `GROUP BY` keyword is used with the `SELECT` statement to group rows into subgroups by one or more columns or expressions. It is extremely useful when several records belong to a category and other records in the same table belong to another category and you want to compare between different categories rather than a single record.

The following statement illustrates the MySQL `GROUP BY` keyword syntax:

```
SELECT c1, c2, ... cn, aggregate_function(expression)
FROM table
WHERE where_conditions
GROUP BY c1, c2, ... cn;
```

JOIN

The MySQL `JOIN` keyword is used to query data from two or more related tables. In MySQL, `JOIN`, `CROSS JOIN`, and `INNER JOIN` are syntactic equivalents and they can replace each other. The following illustrates a sample JOIN syntax:

```
SELECT c1,c2,....cn
FROM table1 INNER JOIN table2
ON conditional_expr # e.g. table1.id = table2.id
```

Note: In the standard SQL, they are not equivalent. The following diagram shows the difference between `FULL JOIN`, `INNER JOIN` and `CROSS JOIN`.

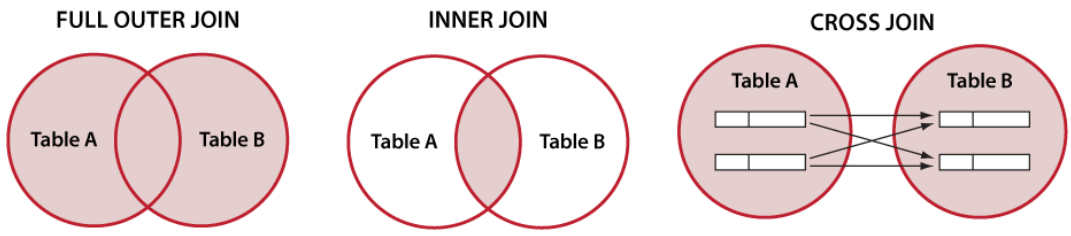

Figure 1: Full Join v.s. Inner Join v.s. Cross Join

NATURAL JOIN

In MySQL, the `NATURAL JOIN` is such a join that performs the same task as an `INNER JOIN` or `LEFT JOIN`, in which the ON or USING clause refers to all columns that the tables to be joined have in common. `NATURAL JOIN` is structured in such a way that, columns with the same name of associate tables will appear once only. The following illustrates a sample JOIN syntax:

```
SELECT c1,c2,....cn
FROM table1 NATURAL JOIN table2
```

OUTER JOIN

`OUTER JOIN` identifies rows without a match in the joined table. Outer joins combine two or more tables in a way that some columns may have NULL values. MySQL separates `OUTER JOIN` into `LEFT JOIN` or `RIGHT JOIN` depending on which table provides the unmatched data.

A `LEFT JOIN` returns all rows from the left table (TableA) with the matching rows from the right table (TableB) or null – if there is no match in the right table.

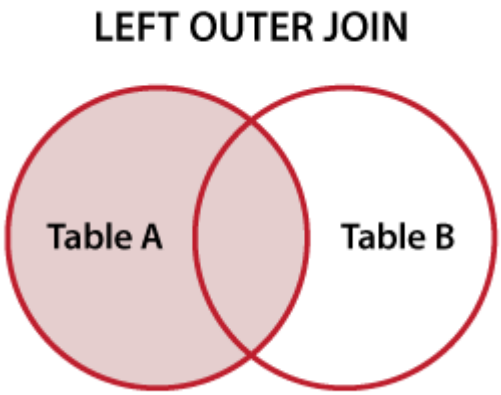

Figure 2: Left Join Diagram

A `RIGHT JOIN` returns all rows from the right table (TableB) with the matching rows from the left table (TableA) or null – if there is no match in the left table.

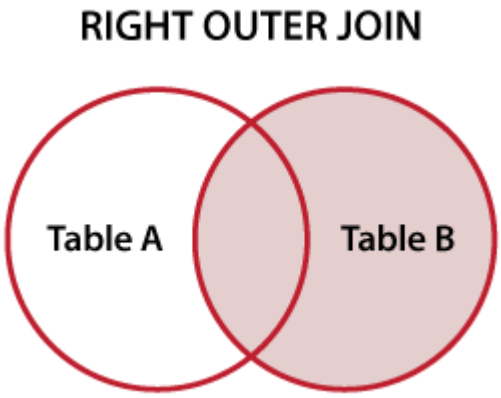

Figure 3: Right Join Diagram

MySQL Indexing

A database index helps speed up the retrieval of data from tables. When you query data from a table, MySQL checks if there exist any applicable indexes on the queried columns. Then MySQL uses the indexes to search the physical storage of the corresponding rows instead of scanning the whole table.

The Mechanism of Indexing

MySQL supports 2 types of indexes: B-Tree and Hash Indexes.

Schema design is based on the structure of the data; index design is based on the data as well as queries. You can build effective indexes only if you are aware of the queries you need. We will discuss different types of queries and appropriate indexes.

Suppose we would like the entries to be sorted by the last name and then the first name. In MySQL syntax, this would look like the following index:

```
CREATE INDEX phone_index ON phone_book (last_name, first_name)
```

Index and Equality Comparison Query

If you want to search a phone number by the last name "Carnegie" and first name "Andrew". The index will narrow down the search quickly. You will look into the section where the last name starts with "C", locate "Carnegie", and search "A" to get "Andrew", and you are effectively performing a binary search.

With this phone book, you can easily find all the entries with the last name "Carnegie" without reading the whole phone book, which can be interpreted as:

```
# an index on phone_book (last_name, first_name) will help with:
SELECT * from phone_book WHERE last_name = "Carnegie";
```

However, what if you need to find all the entries with the first name "Andrew"? The only way is to scan the whole book. Anyone's first name can be "Andrew" and the distribution is not predictable with the design of the phone book.

```
# an index on phone_book (last_name, first_name) will NOT help with:
SELECT * from phone_book WHERE first_name = "Andrew";
```

To speed up the search by the first name, another index is needed which groups and sorts the entries by the first name, and then last name.

Index and Range Query

Similar to equality comparison queries, range queries can be faster with indexes.

For example, the following query can benefit from an index on `phone_book (last_name, first_name)`:

```
SELECT * from phone_book WHERE last_name LIKE "C%";
```

Entries with the last name starting with "C" are grouped and there is no need to scan the whole book, and `C%` is effectively a range query `WHERE last_name >= 'C' AND last_name < 'D'`.

Single-Column Index and Multiple-Column Index

For a search query that refers to multiple columns, there are two indexing approaches: create one index per column, or create one single multiple-column index (a.k.a. composite index).

```
# two single-column indexes, one on last_name and the other on first_name,
# will help with:
SELECT * from phone_book WHERE last_name = "Carnegie" AND first_name = "Andrew";
# but a composite index on (last_name, first_name) is more effective

# two single-column indexes, one on last_name and the other on first_name,
# are as good as it gets:
SELECT * from phone_book WHERE last_name = "Carnegie" OR first_name = "Andrew";
# but a composite index on (last_name, first_name) will NOT work
# because the distribution of first_name = "Andrew" is not predictable.
```

For composite indexes, **the column order matters**. If you have a query to select the rows `WHERE A > 10 and B = 10`, an index on `(B, A)` is much more effective than one on `(A, B)`.

The Mistakes of Indexing

When the workload is read-heavy, indexing is powerful to improve performance. However, you should not overuse indexing everywhere because of the following trade-offs.

Space: Each index takes extra storage space on the disk.

Slow down writes: Each index needs to be modified during writes (`INSERT`, `UPDATE`, `DELETE`).

The Auditing of Indexing and EXPLAIN statements

MySQL provides the `EXPLAIN` statements so that you can predict the performance of a query **without execution**. An `EXPLAIN` statement will display information from the optimizer about the statement execution plan, and you can predict performance with the information. For more details about the output of the `EXPLAIN` statement, please refer to this link (https://dev.mysql.com/doc/refman/8.0/en/explain-output.html).

Predicted Scanned Rows

In the table of the `EXPLAIN` result, the `rows` column indicates the number of rows MySQL predicts that it must examine to execute the query. If you have a proper index given a query, you will notice that the `rows` will differ a lot with or without the index. If not, the index will not work with the query.

1. The value of `rows` is a prediction and can be different from the actual rows to scan during the execution of the query.
2. `EXPLAIN` may return different predictions of scanned rows on effectively equivalent queries. MySQL optimizer is capable to optimize most queries, but some effective equivalence is based on the specific data instead of the schema or indexes, and the optimizer cannot predict or optimize the query ahead.

Another column with much valuable information is the `Extra` column. If you want to make your queries as fast as possible, watch out for Extra column values `Using filesort` and `Using temporary`. They are signals of unoptimized queries. By contrast, the following messages generally indicate the indexes are effective: `Using index`, `Select tables optimized away`.

If you create an index, but it cannot speed up the query, it means either the index cannot match the need of the query, or the query is not designed properly and cannot be optimized by MySQL. We expect you to figure out solutions when you encounter this issue before you ask on Piazza.

The Design of Indexing

You can start the design of indexes by the following points:

- What tables and columns do you need to query?
- What `JOINS` do you need to perform?
- What `GROUP BY`'s and `ORDER BY`'s do you need?

Later on in the task, you will use the practice above to design proper indexes for the queries.

MySQL Storage Engine

`MyISAM` and `InnoDB` are the most commonly used storage engines in MySQL.

MyISAM

The `MyISAM` engine uses `B+Tree` as the index structure. The leaf node stores the **index**, which is the primary key in our following example, and the **address of the real data records**. The index of `MyISAM` is also called `non-clustered` index since the index and data are detached. Below is a schematic diagram of the `MyISAM` index:

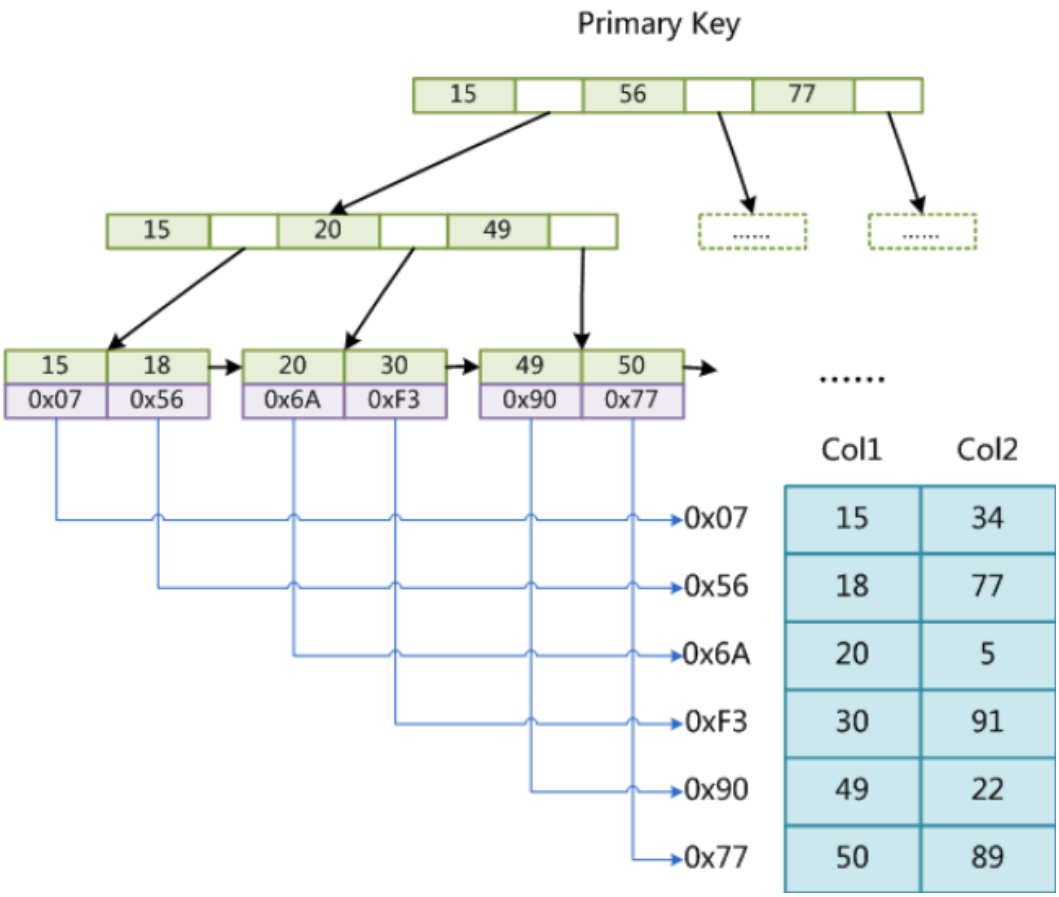

Figure 4: MyISAM Index Structure Diagram

There's no functional difference between the primary key (the attribute that uniquely identifies a row in a table) and secondary keys (fields that are non-identifying but can be used to find rows in a table) in MyISAM since a secondary index's leaf nodes always point directly to the row in the table.

InnoDB

The `InnoDB` engine also uses `B+Tree` as the index structure. The leaf node stores both the **index** and the actual **data records**. The index of `InnoDB` is also called **clustered index** since index and data are stored together. Below is a schematic diagram of the `InnoDB` index:

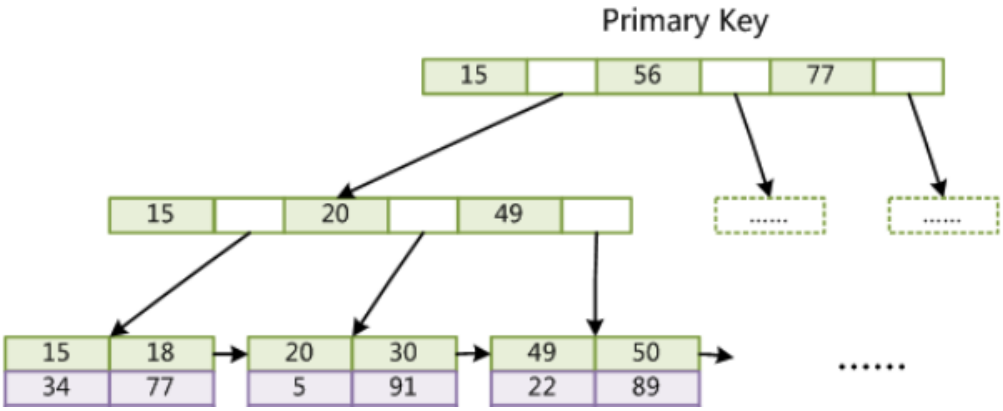

Figure 5: InnoDB Index Structure Diagram

All indexes other than the clustered index are known as **secondary indexes**. In InnoDB, each record in a secondary index contains the primary key columns for the row, as well as the columns specified for the secondary index. InnoDB uses this primary key value to search for the row in the clustered index. If the primary key is long, the secondary indexes use more space, so it is advantageous to have a short primary key.

Comparison between MyISAM and InnoDB

Characteristics	MyISAM	InnoDB
Cache mechanism	Only cache the indexes in MySQL's memory. The data is only cached by the underlying operating system's file system caches.	Maintain its buffer pool that caches both table data and index data, speeding up lookups for queries on columns which are not indexed due to table data caching.
Locking mechanism	table-level locking	row-level locking
Foreign key	No	Yes
Transaction	No	Yes
Read and write operation	Good at reading but bad at writing.	Writing is optimized by an automatic mechanism called change buffering. Allow concurrent read and write access to the same table.
Table level data	Save data as table level so it is easy to read out the saved row number.	Does not save data as table level.
Data Order	Data is in insertion order.	Physically ordered by the primary key.

MySQL Tutorial

MySQL Tutorial

Provision the VM instance

1. First, make sure you are authenticated with Azure.

```
az login
```

2. **Warning**: As you may have multiple subscriptions, make sure you are using the subscription for this project.

```
az account set --subscription $SUBSCRIPTION_ID
```

3. Create a new resource group. Note that you will reuse bash variables in the commands below.

```
RESOURCE_GROUP_NAME="mysql-primer"
az group create --name $RESOURCE_GROUP_NAME --location eastus
```

4. Set an appropriate tag with the tag value being used in the **currently ongoing project**.

```
TAG=<tag_value>
```

5. Create a `Standard_F1s` VM instance from the Ubuntu image with the proper tag.

```
az vm create \
    --resource-group $RESOURCE_GROUP_NAME \
    --name mysqlServer \
    --image UbuntuLTS \
    --admin-username clouduser \
    --size Standard_F1s \
    --tags "project=${TAG}" \
    --admin-password ${PASSWORD}
```

6. To enable the remote MySQL access, we need to open the port `3306` for our VM.

```
az vm open-port \
    --port 3306 \
    --resource-group $RESOURCE_GROUP_NAME \
    --name mysqlServer
```

7. You can find the `publicIpAddress` from the result of `az vm create`. SSH into the VM as `clouduser`.

```
ssh clouduser@PUBLIC_IP
```

Install MySQL

1. Install the MySQL server.

```
sudo apt update
sudo apt install mysql-server
```

2. log in to MySQL as the root user.

```
sudo mysql
```

3. You can use the following commands to create a new user whose name is `clouduser` and password is `dbroot`.

```
# set the working database
> use mysql;

# create a user for local access
> GRANT ALL PRIVILEGES ON *.* TO 'clouduser'@'localhost' IDENTIFIED BY 'db
root' WITH GRANT OPTION;

# create a user for remote access
> GRANT ALL PRIVILEGES ON *.* TO 'clouduser'@'%' IDENTIFIED BY 'dbroot' WI
TH GRANT OPTION;
```

4. After creating the user, you can use the SELECT command to view all of the users. Then exit the MySQL shell.

```
> SELECT host,user FROM user;
> exit;
```

5. Open the configuration file and comment the line `bind-address = 127.0.0.1`.

```
sudo vim /etc/mysql/mysql.conf.d/mysqld.cnf
```

6. Restart MySQL.

```
sudo service mysql restart
```

7. To start a MySQL CLI client and connect to the running MySQL server on your instance, use the following command.

```
mysql -u clouduser -pdbroot
```

You can also create the connection remotely using the following command.

```
mysql -u clouduser -pdbroot -h PUBLIC_IP
```

Load Data

1. The dataset we use is Hourly Weather Surface - Brazil (Southeast region) (https://www.kaggle.com/PROPPG-PPG/hourly-weather-surface-brazil-southeast-region/data#), which covers hourly weather data from 122 weather stations of southeast Brazil. This dataset is published by Kaggle and you can open the link

(https://www.kaggle.com/PROPPG-PPG/hourly-weather-surface-brazil-southeast-region/data#) to know the structure of this dataset and the meaning of every column. You can use the following bash commands to download the data to your VM instance.

```
# Download dataset
wget https://clouddeveloper.blob.core.windows.net/datasets/mysql-primer/weather_brazil.zip

# Download unzip
sudo apt install unzip

# Decompress weather_brazil.zip
unzip weather_brazil.zip
```

2. Download the SQL script `myisam.sql` and `innodb.sql` . Then create the tables `weather_myisam` and `weather_innodb` , whose primary key consists of `wsid` (**weather state id**) and `mdct` (**observation data time**).

```
wget https://clouddeveloper.blob.core.windows.net/assets/cloud-storage/primers/myisam.sql
wget https://clouddeveloper.blob.core.windows.net/assets/cloud-storage/primers/innodb.sql

mysql -uclouduser -pdbroot --local-infile=1 < myisam.sql
mysql -uclouduser -pdbroot --local-infile=1 < innodb.sql
```

MySQL Query

Login to MySQL and choose the working database.

```
mysql -u clouduser -pdbroot

> use kaggle_db;
```

MyISAM saves data as table level so you can easily read the saved row number while **InnoDB** does not save data as table level so counting the number of rows needs to scan the whole table. You can run the following queries to see that `SELECT COUNT(*)` runs much faster on `MyISAM` table than `InnoDB` table.

```
> select count(*) from weather_myisam;
> select count(*) from weather_innodb;
```

The following queries select records that satisfy the condition " `wsid > 178` ", extract the column `wsid` from the table, and count the record number. When we run the following queries, we can find that **MyISAM**'s readabilities outshine **InnoDB**. It is because locking the entire table is quicker than figuring out which rows are locked in the table. The more information in the table, the more time it takes InnoDB to figure out which ones are not accessible.

```
> select count(wsid) from weather_myisam where wsid > 178;
> select count(wsid) from weather_innodb where wsid > 178;
```

In the previous queries, the column `wsid` has been indexed, so both MyISAM and InnoDB can quickly get it from the memory. Now let's try the following queries which extract the column `wsnm` (**station name**). Since `wsnm` isn't indexed, MyISAM needs to get data from the storage while InnoDB can directly get it from the memory and avoid a lot of I/O operations. Thus as we can see, InnoDB is much faster than MyISAM for this situation.

```
> select count(wsnm) from weather_myisam where wsid > 178;
> select count(wsnm) from weather_innodb where wsid > 178;
```

Delete the Azure Resource Group

Azure Resource Group enables you to organize your resources and arrange resources with the same lifecycle in the same resource group.

By using Azure resource group, you can terminate all the resources you used in this project with one single command.

```
az group delete --name $RESOURCE_GROUP_NAME
```