# Indexes

- Indexes help:
  - find the rows matching a WHERE clause quickly.
  - eliminate rows from consideration.
  - retrieve rows from other tables when performing joins.
  - find the MIN() or MAX() value for a specific indexed column.
  - sort or group a table (under certain conditions).
  - optimize queries using only indexes without consulting the data rows.

# Indexes

- MySQL automatically creates an index for primary key, foreign key, and unique constraints.

```
CREATE [UNIQUE] INDEX index_name
    ON [dbname.]table_name (column_name_1 [ASC|DESC]
                            [, column_name_2 [ASC|DESC]]…)
```

- Best practice is to create any additional index(es) when creating the table.

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
       {INDEX | KEY} [index_name] [index_type] (key_part,…) [index_option] ...
```

# MySQL Data Types

- String
- Numeric
- Date & Time

# String

| | |
|---|---|
| CHAR(size) | A FIXED length string (can contain letters, numbers, and special characters). The *size* parameter specifies the column length in characters - can be from 0 to 255. Default is 1 |
| VARCHAR(size) | A VARIABLE length string (can contain letters, numbers, and special characters). The *size* parameter specifies the maximum column length in characters - can be from 0 to 65535 |
| BINARY(size) | Equal to CHAR(), but stores binary byte strings. The *size* parameter specifies the column length in bytes. Default is 1 |
| VARBINARY(size) | Equal to VARCHAR(), but stores binary byte strings. The *size* parameter specifies the maximum column length in bytes. |
| TINYBLOB | For BLOBs (Binary Large OBjects). Max length: 255 bytes |
| TINYTEXT | Holds a string with a maximum length of 255 characters |
| TEXT(size) | Holds a string with a maximum length of 65,535 bytes |

| | |
|---|---|
| BLOB(size) | For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data |
| MEDIUMTEXT | Holds a string with a maximum length of 16,777,215 characters |
| MEDIUMBLOB | For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data |
| LONGTEXT | Holds a string with a maximum length of 4,294,967,295 characters |
| LONGBLOB | For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data |
| ENUM(val1, val2, val3, ...) | A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them |
| SET(val1, val2, val3, ...) | A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list |

From W3Schools.com

# Numeric

| | |
|---|---|
| BIT(*size*) | A bit-value type. The number of bits per value is specified in *size*. The *size* parameter can hold a value from 1 to 64. The default value for *size* is 1. |
| TINYINT(*size*) | A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The *size* parameter specifies the maximum display width (which is 255) |
| BOOL | Zero is considered as false, nonzero values are considered as true. |
| BOOLEAN | Equal to BOOL |
| SMALLINT(*size*) | A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The *size* parameter specifies the maximum display width (which is 255) |
| MEDIUMINT(*size*) | A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The *size* parameter specifies the maximum display width (which is 255) |
| INT(*size*) | A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The *size* parameter specifies the maximum display width (which is 255) |

| | |
|---|---|
| INTEGER(*size*) | Equal to INT(size) |
| BIGINT(*size*) | A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The *size* parameter specifies the maximum display width (which is 255) |
| FLOAT(*size*, *d*) | A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions |
| FLOAT(*p*) | A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE() |
| DOUBLE(*size*, *d*) | A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter |
| DOUBLE PRECISION(*size*, *d*) | |

From W3Schools.com

# Numeric

| | |
|---|---|
| DECIMAL(*size*, *d*) | An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0. |
| DEC(*size*, *d*) | Equal to DECIMAL(size,d) |

From W3Schools.com

# Date & Time

| | |
|---|---|
| DATE | A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31' |
| DATETIME(*fsp*) | A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time |
| TIMESTAMP(*fsp*) | A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition |
| TIME(*fsp*) | A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59' |
| YEAR | A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000.<br>MySQL 8.0 does not support year in two-digit format. |

From W3Schools.com

# Creating Table Structures (1 of 4)

- CREATE TABLE command
  - **CREATE TABLE** is an SQL command that creates a table's structures using the characteristics and attributes given

```
CREATE TABLE tablename (
column1  data type      [constraint] ,
column2  data type      [constraint] ,
PRIMARY KEY (column1    [, column2, …]) ,
FOREIGN KEY (column1    [, column2, …]) REFERENCES tablename] ,
CONSTRAINT  constraint  );
```

# Creating Table Structures (2 of 4)

- SQL Constraints
  - The FOREIGN KEY constraint ensures that you cannot delete a vendor from the VENDOR table if at least one product row references that vendor
  - The NOT NULL constraint ensures that a column does not accept nulls
  - The UNIQUE constraint ensures that all values in a column are unique
  - The DEFAULT constraint assigns a value to an attribute when a new row is added to a table
  - The CHECK constraint is used to validate data when an attribute value is entered

# Creating Table Structures (3 of 4)

- Create a Table with a SELECT Statement
  - SQL provides a way to rapidly create a new table based on selected columns and rows of an existing table using a subquery
  - All of the data rows returned by the SELECT statement are copied automatically
- SQL Indexes
  - **`CREATE INDEX`** improves the efficiency of searches and avoids duplicate column values
  - **`DROP INDEX`** is an SQL command used to delete database objects such as tables, views, indexes, and users

# Creating Table Structures (4 of 4)

Create index, duplicates allowed

```
CREATE INDEX index_name
  ON table_name (column1, column2, …);
```

Create index, duplicates disallowed

```
CREATE UNIQUE INDEX index_name
  ON table_name (column1, column2, …);
```

Example:

```
CREATE INDEX idx_pname
  ON Persons (LastName, FirstName);
```

Dropping an index

```
ALTER TABLE table_name

  DROP INDEX index_name;
```

From W3Schools.com

# Altering Table Structures (1 of 3)

- All changes in the table structure are made by using the **ALTER TABLE** command followed by a keyword that produces the specific change you want to make
  - The following three options are available: `ADD`, `MODIFY`, and `DROP`
- Changing a Column's Data Type
- Changing a Column's Data Characteristics
  - If the column to be changed already contains data, you can make changes in the column's characteristics if those changes do not alter the data type *(why?)*
- Adding a Column
  - You can alter an existing table by adding one or more columns
  - Be careful not to include the `NOT NULL` clause for the new column *(why?)*

# Altering Table Structures (2 of 3)

- Adding `PRIMARY KEY, FOREIGN KEY`, and `CHECK` constraints
  - Primary key syntax is as follows:

```
ALTER TABLE Part
   ADD PRIMARY KEY (Part_Code);
```

  - Foreign key syntax is as follows:

```
ALTER TABLE PART
   ADD FOREIGN KEY (V_Code) REFERENCES Vendor;
```

  - Check constraint syntax is as follows:

```
ALTER TABLE PART
   ADD CHECK (Part_Price >= 0);
```

# Altering Table Structures (3 of 3)

- Dropping a column from a table

```
ALTER TABLE VENDOR
    DROP COLUMN V_ORDER;
```

- Deleting a table from the Database
  - **DROP TABLE** is an SQL command used to remove database tables

```
DROP TABLE PART;
```

# Data Manipulation Commands (1 of 4)

- Adding table rows
  - **`INSERT`** is a SQL command that allows the insertion of one or more data rows into a table using a subquery
  - Syntax:

```
INSERT INTO tablename VALUES (value1, value2, …, valuen)
```

  - To insert rows with null attributes, use a `NULL` entry

# Data Manipulation Commands (2 of 4)

- Inserting table rows with a SELECT subquery
  - Using a subquery with the INSERT command, it is possible to add multiple rows to a table, using another table as the source, at the same time

```
INSERT INTO   target_tablename[(target_columnlist)]
SELECT source_columnlist
  FROM source_tablename;
```

- Saving table changes right now!

```
COMMIT
```

# Data Manipulation Commands (3 of 4)

- **UPDATE** command is used to modify data in a table

```
UPDATE   tablename
    SET  columnname = expression [, columnname =  expression]
 [WHERE  conditionlist ];
```

- **DELETE** command is used to delete table rows

```
DELETE FROM tablename
[WHERE conditionlist ];
```

# Data Manipulation Commands (4 of 4)

- Restoring Table Contents
  - The ROLLBACK command is used to restore the database table contents to the condition that existed after the last COMMIT statements;

```
ROLLBACK;
```

# Auto Increment

- MySQL uses the AUTO_INCREMENT property during table creation to indicate that values for an attribute should be generated in the same fashion
  - Only one column in a table can have AUTO_INCREMENT specified, **and** that column must also be defined as the primary key of the table!

# Procedural SQL

- Persistent stored module (PSM) is a block of code that contains standard SQL statements and procedural extensions that is stored and executed at the DBMS server
- Procedural SQL is an extension of SQL that adds procedural programming capabilities, such as variables and logical flow control, to SQL and is designed to run inside the database
  - The procedural code is executed as a unit by the DBMS when it is invoked by the end user
- End users can use procedural SQL to create the following:
  - Stored procedures
  - Triggers
  - Procedural SQL functions

# Stored Procedures

- A **stored procedure** is a named collection of procedural and SQL statements
  - They are stored in the database
  - A major advantage of stored procedures is that they can be used to encapsulate and represent business transactions
- Using stored procedures offers the following advantages:
  - Stored procedures substantially reduce network traffic and increase performance
  - Stored procedures help reduce code duplication by means of code isolation and code sharing

# Variables

```
DROP PROCEDURE IF EXISTS temp_proc;
delimiter $$
CREATE PROCEDURE TEMP_PROC()
BEGIN
    DECLARE MYNUM1 NUMERIC(1) DEFAULT 5;
    DECLARE MYNUM2 NUMERIC(1);
    DECLARE TOTAL INT;
    DECLARE MESSAGE VARCHAR(15) DEFAULT 'The total is ';

    SET MYNUM2 = 6;
    SET TOTAL = MYNUM1 * MYNUM2;
    SELECT CONCAT(MESSAGE, TOTAL) AS 'Result';
END;
$$
delimiter ;

call temp_proc();
```

# Conditional Execution

```mysql
DROP PROCEDURE IF EXISTS temp_proc;
delimiter $$
CREATE PROCEDURE TEMP_PROC()
BEGIN
    DECLARE MYNUM1 INT DEFAULT 50;
    DECLARE MYNUM2 INT DEFAULT 10;
    DECLARE TOTAL INT;
    DECLARE MESSAGE VARCHAR(50);
    SET TOTAL = MYNUM1 * MYNUM2;
    IF TOTAL < 100 THEN
        SET MESSAGE = 'The total is small: ';
    ELSEIF TOTAL >= 100 AND TOTAL <= 500 THEN
        SET MESSAGE = 'The total is medium: ';
    ELSEIF TOTAL BETWEEN 500 AND 1000 THEN
        SET MESSAGE = 'The total is big: ';
    ELSE
        SET MESSAGE = 'The total is huge: ';
    END IF;
    SELECT CONCAT(MESSAGE, TOTAL) AS 'Result';
END;
$$
delimiter ;
call temp_proc();
```

# Iteration or Looping

```
DROP PROCEDURE IF EXISTS temp_proc;
delimiter $$
CREATE PROCEDURE TEMP_PROC()
BEGIN
    DECLARE MYNUM INT DEFAULT 1;
    DECLARE RESULT VARCHAR(100) DEFAULT MYNUM;
    COUNTER: LOOP
    SET MYNUM = MYNUM + 1;
    IF MYNUM > 20 THEN
        LEAVE COUNTER;
    END IF;
    SET RESULT = CONCAT(RESULT, ', ', MYNUM);
    END LOOP;
    SELECT RESULT;
END;
$$
delimiter ;
call temp_proc();
```

# Iteration or Looping

```
DROP PROCEDURE IF EXISTS temp_proc;
delimiter $$
CREATE PROCEDURE TEMP_PROC()
BEGIN
    DECLARE MYNUM INT DEFAULT 1;
    DECLARE RESULT VARCHAR(100) DEFAULT MYNUM;
    WHILE MYNUM < 20 DO
        SET MYNUM = MYNUM + 1;
        SET RESULT = CONCAT(RESULT, ', ', MYNUM);
    END WHILE;
    SELECT RESULT;
END;
$$
delimiter ;
call temp_proc();
```

# Select Processing with Cursors (1 of 2)

- A **cursor** is a special construct used to hold data **rows** returned by a SQL query

```
DECLARE cursor_name CURSOR FOR select-query;
```

- Cursor-style processing involves retrieving data from the cursor one row at a time
  - When you fetch a row from the cursor, the data from the "current" row in the cursor is copied to the SQL variables you specified

# Cursors example

```
DELIMITER //

DROP PROCEDURE IF EXISTS ProcessSales;

CREATE PROCEDURE ProcessSales()
BEGIN
    DECLARE done INT DEFAULT false;
    DECLARE sale_total DECIMAL(10, 2);
    DECLARE sale_id DECIMAL(10,0);
    DECLARE grand_total DECIMAL(10,2);

    -- Declare the cursor
    DECLARE sales_cursor CURSOR FOR SELECT id, total FROM sales WHERE processed = false;

    -- Declare a SQL exception handler
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN
        -- Handle SQL exceptions
        ROLLBACK;
        SELECT 'An error occurred. Transaction rolled back.';
    END;

    -- Declare a handler for the NOT FOUND condition
    -- Otherwise, we get an error when the data is finished
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = true;

```

# Cursors example

```
26
27  -- Start a transaction
28  START TRANSACTION;
29
30  -- Open the cursor
31  OPEN sales_cursor;
32
33  -- Loop through the rows
34  SET grand_total = 0.0;
35  read_loop: LOOP
36      FETCH sales_cursor INTO sale_id, sale_total;
37      IF done THEN
38          LEAVE read_loop;
39      END IF;
40
41      -- Example logic: Process each sale and handle errors
42      BEGIN
43          -- Some processing logic (e.g., update sale status)
44          SET grand_total = grand_total + sale_total;
45          UPDATE sales SET processed = true WHERE id = sale_id;
46          -- all done
47          SELECT CONCAT("Sale id ", sale_id,
48                        " for amount ", sale_total,
49                        " processed successfully") as message;
50      END;
51
52  END LOOP;
```

# Cursors example

```
    END LOOP;

    SELECT CONCAT("Total sales processed: ", grand_total) as Total_sales_processed;

    -- Close the cursor
    CLOSE sales_cursor;

    -- Commit the transaction
    COMMIT;
END //

DELIMITER ;

```

# Cursors example

```
mysql> CALL ProcessSales();
+-------------------------------------------------------+
| message                                               |
+-------------------------------------------------------+
| Sale id 100 for amount 7.28 processed successfully |
+-------------------------------------------------------+
1 row in set (0.00 sec)

+-------------------------------------------------------+
| message                                               |
+-------------------------------------------------------+
| Sale id 101 for amount 39.20 processed successfully |
+-------------------------------------------------------+
1 row in set (0.00 sec)

+-------------------------------------------------------+
| message                                               |
+-------------------------------------------------------+
| Sale id 102 for amount 91.82 processed successfully |
+-------------------------------------------------------+
1 row in set (0.00 sec)

+-------------------------------------------------------+
| message                                               |
+-------------------------------------------------------+
| Sale id 103 for amount 3.07 processed successfully |
+-------------------------------------------------------+
1 row in set (0.00 sec)

+-------------------------------------------------------+
| message                                               |
+-------------------------------------------------------+
| Sale id 199 for amount 34.64 processed successfully |
+-------------------------------------------------------+
1 row in set (0.01 sec)

+-------------------------------+
| Total_sales_processed         |
+-------------------------------+
| Total sales processed: 4987.71 |
+-------------------------------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)

mysql> SELECT COUNT(*) AS number_of_false  FROM sales
WHERE processed = "false";
+-----------------+
| number_of_false |
+-----------------+
|               0 |
+-----------------+
1 row in set (0.00 sec)

mysql>
```