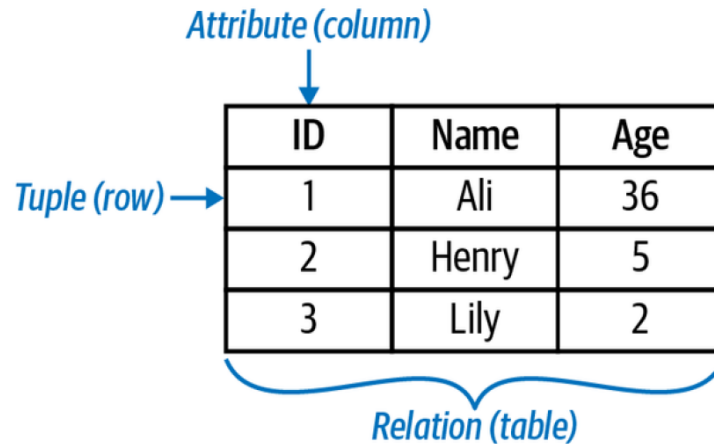


# SQL Notes

SQL is a structured query language that involves using relational databases, which requires pre-defined schemas.



CRUD Operations are most used in SQL. (Create, Read, Update, Delete)

SQL statements are divided into 4 main categories:

1. Data Definition Language (DDL)
  - CREATE (Create new database/table)
  - ALTER (Modify database/table structure)
  - DROP (Delete database or table)
  - TRUNCATE (Remove table records)
2. Data Manipulation Language (DML)
  - SELECT (Retrieve data from table)
  - INSERT (Insert new records into table)
  - UPDATE (Update existing records from a table)
  - DELETE (Remove existing records from a table)
3. Data Control Language (DCL)
  - GRANT (Assign privilege to users for accessing data)
  - REVOKE (Remove privilege to users for accessing data)
4. Transaction Control Statement (TCS)
  - COMMIT (Permanent work save into database)
  - ROLLBACK (Restore database to previous form since last commit)
  - SAVEPOINT (Create save point for future rollbacks)
  - SET TRANSACTION (Setting transaction to read-write/read only access)

## **SQL Query Basic Template**

SELECT <column\_names>

FROM <table\_name>

WHERE <condition on columns from table>

GROUP BY <column\_names>

HAVING <condition on grouped-by columns>

ORDER BY <column\_names>

SQL works in the following order of statement execution:

1. Gathers all data with FROM clause
2. Filters the data with WHERE clause
3. Groups rows together with GROUP BY clause
4. Filters grouped rows with HAVING clause
5. Specifies columns to display with SELECT clause
6. Sorts the results with ORDER BY clause

Types of RDBMS and its database tools:

1. SQLite: DB Browser for SQLite
2. MySQL: MySQL Workbench
3. Oracle: Oracle SQL Developer (PL/SQL – Procedural Language Extension to SQL)
4. PostgreSQL: pgAdmin
5. SQL Server: SQL Server Management Studio (T-SQL – Transact SQL)

ANSI (American National Standards Institute) standard in SQL refers to SQL code that will run in any RDBMS software.

## **Identifiers vs Aliases**

Identifiers are name of database objects

Aliases rename column or table temporarily, mostly useful in subqueries.

## **Statements vs Clauses**

Statements are blocks of code that starts with a SQL keyword like SELECT and ends with a semicolon.

Clauses are specific sections of the statement that refers to specific SQL keywords like WHERE, FROM etc.

## **Single vs Multi-Line Comments**

Single Line Comment:

-- This is a single line comment

Multi Line Comment:

/\* These are  
multi line comments \*/

## **Single vs Double Quotes**

Single quote: Used for string reference

Double quote: Used for identifier reference

## **Wildcard expressions**

%: Represents any n number of characters

\_: Represents any single character

Note that these wildcard expressions are used together with LIKE or NOT LIKE keyword in WHERE clause.

## **SQL Data Types**

### 1. Numeric (INT, DECIMAL, FLOAT)

- INT: Used for values that do not allow for decimals (i.e. 45)
- DECIMAL: Used for fixing number of decimals (i.e. 24.524)
- FLOAT: Store limited number of decimals with power notation (i.e.  $2.4524 * 10^5$ )

Note: MySQL has the option of setting numeric variables as positive only using UNSIGNED keyword.

### 2. String (CHAR, VARCHAR, TEXT, NCHAR, NVARCHAR)

- CHAR: Stores fixed length of characters as ASCII data
- VARCHAR: Stores maximum length of specified characters as ASCII data
- TEXT: Used for storing long strings of text like paragraphs
- NCHAR: Stores fixed length of characters as Unicode data
- NVARCHAR: Stores maximum length of specified characters as Unicode data

Note: Unicode data refers to non-ASCII characters (non-English mostly)

### 3. Date (DATE, TIME, DATETIME, TIMESTAMP, YEAR – For MYSQL)

- DATE: YYYY-MM-DD
- DATETIME: YYYY-MM-DD hh:mm:ss
- TIMESTAMP: YYYY-MM-DD hh:mm:ss UTC
- TIME: hh:mm:ss
- YEAR: YYYY

Note: DATETIME variable type does not store time zone, while TIMESTAMP variable type does store time zone.

### 4. Boolean (BOOLEAN): FALSE value as 0 and TRUE value as 1

Note: Boolean variable type is currently not supported in Oracle and SQL server.

### 5. External files like images, documents etc.

Approach 1: Store links to files using VARCHAR variable type

Approach 2: Convert files to binary format and store files using BLOB variable type

## **Common SQL Operators**

### **Logical Operators**

**AND:** Returns TRUE if both conditions are true or otherwise

**OR:** Returns TRUE if either condition is true or otherwise

**NOT:** Returns TRUE if condition is FALSE or otherwise

### **Comparison Operators (Used in conditional statements)**

=	Equality	<=	Less than or equal to
!=, <>	Inequality	>	Greater than
<	Less than	>=	Greater than or equal to

### **Comparison Keywords (Used in conditional statements)**

**BETWEEN:** Checks if value lies within given range (inclusive)

**EXISTS:** Checks if row exist in subquery (also known as semi-join)

**IN:** Checks if value is contained within a list using brackets symbol '(' )' (Note that having a single null value within the list will always result in FALSE for conditional statements. Thus, recommend using **NOT EXISTS** keyword instead.)

**IS NULL:** Checks if a value is null

**IS NOT NULL:** Checks if a value is not null

**LIKE:** Checks if value matches a simple pattern (Used with wildcard expressions)

**ALL:** Returns TRUE if all subquery values meet specified condition (Requires comparison operator)

**ANY:** Returns TRUE if any subquery values meet specified condition (Requires comparison operator)

Note that "**= ANY**" can be represented by IN keyword and "**!= ANY**" can be represented by NOT IN keyword.

## Math Operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (Remainder)

## Bitwise Operators

&: Bitwise AND

|: Bitwise OR

^: Bitwise XOR

## Aggregate Functions

Aggregate functions perform calculations on multiple rows to return a single result.

Aggregate functions are mostly used in SELECT and HAVING clause

**COUNT():** Counts number of values for a given attribute

**SUM():** Sum of values for a given attribute

**AVG():** Average of values for a given attribute

**MIN():** Minimum value for a given attribute

**MAX():** Maximum value for a given attribute

**LEAST():** Minimum value from set of attributes for every row

**GREATEST():** Maximum value from set of attributes for every row

Note that most of these aggregate functions in SQL applies to only non-null values, except for COUNT() function that applies to both null and non-null values.

## Numeric Functions

Numeric functions are usually applied to attributes with numerical values only.

<b>ABS(x)</b>	Returns absolute value of x	<b>LOG10(x)</b>	Returns log of x base 10
<b>SIGN(x)</b>	Returns sign of value of x (-1 for negative, 0 for zero or 1 for positive)	<b>MOD(x,y)</b>	Returns remainder of x/y
<b>POWER(x,y)</b>	Returns x to the power of y	<b>RAND()</b>	Returns a random number between 0 and 1
<b>SQRT(x)</b>	Returns square root of x	<b>CEIL(x)</b>	Returns upper integer boundary of value x
<b>EXP(x)</b>	Returns exponent of x	<b>FLOOR(x)</b>	Returns lower integer boundary of value x
<b>LOG(y,x)</b>	Returns log of y base x	<b>ROUND(x,n)</b>	Rounds x value to n decimal places
<b>LN(x)</b>	Return natural log of x base e	<b>TRUNC(x,n)</b>	Cuts off x value at n decimal places without rounding

## String Functions

String functions are usually applied to attributes with string data types.

**LENGTH(string):** Returns number of characters of a string

**UPPER(string):** Returns upper case characters of the whole string

**LOWER(string):** Returns lower case characters of the whole string

**TRIM(string):** Removes trailing and leading whitespaces from string

**TRIM(char FROM string):** Removes trailing and leading specified char from string

**LTRIM(string):** Removes leading whitespaces from string

**RTRIM(string):** Removes trailing whitespaces from string

**CONCAT(string1, string2, ...):** Concatenate multiple strings into a single string

**SUBSTRING(string, start, length):** Returns a portion of string starting from start index for n length.

**REPLACE(string, old\_substring, new\_substring):** Replace old\_substring from a given string with new\_substring.

**REGEXP 'Regular\_Expression':** Search for regular expression pattern

Finding index location of string syntax differs based on the software used.

**INSTR(string, substring) :** MySQL

**CHARINDEX(substring, string, position) :** SQL Server

**POSITION(substring IN string) :** PostgreSQL

Note: Regular expression syntax is best tested with the following website, instead of memorizing syntax: <https://regex101.com/>

### **Regular Expressions in SQL**

*	Zero or more instances of previous element	[...]	Any character within brackets
+	One or more instances of previous element	[^...]	Any character not within brackets
.	Any single character	{n}	N instance of previous element
?	Zero or one instance of previous element	{n,}	At least N instance of previous element
^	Beginning of string	{n,p}	N to p instance of previous element
\$	End of string	p1   p2	Matches any patterns p1, p2, ...

These quantifiers are usually used together with REGEXP keyword.

### **Datetime Format Specifiers**

%Y	4-digit year	%d	Day (1-31)
%y	2-digit year	%h	12 hours (1 – 12)
%m	Numeric month (1-12)	%H	24 hours (1 – 24)
%b	Abbreviated month (Jan – Dec)	%i	Minutes (0-59)
%M	Name of month (January – December)	%s	Seconds (0-59)

Note that datetime functions for different SQL engines have different syntax used. Thus, it is recommended to google it rather than memorizing syntax.

### **Null Functions**

Null functions are triggered when there is a missing value from attributes and an alternative output is provided as part of the function.

**COALESCE(attribute, alternative):** Returns “alternative” if null value exist in attribute



## **SELECT clause**

1. Select specified columns  
**SELECT c1, c2, ... FROM table1**
2. Select all columns  
**SELECT \* FROM table1**
3. Select expressions  
**SELECT UPPER(c1), ROUND(c1,2) FROM table1**
4. Select qualifying columns from multiple tables  
**SELECT table1.c1, table2.c2 FROM table1  
INNER JOIN table2 ON table1.c3 = table2.c3**
5. Select subqueries  
**SELECT c1, (SELECT COUNT(\*) FROM table2 WHERE table1.c2 =  
table2.c2) AS count\_name  
FROM table1**
6. Select all distinct values from columns  
**SELECT DISTINCT c1, c2, ... FROM table1**
7. Select number of distinct values from columns  
**SELECT COUNT(DISTINCT c1, c2, ...) FROM table1**

Note that selecting number of distinct values from multiple columns is currently supported in MySQL and PostgreSQL.

## **FROM clause**

1. Retrieve data from single table  
**SELECT \* FROM table1**
2. Retrieve data from multiple tables  
**SELECT a.\*, b.\* FROM table1 a  
INNER JOIN table2 b ON a.c1 = b.c1**
3. Retrieve data from table derived from subquery  
**SELECT c1, total  
FROM  
(SELECT c1, c2, COUNT(\*) AS total FROM table2 GROUP BY c1, c2  
HAVING COUNT(\*)>5) A**

### **WHERE clause**

1. Filters table based on single predicate (conditional statement)  
**SELECT \* FROM table1 WHERE condition1**
2. Filters table based on multiple predicates  
**SELECT \* FROM table2 WHERE condition1 AND condition2**
3. Filters table based on subquery  
**SELECT c1, c2 FROM table1 WHERE c3 IN (SELECT c3 FROM table1 WHERE condition)**

### **GROUP BY clause**

GROUP BY keyword used for summarizing rows into various groups, where collection of rows is specified in GROUP BY clause and aggregation functions are used in SELECT clause.

**SELECT c1, c2, SUM(c3)**

**FROM table1**

**GROUP BY c1, c2**

Note that all non-aggregated columns listed in SELECT clause must be included in GROUP BY clause and at least one aggregation function must be included in SELECT clause for query to work.

### **HAVING clause**

HAVING keyword used for filtering rows based on aggregation functions only after GROUP BY clause is used.

**SELECT c1, c2, SUM(c3) as total**

**FROM table1**

**GROUP BY c1, c2**

**HAVING SUM(c3) >5**

Note that filtering rows based on non-aggregated columns must be used in WHERE clause and using aliases on HAVING clause will result in error due to order of SQL statement execution.

## **ORDER BY clause**

ORDER BY keyword used for specifying order of results from query to be sorted.

Default sorting is in ascending order (ASC), unless specify DESC for descending order.

**SELECT c1, c2, ROUND(c4, 3) AS rounder FROM table1**

**ORDER BY c3 ASC, c2 DESC, rounder ASC**

Columns with aliases, other columns and expressions not within the SELECT clause can be used for sorting since FROM clause is executed first and SELECT clause is executed before ORDER BY clause.

Note that ORDER BY clause cannot be used within a subquery.

## **Subqueries**

Subqueries are queries nested inside another query, which can be found in various clauses like SELECT, FROM, WHERE or JOIN clause.

Subqueries must be surrounded by parenthesis () and including aliases except for subqueries in WHERE clause.

Subqueries are always executed from inner-most query to outer-most query.

Caveats to note for subqueries:

1. Using subqueries in SELECT clause requires result to return a single column and either zero or one row. If subquery returns one column and zero rows, the results will be filled with NULL values
2. Using subqueries in FROM clause requires the query to be independent without referring to values outside of the subquery.
3. Using subqueries in WHERE clause requires use of IN() or NOT IN() keywords if subquery returns multiple rows.

Advantages of subqueries in FROM clause:

1. Reduce problems to smaller steps required
2. Reduce table size in FROM clause to improve query performance

Advantages of subqueries in WHERE clause: Easier for code understanding

### **Correlated subqueries**

Correlated subqueries are queries that refer to values obtained in outer queries.

Correlated subqueries execute once for every row from the outer query. Thus, correlated subqueries come with read performance issues.

It is recommended to use other approaches like Joins or CTE (Common table expressions) when possible.

### **Limiting view of number of rows**

Different syntax is used for this task depending on SQL software used.

For MySQL, PostgreSQL (Note that LIMIT clause is executed last after ORDER BY clause):

**SELECT cl1, cl2**

**FROM owner**

**LIMIT n**

For SQL Server:

**SELECT TOP n cl1, cl2**

**FROM owner**

For Oracle:

**SELECT cl1, cl2**

**FROM owner**

**WHERE ROWNUM <= n**

## **Functions for summarizing rows into lists**

Using together with GROUP BY clause, the following functions can be used for summarizing multiple rows into list of values:

MYSQL	<b>GROUP_CONCAT (DISTINCT x ORDER BY x SEPARATOR ‘,’)</b>
SQL Server	<b>STRING_AGG(x, ‘,’) WITHIN GROUP (ORDER BY x)</b>
PostgreSQL	<b>ARRAY_AGG(DISTINCT x ORDER BY x)</b>

Note the following caveats:

1. SQL server currently does not support “Unique” list
2. PostgreSQL only accepts ‘,’ as default separator
3. SQL server requires input of separator, while input of separator is optional for MYSQL

## **Complex Group-By Functions**

Additional summary information can also be included in queries using the following complex group-by functions:

1. GROUP BY ROLLUP (a,b, ...,e)
  - Group by sets of a – e, a – d, .... a and ()
2. GROUP BY CUBE (a,b, ..., e)
  - Group by all combination of sets and ()
3. GROUP BY GROUPING SETS(a,b,...,())
  - Group by individual categories and (), which represents group function on entire table

Note that these functions are mostly supported in Oracle, PostgreSQL and SQL server.

MySQL only supports rollup function using the following syntax:

**GROUP BY a, b, ..., e WITH ROLLUP**

## **Window Functions**

Window functions return value for each row of the dataset that relates to result from current query.

Window functions are mostly used in SELECT & ORDER BY clause and useful for combining non-aggregated columns and aggregated columns together without relying on GROUP BY clause.

Filtering columns related to window functions requires use of subqueries, because window functions are executed 2<sup>nd</sup> last prior to ORDER BY clause.

Basic syntax:

**AGG\_FUNC() OVER (PARTITION BY c1 ORDER BY c2)**

Note that PARTITION BY clause has similar behavior as GROUP BY clause and basic aggregation functions can also be used in the syntax above.

By default, ORDER BY clause within OVER clause has “Cumulative function” behavior (hidden syntax is **RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**)

The behavior (sliding windows) of ORDER BY clause within OVER clause can be adjusted to the following:

1. Moving N previous records  
**RANGE BETWEEN N PRECEDING AND CURRENT ROW**
2. Moving N following records  
**RANGE BETWEEN CURRENT ROW AND N FOLLOWING**
3. Moving N previous and N following records  
**RANGE BETWEEN N PRECEDING AND N FOLLOWING**

Note that **RANGE BETWEEN** can be replaced by **ROWS BETWEEN** instead.

RANGE BETWEEN will return same running value for each duplicated time, while ROWS BETWEEN returns different running value for each row

List of window functions:

1. **ROW\_NUMBER()** : Returns row number of every record without “ties”
2. **RANK()** : Returns rank of current row with “ties” allowed and “gaps” allowed
3. **DENSE\_RANK()** : Returns rank of current row with “ties” allowed and “gaps” not allowed
4. **FIRST\_VALUE(cl1)** : Returns value of cl1 at 1<sup>st</sup> row of window frame
5. **LAST\_VALUE(cl1)** : Returns value of cl1 at last row of window frame
6. **NTH\_VALUE(cl1, n)** : Returns value of cl1 at nth row of window frame
7. **LEAD(cl1, n)** : Returns value of cl1 at n rows after current row (Default n is 1)
8. **LAG(cl1, n)** : Returns value of cl1 at n rows before current row (Default n is 1)
9. **NTILE(n)** : Returns rank of current row by dividing partition into n-sections

### Types of Joins

Joins can be used in SQL for combining multiple tables with similar characteristics.

**SELECT \***

**FROM table1 AS A**

**INNER/LEFT/RIGHT JOIN table2 AS B**

**ON A.cl1 = B.cl1**

Note that ON clause can be replaced with **USING(cl1)** if both tables are joined together using the same column name cl1

Cross Joins can also be used for returning all combination of rows in multiple tables.

**SELECT \***

**FROM table1**

**CROSS JOIN table2**

Self joins can also be used for comparing between rows in the same table.

```
SELECT *  
FROM table1 a  
INNER JOIN table1 b  
ON a.cl1 = b.cl1 AND a.cl2 != b.cl2
```

### **Stacking Results of Multiple Queries**

While joins can be used to link multiple tables based on common columns, UNION operator can be used to stack results of multiple queries.

```
(SELECT cl1, cl2 FROM table1  
WHERE condition1)  
UNION/UNION ALL  
(SELECT cl1, cl2 FROM table2  
WHERE condition2  
ORDER BY cl1)
```

Note that UNION/UNION ALL operator requires the following conditions:

1. Same number of columns
2. Same order of columns
3. Similar data types between columns from multiple queries

UNION operator is used for selecting results that are distinct between queries, while UNION ALL operator is used for selecting all results including duplicates from multiple queries.

### **CASE statements**

Case statements are similar to if-then-else statements.

```
CASE  
    WHEN condition1 THEN result1  
    WHEN condition2 THEN result2  
    ELSE result3  
END AS Alias_Name
```



## Common Table Expressions

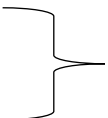

Common table expressions (CTE) are temporary results that can be used in other queries.

Two types of CTE:

1. Non-recursive CTE

```
WITH cte_name (cl1, cl2, ...) AS (  
Query1  
)  
SELECT cl1, cl2, ...  
FROM cte_name
```

2. Recursive CTE

```
WITH RECURSIVE cte_name (cl1, cl2, ...) AS(  
Query1  
UNION  Anchor members (Starting points)  
Query2  
UNION ALL  
SELECT ...  
FROM cte_name  Recursive member  
WHERE condition # Termination Check  
)  
SELECT cl1, cl2, ...  
FROM cte_name
```

Note that MySQL and PostgreSQL does not require RECURSIVE keyword, while SQL server requires RECURSIVE keyword.

Note that multiple CTE can be created within the same WITH clause.

```
WITH cte_name1(cl1, cl2, ...) AS(query),  
cte_name2(cl1, cl2, ...) AS(query) ...
```

Advantages of using CTE over Subqueries:

1. CTE name can be referenced multiple times in multiple queries
2. Multiple CTE names can be defined upfront before main query
3. CTE can be referenced within each other

However, older versions of SQL software do not support CTE. Thus, subqueries are still widely used.

Recursive CTE are mostly used in the following scenarios:

1. Filling in missing rows of sequence of data
2. Return all parents of a child row (hierarchical data structure)

Note the following caveats for CTE:

1. Number of columns defined in CTE must match number of columns defined within WITH clause
2. Same number of columns and data types in both anchor and recursive members of the CTE
3. FROM clause of recursive member refers only one time to cte\_name
4. UNION used to stack multiple anchor members together
5. UNION ALL used to stack last anchor member with multiple recursive members together
6. Following keywords/functions not allowed in recursive members:
  - SELECT DISTINCT
  - GROUP BY
  - HAVING
  - ORDER BY
  - All joins (except for inner join)
  - Subqueries
  - Group functions (Aggregation and Window functions)

## Data model/Schema

Data model/Schema describes how database objects are organized within a database

The following actions can be done with respect to databases:

1. Creating new database

**CREATE DATABASE db\_name**

2. Delete existing database

**DROP DATABASE db\_name**

3. Using database (MySQL and SQL Server)

**USE db\_name**

4. Display names of all existing databases

**SHOW databases** (MySQL)

**SELECT name FROM master.sys.databases** (SQL Server)

5. Display name of current database

**SELECT database()** (MySQL)

**SELECT db\_name()** (SQL Server)

## Creating/Modifying Tables

The following actions can be done with respect to tables:

1. Creating new tables

```
CREATE TABLE IF NOT EXISTS tb_name(
    var1 dtype1,
    var2 dtype2,
    ...
)
```

2. Remove whole data inside table

**TRUNCATE TABLE tb\_name**

3. Remove whole table

**DROP TABLE IF EXISTS tb\_name**

4. Insert records into tables

```
INSERT INTO tb_name (cl1, cl2, ...)  
VALUES (cl1, cl2, ...), (cl1, cl2, ...)
```

5. Delete records from tables

```
DELETE FROM tb_name  
WHERE condition
```

6. Update records from tables

```
UPDATE tb_name  
SET cl1 = value1, ...  
WHERE condition
```

7. Display names of existing tables

```
SHOW tables (MySQL)
```

```
SELECT tb_name FROM information_schema.tables (SQL Server)
```

8. Display columns of existing tables

```
DESCRIBE tb_name (MySQL)
```

```
SELECT cl1, cl2, ...  
FROM information_schema.columns WHERE table_name = tb_name (SQL  
Server)
```

9. Adding new columns

```
ALTER TABLE tb_name  
ADD cl_name dtype
```

10. Removing columns

```
ALTER TABLE tb_name  
DROP COLUMN cl_name
```

11. Change data type of columns

```
ALTER TABLE tb_name  
ALTER COLUMN cl_name dtype
```

## **SQL Constraints**

Constraints are rules that specify what data can be inserted into a table.

Types of SQL Constraints:

1. NOT NULL: Restrict columns to not allow null values
2. DEFAULT: Setting default values that replaces missing values
3. CHECK: Filters inputs based on certain conditions
4. UNIQUE: Restrict columns to only allow unique values
5. PRIMARY KEY: Candidate key that uniquely identifies a row
6. FOREIGN KEY: Candidate key within one table that matches with candidate key in other tables.

Note that primary key must be created first, before creating foreign keys.

Example:

```
CREATE TABLE table1(
    id INTEGER PRIMARY KEY
)
CREATE TABLE table2(
    id INTEGER PRIMARY KEY,
    FOREIGN KEY id1 REFERENCES table1(id)
)
```

For creating table with auto-generated fields, following syntax is used:

**cl\_name dtype PRIMARY KEY AUTO\_INCREMENT** (MySQL)

**cl\_name dtype IDENTITY(1,1)** (SQL Server)

The following actions can be done for constraints:

1. Create new constraints when creating tables  
**CONSTRAINT ct\_name .....**
2. Create new constraints when altering tables  
**ALTER TABLE tb\_name  
ADD CONSTRAINT ct\_name .....**
3. Modify existing constraints when altering tables  
**ALTER TABLE tb\_name  
ALTER COLUMN cl\_name dtype new\_ct**
4. Remove existing constraint  
**ALTER TABLE tb\_name  
DROP CONSTRAINT ct\_name**
5. Display table constraints  
For MySQL:  
**SHOW CREATE TABLE tb\_name**

For non-default constraints in SQL Server:

```
SELECT table_name, constraint_name, constraint_type  
FROM information_schema.table_constraints  
WHERE table_name = 'tb_name'
```

For default constraints in SQL Server:

```
SELECT OBJECT_NAME(parent_object_id),  
COL_NAME(parent_object_id, parent_column_id), definition  
FROM sys.default_constraints  
WHERE OBJECT_NAME(parent_object_id) = 'tb_name'
```

## Indexes

Indexes are useful for retrieving data from very large datasets more quickly that speeds up SELECT and WHERE queries.

However, index is not useful under following scenarios:

1. Columns with frequent, large batch updates on insert operations
2. Columns with large number of null values
3. Tables with small number of records
4. Columns that are frequently manipulated

The following actions can be done for indexes:

1. Creating new index  
**CREATE INDEX index\_name**  
**ON tb\_name (cl1, cl2, ...)**
2. Remove existing index  
**DROP INDEX index\_name ON tb\_name**

## Views

Views can be created for the following benefits:

1. Restrict users from accessing sensitive information on the original database
2. Hide and reuse complex queries
3. Provide more meaningful aliases to variables in the original database

There are two types of views in SQL:

1. Inline views: Temporary results from subqueries used in FROM clause
2. Views: Referenced virtual tables from result of query that does not take up additional storage space and can be reused

The following actions can be done for views:

1. Create new views/replace existing views  
**CREATE VIEW vw\_name AS**  
**Query**
2. Remove existing views  
**DROP VIEW vw\_name**

3. Display existing views

For MySQL:

**SHOW FULL\_TABLES WHERE table\_type = 'VIEW'**

For SQL Server:

**SELECT table\_name FROM information\_schema.views**

Note that data cannot be inserted or removed from views once created.

## **Transactions**

Transaction allows for safer updates on database through a sequence of SQL operations as a single unit.

Using the concept of atomicity, either all SQL operations are executed or not under a transaction.

Transaction Keywords in SQL:

1. **START TRANSACTION / BEGIN TRANSACTION:** Used to initiate a new transaction (Start transaction keyword used for MySQL and PostgreSQL, while Begin transaction keyword used for SQL Server)
2. **SAVEPOINT point\_name:** Used to save state of transaction at specific points of the query code in which to rollback.
3. **RELEASE SAVEPOINT point\_name:** Used to remove state of transaction saved.
4. **ROLLBACK TO point\_name:** Rollback changes to before transaction occurs or back to a previous savepoint.
5. **COMMIT:** Make permanent changes to the database

Note that once a transaction is committed, it is no longer reversible.