# Database Technologies

DAC 2021

# Content

- Indexes
- Benefit of Indexes
- Type of Indexes
- Temporary Tables
- ACID Properties
- Concept of Database Instance and Schema
- MySQL Storage Engines (InnoDB, MyISAM and others)

# Index

- Technique to speed up  the queries
- Can be created using one or more columns
- While creating index, it should be taken into consideration which all columns will be used to make SQL queries and create one or more indexes on those columns.
- Indexes are also a type of tables, which keep primary key or index field and a pointer to each record into the actual table.
- The users cannot see the indexes, they are just used to speed up queries and will be used by the Database Search Engine to locate records very fast.

# Index - Need

**Scenario :**

A contact book that contains names and mobile numbers of the user.

To find the mobile number of “Kelvin”. If the contact book is an unordered format, sequential scan needs to be performed until the desired name is found. This type of searching name is known as sequential searching.

# Index - Need

```
mysql> SELECT     EmpName, DeptName, dept.DeptCode FROM emp,dept  WHERE emp.Dept
Code = dept.DeptCode;
+---------+----------+----------+
| EmpName | DeptName | DeptCode |
+---------+----------+----------+
| Reddy   | Accounts | ACCT     |
+---------+----------+----------+
1 row in set (0.05 sec)
```

# Benefit of Indexes

- Finding a row for a particular employee and corresponding department requires the full table scan requires examination of each row in the table to see whether it matches the desired value.
- This involves a full table scan, which is slow, as well as tremendously inefficient if the table is large but contains only a few rows that match the search criteria

# Benefit of Indexes

- Indexes are used to speed up searches for rows matching terms of a WHERE clause
- Rows that match rows in other tables when performing joins.
- For queries that use the MIN() or MAX() functions, the smallest or largest value in an indexed column can be found quickly without examining every row.
- Indexes  can be used to perform sorting and grouping operations quickly for ORDER BY and GROUP BY clauses.

# Choosing Indexes

- Index columns that you use for searching, sorting, or grouping, not columns you select for output
  - The columns that appear in WHERE clause, columns named in join clauses, or columns that appear in ORDER BY or GROUP BY clauses.
  - Columns that appear only in the output column list following the SELECT keyword are not good candidates:

# Choosing Indexes

```
SELECT
    col_a                          ← not a candidate
FROM
    tbl1 LEFT JOIN tbl2
    ON tbl1.col_b = tbl2.col_c     ← candidates
WHERE
    col_d = expr;                  ← a candidate
```

# Choosing Indexes

- Consider column cardinality.
  - Indexes work best for columns that have a high cardinality relative to the number of rows in the table (that is, columns that have many unique values and few duplicates).
  - For a column that contains many different age values, an index readily differentiates rows.
  - For a column that is used to record gender and contains only the two values 'M' and 'F', an index will not help.
    - Under these circumstances, the index might never be used at all, because the query optimizer generally skips an index in favor of a full table scan if it determines that a value occurs in a large percentage of a table’s rows.

# Choosing Indexes

- **Index short values.**
  - Use smaller data types when possible.
  - Don’t use a BIGINT column if a MEDIUMINT is large enough to hold the values you need to store, and don’t use CHAR(100) if none of your values are longer than 25 characters.
  - Shorter values can be compared more quickly, so index lookups are faster.
  - Smaller values result in smaller indexes that require less disk I/O.
  - With shorter key values, index blocks in the key cache hold more key values.

# Choosing Indexes

- **Take advantage of leftmost prefixes.**
- A composite index serves as several indexes because any leftmost set of columns in the index can be used to match rows.
- Such a set is called a “leftmost prefix.”
- Suppose that you have a table with a composite index on columns named state, city, and zip.
- Rows in the index are sorted in state/city/zip order, so they’re automatically sorted in state/city order and in state order as well.
- This means that MySQL can take advantage of the index even if you specify only state values in a query, or only state and city values.
- Thus, the index can be used to search the following combinations of columns: state, city, zip
- state, city
- state
- if you search by city or by zip, the index isn’t used.
- If you’re searching for a given state and a particular ZIP code, the index can’t be used for the combination of values

# Choosing Indexes

- **Don’t over-index.**
- Every additional index takes extra disk space and hurts performance of write operations, as has already been mentioned.
- Indexes must be updated and possibly reorganized when you modify the contents of your tables, and the more indexes you have, the longer this takes.
- If you have an index that is rarely or never used, you’ll slow down table modifications unnecessarily.
- In addition, MySQL considers indexes when generating an execution plan for retrievals.
- Creating extra indexes creates more work for the query optimizer.
- It’s also possible (if unlikely) that MySQL will fail to choose the best index to use when you have too many indexes.
- If you’re thinking about adding an index to a table that is already indexed, consider whether the index you’re considering adding is a leftmost prefix of an existing multiple-column index.
  - For example, if you already have an index on state, city, and zip, there is no point in adding an index on state.

# Index - Issues

- The INSERT and UPDATE statements take more time on tables having indexes, whereas the SELECT statements become fast on those tables.
- The reason is that while doing insert or update, a database needs to insert or update the index values as well.

# Index- Type

- **Unique index**
  - This disallows duplicate values.
  - For a single-column index, this ensures that the column contains no duplicate values.
  - For a multiple-column (composite) index, it ensures that no combination of values in the columns is duplicated among the rows of the table.
- **Non-unique index**
  - This gives you indexing benefits but allows duplicates.
- **FULLTEXT index**
  - Used for performing full-text searches.
  - This index type is supported only for MyISAM tables.

# Index- Type

- **SPATIAL index :**
  - These can be used only with MyISAM tables for the spatial data types(storing the coordinates)
- **HASH index**
  - This is the default index type for MEMORY tables, although you can override the default to create BTREE indexes instead.

# Index command

- CREATE INDEX index_name ON tbl_name (index_columns);
- CREATE UNIQUE INDEX index_name ON tbl_name (index_columns);
- To drop an index, use either a DROP INDEX or an ALTER TABLE statement.
  - DROP INDEX index_name  ON tbl_name;
  - ALTER TABLE tbl_name DROP INDEX index_name;
- Show index
  - show index from emp \G;
- INDEX, you must name the index to be dropped:
- DROP INDEX index_name ON tbl_name;
- multiple indexes can not be created with a single statement.

# Index command

- CREATE TABLE tbl_name
  (
  ... column definitions ...
  INDEX index_name (index_columns),
  UNIQUE index_name (index_columns),
  PRIMARY KEY (index_columns),
  FULLTEXT index_name (index_columns),
  SPATIAL index_name (index_columns),
  ...
  );
- If  a column is dropped that is a part of an index, MySQL removes the column from the index as well.
- If you drop all columns that make up an index, MySQL drops the entire index

# Temporary Tables

- If you add the TEMPORARY keyword to a table-creation statement, the server creates a temporary table that disappears automatically when your connection to the server terminates
- DROP TABLE statement is not needed to get rid of the table
- A TEMPORARY table is visible only to the client that creates the table.
  - Different clients can each create a TEMPORARY table with the same name and without conflict because each client sees only the table that it created.

# Temporary Tables

- The name of a TEMPORARY table can be the same as that of an existing permanent table.
  - This is not an error, nor does the existing permanent table get clobbered.
  - Instead, the permanent table becomes hidden (inaccessible) to the client that creates the TEMPORARY table while the TEMPORARY table exists.
  - Suppose that you create a TEMPORARY table named member in the sampdb database.
  - The original member table becomes hidden, and references to member refer to the TEMPORARY table.
  - If you issue a DROP TABLE member statement, the TEMPORARY table is removed and the original member table “reappears.”
  - If you disconnect from the server without dropping the TEMPORARY table, the server automatically drops it for you.

# Temporary Tables

- CREATE TEMPORARY TABLE tbl_name;
- DROP TEMPORARY TABLE tbl_name

# MySQL Storage Engines

- MySQL supports multiple storage engines/ table handlers
- Each storage engine implements tables that have a specific set of properties or characteristics.

# MySQL Storage Engines

| Storage Engine | Description |
|---|---|
| ARCHIVE | Archival storage (no modification of rows after insertion) |
| BLACKHOLE | Engine that discards writes and returns empty reads |
| CSV | Storage in comma-separated values format |
| EXAMPLE | Example ("stub") storage engine |
| Falcon | Transactional engine |
| FEDERATED | Engine for accessing remote tables |
| InnoDB | Transactional engine with foreign keys |
| MEMORY | In-memory tables |
| MERGE | Manages collections of MyISAM tables |
| MyISAM | The default storage engine |
| NDB | The engine for MySQL Cluster |

# MySQL Storage Engines

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
      Engine: ARCHIVE
     Support: YES
     Comment: Archive storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 2. row ***************************
      Engine: BLACKHOLE
     Support: YES
     Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 3. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 4. row ***************************
      Engine: FEDERATED
     Support: NO
     Comment: Federated MySQL storage engine
Transactions: NULL
          XA: NULL
  Savepoints: NULL
```

# MySQL Storage Engines

- The value in the Support column is YES or NO to indicate that the engine is or is not available,
- DISABLED if the engine is present but turned off,
- DEFAULT for the storage engine that the server uses by default.
- The engine designated as DEFAULT should be considered available.
- The Transactions column indicates whether an engine supports transactions.
- XA and Savepoints indicate whether an engine supports distributed transactions and partial transaction rollback.

# InnoDB Storage Engine

- Transaction-safe tables with commit and rollback.
- Savepoints can be created to enable partial rollback.
- Automatic recovery after a crash.
- Foreign key and referential integrity support, including cascaded delete and update.
- Row-level locking and multi-versioning for good concurrency performance under query mix conditions that include both retrievals and updates.

# MyISAM Storage Engine

- MyISAM provides key compression.
- Uses compression when storing runs of successive similar string index values.
- MyISAM provides more features for AUTO_INCREMENT columns than do other storage engines.
- MyISAM tables also have a flag indicating whether a table was closed properly when last used.
- If the server shuts down abnormally or the machine crashes, the flags can be used to detect tables that need to be checked.