Structured Query Language (SQL)

SQL is scripting syntax for running database queries and performing Create, Read, Update, and Delete (CRUD) operations in a relational database system. Depending on the database management system (DBMS), different versions of SQL will have slight syntactical variations. However, foundational CRUD syntax will typically be common among DBMSs (e.g. SELECT, INSERT, UPDATE, DELETE, etc.).

Here we will explore how SQL is used to build database architectures and navigate the data within it. We start by understanding how basic tables are constructed in a relational database, how SQL builds tables that comprise an architecture, and how SQL creates, reads, updates, and deletes data, among other operations.

**Primary Key:**  Primary keys in a database are records’ unique identifiers.  These values cannot be repeated across records within a given table (unique values), and are used to index records for efficient data retrieval.  When implemented properly in a relational database, primary keys are referenced in other tables as foreign keys.

* Often the first column in a table
* Each table in a relational database usually only assigns one column as its primary key field.
* Never null values
* Always unique values (never reused among records)
* Can never be modified once created

**Foreign Key:**  Foreign keys in a database are reference values with which records in a data table are related to records in another data table.  These reference values are primary keys in their source tables that encapsulate all the data in the corresponding record. Note that it is not good practice for a table reference its own values.

**Applying Keys to Database Tables**

The table below shows records for an office supply company. It tracks their products and indicates their respective product managers. While this data is valuable to track, its database implementation is inefficient given the number of repeated records.

A screenshot of a computer screen

AI-generated content may be incorrect.

Implementing primary keys and foreign keys is a valuable strategy for designing efficient databases. The goal is to reduce the number of repeated records in any given table, thus reducing the memory required to store the database. A key benefit in reducing repeated records in a database is more efficient queries.

The orange table above can be broken out into two tables to eliminate the repeated product manager records. By creating a managers table comprised of unique manager records, the products table becomes less cluttered by eliminating 4 columns of repeated values. This is where primary keys and foreign keys become useful!

A screenshot of a computer

AI-generated content may be incorrect.

The blue table above shows records for product managers' contact infromation. The 'manager\_id' column is defined as a primary key and cast as an integer type. The 'first\_name', 'last\_name', 'email\_address', and 'phone\_number' columns are cast as string types.

Note that the 'phone\_number' column contains numeric values, but the dashes in each value make the records ineligible for numeric types. Alternatively, the dashes could be removed in extract, transform, load (ETL) and the records can then be stored as numeric types, specifically integer types.

A screen shot of a product

AI-generated content may be incorrect.

The green table above implements primary keys and foreign keys to detail a company's products. The 'product\_id' column is defined as a primary key and cast as an integer type. The 'product\_number', 'product\_name', and 'unit' columns are cast as string types, and the 'product\_unit\_price' column is cast as a float type.

The 'product\_manager' column is defined as a foreign key to the managers table primary keys and cast as and integer type, which relates each product to a manager. This also simplifies manager reassignments, as only the foreign key value needs to be updated to the reassigned manager. Furthermore, the manager table can be used in other tables throughout the databaase, keeping a consistent link between managers, products, and other tracked instances.

The foreign keys in the 'product\_manager' column reference the primary keys in the managers table, relating the records in the products table to their product manager.

* The products table primary key, product\_id, 1 is Pencil, encapsulating all recorded pencil data within the key. Among the pencil's data is the foreign key, product\_manager, 3 which references the managers table primary key, manager\_id. So Pencil is associated Alex Tan.
* Similarly:
  + Alex Tan (3) is also the product manager for paper (2).
  + Ben Smith (1) is the product manager for eraser (3), tape (6), and stickynote (7).
  + Sam Gomez (2) is the product manager for paperclip (4)
  + Sarah Lewis (4) is the product manager for stapler (5) and folder (8)

Once the database is architected and loaded with data, SQL is used to interact with the data. Interactions include, but are not limited to:

* Retrieving existing data
* Inserting new data
* Modifying existing data
* Deleting existing data

Simply put, SQL performs queries, which are action requests.

***SQL Database Architecture Syntax***

Databases are structured storage containers that store data. They are comprised of tables that are formatted into rows and columns, facilitating data retrieval and manipulation.

Each row is a data record, comprised of data distributed among its column. Each database column, or field, is assigned a label to identify the data stored in that table cell.

Rows are not inherently ordered, so they must be assigned an identifying value, known primary keys. Using row indices or record primary keys with column names, records can be accessed by identifying their row-column intersection in the table.

Below are some useful SQL operations for navigating and building a database architecture. Note that SQL syntax can vary across DBMSs, so red notes have been added where syntax variation is possible.

**SHOW DATABASE**

This query shows all existing databases

SHOW DATABASE

**CREATE DATABASE**

This query creates a new database by the name of <DATABASE\_NAME> in a server.

CREATE DATABASE <DATABASE\_NAME>

Attempting to create a database by the name of an existing database in the same server will yield an error, so adding the keyword 'IF NOT EXISTS' first checks if the database name already exists without raising an error.

CREATE DATABASE IF NOT EXISTS <DATABASE\_NAME>

In both cases above, if a database creation by the name of an existing database is attempted , the attempt will not complete.

**DROP DATABASE**

This query deletes an existing database by the name of <DATABASE\_NAME> in a server.

DROP DATABASE <DATABASE\_NAME>

Attempting to delete a database that does not exist in a given server will yield an error, so adding the keyword 'IF NOT EXISTS' first checks if the database name exists without raising an error.

DROP DATABASE IF EXISTS <DATABASE\_NAME>

**USE DATABASE**

Working with a database's tables requires first telling SQL which database to use by its <DATABASE\_NAME>.

USE <DATABASE\_NAME>

**SHOW TABLES**

This query shows all existing tables in a database

SHOW TABLES

**EXPLAIN**

This query examines <TABLE>'s column specifications. Note that this simply lists column names, details about the type of data it stores, and its restrictions - not the data itself.

EXPLAIN <TABLE>

**CREATE TABLE**

This query creates a new table by the name <TABLE>.

The 'IF NOT EXISTS' keyword is available but not required. This first checks if <TABLE> exists in the database before creating one without raising an error.

CREATE TABLE IF NOT EXISTS <TABLE>

(

<FIELD\_NAME> <FIELD\_DATA\_TYPE> <FIELD\_MODIFIER>,

<FIELD\_NAME> <FIELD\_DATA\_TYPE> <FIELD\_MODIFIER>,

<FIELD\_NAME> <FIELD\_DATA\_TYPE> <FIELD\_MODIFIER

)

**FIELD DATA TYPES**

Data types are used to define the type of data that is stored in a field. The field's data type is defined when the table is created, and all values within that column must adhere to the defined data type.

|  |  |  |
| --- | --- | --- |
| BOOL | Boolean value | TRUE |
| INT | Integer number | 100 |
| DECIMAL(<M>, <N>) | Decimal, or floating point, that specifies the number of permissible digits, where <M> is the total number of digits in the value and <N> is the number of digits after the decimal | DECIMAL(5, 2): 100.25 |
| DOUBLE | Double-precision floating point number |  |
| DATE | Date formatted as YYYY-MM-DD | 1970-01-01 |
| TIME | Time formatted as HH:MM:SS | 12:30:56 |
| DATETIME | Combined date and time formatted as YYYY-MM-DD HH:MM:SS | 1970-01-01 12:30:56 |
| YEAR | Year formatted as YY or YYYY | 1970 |
| TIMESTAMP | Automatic date and time of record entry | 1970-01-01 12:30:56 |
| CHAR(<N>) | String of defined length up to 255 characters that is stored with padding | CHAR(15): 'HELLO WORLD\_ \_ \_ \_" |
| VARCHAR(<N>) | String of variable length up to 255 characters that is stored without padding | VARCHAR(15): 'HELLO WORLD" |
| TEXT | String of up to 65,535 characters | "HELLO WORLD" |
| BLOB | Binary type for variable data |  |
| ENUM(<VAL\_0>, <VAL\_1>) | Single string value from a defined list of multiple permitted values (only 1 value), up to thousands | ENUM('RED', 'YELLOW', 'BLUE') 'BLUE' |
| SET(<VAL\_0>, <VAL\_1>) | String of multiple strings from a defined list of multiple permitted values (one or more values), up to 64 options | ENUM('RED', 'YELLOW', 'BLUE') ('RED', 'BLUE') |

*ENUM() and SET() are mote involved data types that will be elaborated upon later.*

**FIELD MODIFIERS**

Field modifiers define the behavior of a field. The field's modifier is defined when the table is created, and all values within that column adopt the defined modifier's behavior.

|  |  |
| --- | --- |
| PRIMARY KEY | Indicates that the values in that field are primary keys for the records in that table |
| UNIQUE | Indicates that values cannot be repeated among records in that field |
| NOT NULL | Indicates that records must contain a value in that field |
| AUTO\_INCREMENT | For numeric fields, automatically increments values by one from the previous record's value |
| DEFAULT | Indicates a value to use when no value is given for that field upon record insertion |

**ALTER TABLE**

A <TABLE>'s format can be modified with the 'ALTER\_TABLE' query. One or many modifications can be made to a table by creating a comma-separated list of intended alteration in the query.

ALTER TABLE <TABLE>

<ALTERATION\_KEYWORD> <ALTERATION\_SYNTAX>,

<ALTERATION\_KEYWORD> <ALTERATION\_SYNTAX>,

<ALTERATION\_KEYWORD> <ALTERATION\_SYNTAX>

|  |  |  |
| --- | --- | --- |
| **ALTERATION\_KEYWORD** | **Definition** | **ALTERATION\_SYNTAX** |
| ADD | Add a new column to an existing table | ALTER TABLE <TABLE> ADD <FIELD\_NAME> <FIELD\_DATA\_TYPE> <FIELD\_MODIFIER> |
| ADD PRIMARY KEY | Adds a primary key to an existing column | ALTER TABLE <TABLE> ADD PRIMARY KEY <COL> |
| RENAME COLUMN | Renames the indicated existing column | ALTER TABLE <TABLE> RENAME COLUMN <OLD\_FIELD\_NAME> TO <NEW\_FIELD\_NAME> |
| DROP COLUMN | Permanently deletes a column from a table | ALTER TABLE <TABLE> DROP COLUMN <FIELD\_NAME> |
| ALTER (or MODIFY) \**Syntax varies by DBMS* | Changes the field's data type | ALTER TABLE <TABLE> ALTER COLUMN <FIELD\_NAME> TYPE <NEW\_DATA\_TYPE> |

**DROP TABLE**

This query deletes a new table by the name <TABLE>.

The 'IF EXISTS' keyword is available but not required. This first checks if <TABLE> exists in the database before deleting it without raising an error.

DROP TABLE IF EXISTS <TABLE>

Table dependencies must be considered when deleting tables from a relational database. Tables that are depended on by other tables - in other words, tables that contain primary keys that other tables have foreign keys depend on - cannot be deleted. Doing so would break the dependency chain, which SQL constraints protect from.

To delete these tables, first delete tables that contain foreign keys but no depended-on primary keys. Iterate through this process until the desired table is deleted, but do not disable table constraints as that can compromise data integrity.

Alternatively, SQL offers a CASCADE option for the 'DROP TABLE' query, which deletes the table and its dependent objects.

DROP TABLE IF EXISTS <TABLE> CASCADE

*SQL Data Query Syntax*

**COMPARISON OPERATORS**

* **=:** Equal to
* **>:** Greater than
* **<:** Less than
* **>=:** Greater than or equal to
* **<=:** Less than or equal to
* **<>:** Not equal to
* **BETWEEN <MIN> AND <MAX>:** Within range of min to max
* **IS NULL:** Is a null value
* **IS NOT NULL:** Is not a null value

**LOGICAL OPERATORS**

* **AND:** Operator that selects data when ALL conditions are TRUE
* **OR:** Operator that selects data when ANY conditions are TRUE

**COMMENTING QUERIES**

Comments can be placed in a query by enclosing the comment text in /\* ... \*/. Any text enclosed within this syntax will be ignored when executing a query.

/\* THIS IS SQL COMMENT SYNTAX \*/

**SELECT**

The 'SELECT' statement selects data for retrieval from a database. The <COL> portion following 'SELECT' are the field names, or table column names, each separated by a comma.

After indicating fields, the 'FROM' statement indicates the table from which data is being retrieved. Note that if specific field names are indicated in the 'SELECT' statement, then those fields must be present in the indicated table.

Depending on the database architecture, the query may need to indicate the schema in which the table is stored:

**<SCHEMA\_NAME>.<TABLE\_NAME>**

SELECT <COLS>

FROM <TABLE>

If all fields are to be retrieved, the '\*' can be used for the <COL> portion instead of explicitly stating each field name in the query.

SELECT \*

FROM <TABLE>

**DISTINCT**

The 'DISTINCT' keyword is used in the 'SELECT' statement to retrieve unique values from the indicated <COL>. Multiple <COL>s can be listed after 'DISTINCT', each separated by a comma, which will prompt 'DISTINCT' to retrieve unique data combinations by row of data in the indicated <COL>s.

SELECT DISTINCT <COLS>

FROM <TABLE>

Like standard 'SELECT' statements, a 'DISTINCT' keyword can select all table fields with the '\*'.

SELECT DISTINCT \*

FROM <TABLE>

**WHERE**

The 'WHERE' clause is used to conditionally filter data, where only records satisfying the indicated conditions are retrieved. It uses comparison operators, to compare record values to an indicated value. If the condition yields TRUE, then the entire record is retrieved in the query.

SELECT <COLS>

FROM <TABLE>

WHERE <CONDITIONAL\_COL> <COMPARISON\_OPERATOR> <VALUE>

Multiple conditions can be indicated in the 'WHERE' clause using logical operators

SELECT <COLS>

FROM <TABLE>

WHERE <CONDITIONAL\_COL> <COMPARISON\_OPERATOR> <VALUE>

<LOGICAL\_OPERATOR> <CONDITIONAL\_COL> <COMPARISON\_OPERATOR> <VALUE>

<LOGICAL\_OPERATOR> <CONDITIONAL\_COL> <COMPARISON\_OPERATOR> <VALUE>

Queries can also be performed to identify records that DO NOT match the defined conditions using one of two methods:

1. The first method places the keyword 'NOT' before the <CONDITIONAL\_COL>.

SELECT <COLS>

FROM <TABLE>

WHERE NOT <CONDITIONAL\_COL> <COMPARISON\_OPERATOR> <VALUE>

2. The second method places a '!' before the operator to negate it.

SELECT <COLS>

FROM <TABLE>

WHERE <CONDITIONAL\_COL> !<COMPARISON\_OPERATOR> <VALUE>

Note that some versions of SQL do not recognize '!>' and '!<', as they are

equivalent to '<' and '>', respectively. '<>' is acceptable in place of '!=',

but not encouraged.

**NULL**

The 'NULL' keyword is used to indicate record fields that do not contain values. This keyword exists because null fields cannot be identified with logical operators.

SELECT <COLS>

FROM <TABLE>

WHERE <CONDITIONAL\_COL> IS NULL

The 'NULL' keyword can also be used with the 'NOT' operator, negating the 'NULL' and instead returning records with fields that do contain values.

SELECT <COLS>

FROM <TABLE>

WHERE <CONDITIONAL\_COL> IS NOT NULL

**IN**

The 'IN' operator enables queries to apply multiple 'WHERE' parameters with the 'OR' logical operator in a single query.

In other words,

SELECT <COLS>

FROM <TABLE>

WHERE <CONDITIONAL\_COL> = <VAL\_0>

OR <CONDITIONAL\_COL> = <VAL\_1>)

can alternately be expressed as

SELECT <COLS>

FROM <TABLE>

WHERE <CONDITIONAL\_COL> IN (<VAL\_0>, <VAL\_1>)

The 'IN' operator can also be used with the 'NOT' operator, negating 'IN' and instead selecting records where values are not among the the indicated values.

SELECT <COLS>

FROM <TABLE>

WHERE <CONDITIONAL\_COL> NOT IN (<VAL\_0>, <VAL\_1>)

**BETWEEN**

The 'BETWEEN' operator is used with the 'WHERE' clause to select records where field values fall within the indicated range.

The indicated range can be numeric or alphabetic, where alphabetic values are strings enclosed in quotations and numeric values are integers or floats.

SELECT <COLS>

FROM <TABLE>

WHERE <COL> BETWEEN <MIN\_VAL> AND <MAX\_VAL>

The 'BETWEEN' operator can also be used with the 'NOT' operator, negating 'BETWEEN' and instead returning records where values do not fall in the indicated range.

SELECT <COLS>

FROM <TABLE>

WHERE <COL> NOT BETWEEN <MIN\_VAL> AND <MAX\_VAL>

**LIKE**

The 'LIKE' operator is used in a 'WHERE' clause to search for an indicated pattern in the indicated <COL>. This is especially useful when querying string values for which exact values may be unknown. The syntax uses wildcard characters to define the patern:

* **'%'**: Represents zero to many characters
* **'\_'**: Represents single characters

'LIKE' operator syntax with wildcard characters is detailed below.

SELECT <COLS>

FROM <TABLE>

WHERE <COL>

* LIKE ‘A%’:  Records that start with ‘A’
* LIKE ‘%A’:  Records that end with 'A'
* LIKE ‘%A%’:  Records that contain ‘A’
* LIKE ‘A%B’:  Records that start with ‘A’ and end with ‘B’
* NOT LIKE ‘A%’:  Records that DO NOT start with ‘A’
* LIKE ‘\_A%’:  Records that have ‘A’ in the second letter place
* LIKE ‘\_\_A%’:  Records that have ‘A’ in the third letter place
* LIKE ‘\_\_A\_’:  Records that have ‘A’ in the third letter place of a 4-character string
* LIKE ‘[ACF]%’:  Records that start with ‘A’, ‘C’, or ‘F’
* LIKE ‘[A-F]%’:  Records that start with any letter from ‘A’ to ‘F’
* LIKE ‘[!ACF]%’:  Records that do not start with ‘A’, ‘C’, or ‘F’

**ORDER BY**

The 'ORDER BY' keyword sorts the queried records by the parameter values in the indicated <COL>.

The sorting direction can be defined with an optional <SORT\_KEYWOR>:

* **ASC (default):**  Sort from first to last, low to high
* **DESC:**  Sort from last to first, high to low

SELECT <COLS>

FROM <TABLE>

ORDER BY <SORT\_COL> <SORT\_KEYWORD>

An 'ORDER BY' can be performed along multiple fields. So fields will be sorted in the order in which they are indicated in the 'ORDER BY' clause. Assuming we sort <TABLE> along <SORT\_COL\_0>, then <SORT\_COL\_1>:

SELECT <COLS>

FROM <TABLE>

ORDER BY <SORT\_COL\_0> <SORT\_KEYWORD>, <SORT\_COL\_1> <SORT\_KEYWORD>

So, <TABLE> will be sorted along <SORT\_COL\_0>, then records that have common values along the first sort will be sorted further along <SORT\_COL\_1>.

**GROUP BY**

The 'GROUP BY' statement groups records that have matching values in the indicated <GROUPING\_COL>, then performs the indicated <FUNCTION> on each group's values. This query returns the unique <GROUPING\_COL> groups and the <FUNCTION> results.

SELECT <FUNCTION>(<COL>)

FROM <TABLE>

GROUP BY <GROUPING\_COL>

Multiple <FUNCTION>s can be applied to the groupings, and the results for each function are returned in their own columns, as indicated in the 'SELECT' statement.

SELECT <FUNCTION\_0>(<COL>), <FUNCTION\_1>(<COL>), <GROUPING\_COL>

FROM <TABLE>

GROUP BY <GROUPING\_COL>

**INSERT**

The 'INSERT INTO' operator inserts new records into the indicated table. The query first indicates the field names, which must match those in the indicated table. The 'VALUES' are then listed in the order in which their corresponding columns are indicated.

INSERT INTO <TABLE> (<COL\_0>, <COL\_1>, ... ,<COL\_N>)

VALUES(<COL\_0\_VALUE>, <COL\_1\_VALUE>, ..., <COL\_N\_VALUE>)

Column names do not need to be indicated if values are being inserted into all table columns. However, the order in which the values indicated for insertion are presented in the query must match their placement sequence in the table if the 'INSERT INTO' operator is not going to indicate columns.

Multiple records can also be inserted in a single query by placing each set of new record values in parentheses and comma separating each new record in the statement.

INSERT INTO <TABLE> (<COL\_0>, <COL\_1>, ... ,<COL\_N>)

VALUES

(<COL\_0\_VALUE\_0>, <COL\_1\_VALUE\_0>, ..., <COL\_N\_VALUE\_0>),

(<COL\_0\_VALUE\_1>, <COL\_1\_VALUE\_1>, ..., <COL\_N\_VALUE\_1>),

(<COL\_0\_VALUE\_2>, <COL\_1\_VALUE\_2>, ..., <COL\_N\_VALUE\_2>),

...

(<COL\_0\_VALUE\_M>, <COL\_1\_VALUE\_M>, ..., <COL\_N\_VALUE\_M>)

**UPDATE**

The 'UPDATE' statement updates existing record values in a table with new indicated values.

UPDATE <TABLE>

SET <COL\_0> = <VALUE\_0>

The 'UPDATE' statement can use the 'WHERE' clause to conditionally update values. Multiple field updates can be performed simultaneously by separating each record-to-update with a comma.

UPDATE <TABLE>

SET <COL\_0> = <VALUE\_0>,

<COL\_1> = <VALUE\_1>

WHERE <CONDITIONAL\_COL> <OPERATOR> <CONDITIONAL\_VALUE>

**DELETE**

The 'DELETE' statement removes records from the indicated table. If no conditions are defined in a 'DELETE' operation, then all records will be deleted from the indicated table.

DELETE FROM <TABLE>

Alternatively, 'TRUNCATE TABLE' will also delete all data from <TABLE>. This is typically faster than 'DELETE'.

TRUNCATE TABLE <TABLE>

Record deletion with 'DELETE' can use the 'WHERE' clause to delete records conditionally. Instead of deleting all records from a table, records are deleted only if the indicated conditions are satisfied.

DELETE FROM <TABLE>

WHERE <CONDITIONAL\_COL> <OPERATOR> <CONDITIONAL\_VALUE>

Note that 'TRUNCATE' does not allow 'WHERE' clauses.

**FUNCTIONS**

Conditional statements can be applied to conditionally execute the functions in the queries below.

Return the smallest value record in the indicated column

SELECT MIN(<COL>)

FROM <TABLE>

Return the largest value record in the indicated column

SELECT MAX(<COL>)

FROM <TABLE>

Return the number of records in the indicated column

SELECT COUNT(<COL>)

FROM <TABLE>

* If <COL> is ‘\*’, then null values will be counted
* If <COL> is defined, the null values will NOT be counted

Return the average of values in the indicated column

SELECT AVG(<COL>)

FROM <TABLE>

Returns the sum of values in the indicated column

SELECT SUM(<COL>)

FROM <TABLE>

Generate concatenated fields by listing fields and separators as inputs

SELECT CONCAT(<COL\_0>, <SEPARATOR>, <COL\_1>, <SEPARATOR>, <COL\_2)

FROM <TABLE>

CONCAT\_WS function behaves similarly to CONCAT, but it automatically recognizes the first input as the separator to be applied between each field that follows

SELECT CONCAT\_WS(<SEPARATOR>, <COL\_0>, <COL\_1>, <COL\_2>)

FROM <TABLE>

**ALIAS**

Aliases are temporary names that are assigned to tables or columns within a given query, using the 'AS' keyword. This will not affect the database architecture, and is particularly useful for consolidating subqueries (AKA inner queries) when performing nested or complex queries. This can drastically improve query readability.

Encapsulate a query in <ALIAS>

SELECT <ALIAS>.<COL>

FROM <TABLE> AS <ALIAS>

* See how it simplifies the query code in the 'JOIN' section

Create a new data column titled <ALIAS> comprised of <COL> values

SELECT <COL> AS <ALIAS>

FROM <TABLE>

Create a new data column titled <ALIAS> comprised of <COL\_0> and <COL\_1> values

SELECT <COL\_0> + <COL\_1> AS <ALIAS>

FROM <TABLE>

Create a new data column titled <ALIAS> to store an aggregate function value

SELECT <FUNCTION>(<COL>) AS <ALIAS>

FROM <TABLE>

As a personal preference, I try to be as explicit as possible when referencing table elements to reduce the likelihood of a query error. Depending on the element I'm calling in my query, I usually follow the following convention:

**<SCHEMA\_NAME>.<TABLE\_NAME>.<COLUMN\_NAME>**

However, this convention can be cumbersome to write/read, so aliasing encapsulates the entire element path in a temporary variable name.

**TOP/LIMIT/FETCH FIRST**

This is an example of SQL syntax variations. Generally referred to as 'TOP', this clause is used with the 'SELECT' statement to identify the first N records in a table. This is particularly useful when querying larger data sets and only wanting to view small sample of the data.

'LIMIT' and 'FETCH FIRST' perform the same action as 'TOP', but their syntaxes vary by the version of SQL implemented in a database.

**TOP**

SELECT TOP <N> <COLS>

FROM <TABLE>

**LIMIT**

SELECT <COLS>

FROM <TABLE>

LIMIT <N>

**FETCH FIRST**

SELECT <COLS>

FROM <TABLE>

FETCH FIRST <N> ROWS ONLY

These clauses can also be used with other SQL operations to achieve different views of data. For example, the last N records in a table can be queried by applying the 'ORDER BY' operation with the 'DESC' sort keyword to apply the 'TOP' clause:

**TOP**

SELECT TOP <N> <COLS>

FROM (

SELECT <COLS>

FROM <TABLE>

ORDER BY <SORT\_COL> DESC)

**LIMIT**

SELECT <COLS>

FROM <TABLE>

ORDER BY <SORT\_COL> DESC

LIMIT <N>

**FETCH FIRST**

SELECT <COLS>

FROM <TABLE>

ORDER BY <SORT\_COL> DESC

FETCH FIRST <N> ROWS ONLY

The 'TOP' and 'FETCH FIRST' clauses also offers the 'PERCENT' keyword, which enables a query to select the top N percent of records in a table.

**TOP**

SELECT TOP <N> PERCENT <COLS>

FROM <TABLE>

**LIMIT**

*not available*

**FETCH FIRST**

SELECT <COLS>

FROM <TABLE>

FETCH FIRST <N> PERCENT ROWS ONLY

**RANK**

The 'RANK' function ranks data based on an indicated value field. It requires an 'OVER' clause, which specifies a <RANKING\_SCHEME> that defines how the ranking should be performed. The <RANKING\_SCHEME> is performed on a defined field and outputs a rank value in rank column titled <ALIAS> for every record.

SELECT <COLS>,

RANK() OVER (<RANKING\_SCHEME>) AS <ALIAS>

FROM <TABLE>

Multiple 'RANK' functions can be implemented in a single query by comma-separating them as individual columns in a query output.

SELECT <COLS>,

RANK() OVER (<RANKING\_SCHEME>) AS <ALIAS>,

RANK() OVER (<RANKING\_SCHEME>) AS <ALIAS>

FROM <TABLE>

The 'RANK' function adds a rank values column to the quieried table indicating a record's ranking in the data according to the indicated <RANKING\_SCHEME>, while 'ORDER BY' presents the data according to the <SORT\_COL> and <SORT\_KEYWORD>. However, it is acceptable to define the <RANKING\_SCHEME> with an 'ORDER\_BY'.

SELECT <COLS>,

RANK() OVER (ORDER BY <SORT\_COL> <SORT\_KEYWORD>) AS <ALIAS>

FROM <TABLE>

'RANK' can also use the 'PARTITION BY' keyword in the 'OVER' clause, which ranks records among an indicated subset of records. In other words, the records are grouped into subsets by the unique values in the indicated <PART\_COL> (similar to using a 'WHERE' clause), and the <RANKING\_SCHEME> is iteratively applied to each subset.

SELECT <COLS>,

RANK() OVER (PARTITION BY <PART\_COL> <RANKING\_SCHEME>) AS <ALIAS>

FROM <TABLE>

The 'RANK' function proves utile in data exploration when retrieving top or bottom values (highest values, lowest values) more efficiently than layered 'ORDER BY', 'GROUP BY', and 'TOP'/'LIMIT'/'FETCH FIRST' can.

Note that 'RANK' will give tied values the same rank value, but will skip however many ranking values were tied when indexing the output values.

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **employee** | **age** | **rank** |
| 1 | Ben Smith | 42 | 1 |
| 2 | Sam Gomez | 34 | 2 |
| 3 | Alex Tan | 34 | 2 |
| 4 | Sarah Lewis | 24 | 4 |

**ROW\_NUMBER**

The 'ROW\_NUMBER' function is similar to 'RANK' in that values are ranked among all records in the data set, but it assigns a row number indicating its placement in a ranked list instead of its actual ranking value. Like 'RANK', 'ROW\_NUMBER' requires an OVER clause to define the <RANKING\_SCHEME> and an <ALIAS> to title the ranking column in the query output.

SELECT <COLS>,

ROW\_NUMBER() OVER (<RANKING\_SCHEME>) AS <ALIAS>

FROM <TABLE>

If there are tied value in the <RANKING\_SCHEME> column, 'ROW\_NUMBER' will assign a ranking row number at random, which can be problematic.

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **employee** | **age** | **rank** |
| 1 | Ben Smith | 42 | 1 |
| 2 | Sam Gomez | 34 | 2 |
| 3 | Alex Tan | 34 | 3 |
| 4 | Sarah Lewis | 24 | 4 |

A solution is to add a secondary ranking scheme that distinguishes tied records. An example with 'ORDER BY' as the <RANKING\_SCHEME> demonstrates a possible solution.

SELECT <COLS>,

ROW\_NUMBER() OVER (ORDER BY <SORT\_COL\_0> <SORT\_KEYWORD>, <SORT\_COL\_1> <SORT\_KEYWORD>) AS <ALIAS>

FROM <TABLE>

Note that attempting this method for breaking ties with 'RANK' will be inaccurate, because 'RANK' will assign rankings based on the combination of the 'ORDER BY' parameters. Rank already handles tied values.

The 'PARTITION BY' is keyword is available for the 'ROW\_NUMBER' clause.

**DENSE\_RANK**

The 'DENSE\_RANK' method combines the best of 'RANK' and 'ROW\_NUMBER' in that it ranks values according to the defined <RANK\_SCHEME>, accounts for tied values, but does not skip assigned ranking values like 'RANK' inherently does.

SELECT <COLS>,

DENSE\_RANK() OVER (<RANKING\_SCHEME>) AS <ALIAS>

FROM <TABLE>

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **employee** | **age** | **rank** |
| 1 | Ben Smith | 42 | 1 |
| 2 | Sam Gomez | 34 | 2 |
| 3 | Alex Tan | 34 | 2 |
| 4 | Sarah Lewis | 24 | 3 |

The 'PARTITION BY' is keyword is available for the 'DENSE\_RANK' clause.

**JOIN**

**Join Clauses:**

* **[INNER] JOIN**:  All records with matching values in the left and right tables

A blue and black circle with a blue oval

AI-generated content may be incorrect.

* **FULL JOIN**:  All records where there is a match in either the left or right table

A diagram of a table

AI-generated content may be incorrect.

* **LEFT JOIN**:  All records from the left table, and matching records from the right table

A blue circle with black text

AI-generated content may be incorrect.

* **RIGHT JOIN**:  All records from the right table, and matching records from the left table

A blue circles with black text

AI-generated content may be incorrect.

SELECT <COLS>

FROM <LEFT\_TABLE>

<JOIN\_CLAUSE> <RIGHT\_TABLE>

ON <LEFT\_TABLE>.<JOIN\_KEY>=<RIGHT\_TABLE>.<JOIN\_KEY>

Multiple joins can be performed in a single query. The columns in the 'SELECT' statement must source from one of the JOIN'd tables in the query statement. A 'JOIN' can be performed between a newly JOIN'd table and a previously JOIN'd table in a query statement with multiple joins.

SELECT <COLS>

FROM <LEFT\_TABLE>

<JOIN\_CLAUSE> <RIGHT\_TABLE>

ON <LEFT\_TABLE>.<JOIN\_KEY>=<RIGHT\_TABLE>.<JOIN\_KEY> <JOIN\_CLAUSE> <RIGHT\_TABLE>

ON <LEFT\_TABLE>.<JOIN\_KEY>=<RIGHT\_TABLE>.<JOIN\_KEY>

<JOIN\_CLAUSE> <RIGHT\_TABLE>

ON <LEFT\_TABLE>.<JOIN\_KEY>=<RIGHT\_TABLE>.<JOIN\_KEY>

**JOIN ALIASING**

Sometimes multiple tables can have columns with similar names, in which case the columns indicated in the 'SELECT' statement will have to indicate the table from which the query will pull data. This can complicated the SQL syntax, so we will implement aliases to improve readability.

SELECT <ALIAS\_0>.<COSL>, <ALIAS\_1>.<COLS>, <ALIAS\_2>.<COLS>

FROM <LEFT\_TABLE> AS <ALIAS\_0>

<JOIN\_CLAUSE> <RIGHT\_TABLE> AS <ALIAS\_1>

ON <ALIAS\_0>.<JOIN\_KEY>=<ALIAS\_1>.<JOIN\_KEY>

<JOIN\_CLAUSE> <RIGHT\_TABLE> AS <ALIAS\_2>

ON <ALIAS\_0>.<JOIN\_KEY>=<ALIAS\_2>.<JOIN\_KEY>

When performing joins, it is good practice to explicitly indicate a column's source table in the 'SELECT' statement and 'JOIN' clause, with or without an alias. This improves the query's readability and reduces any chance for error.

**UNION**

The 'UNION' operator combines the results of two 'SELECT' operations into one query output. However, every 'SELECT' operation within the UNION must have the same number of fields, and each like-index field must have similar data types for a 'UNION' to be valid.

SELECT <TABLE\_0.COLS>

FROM <TABLE\_0>

UNION

SELECT <TABLE\_1.COLS>

FROM <TABLE\_1>

Clauses like 'WHERE' can be applied to can applied to the individual 'SELECT' operations when selecting data.

Keywords like 'ORDER BY' can be applied to the <UNION> to organize how the query output is presented.

**EXAMPLE:**

**TABLE\_0**

|  |  |  |
| --- | --- | --- |
| **id** | **student\_name** | **age** |
| 1 | Ben Smith | 42 |
| 2 | Sam Gomez | 34 |

**TABLE\_1**

|  |  |  |
| --- | --- | --- |
| **id** | **student\_name** | **phone\_numer** |
| 1 | Alex Tan | 555-222-2222 |
| 2 | Sarah Lewis | 555-333-3333 |

SELECT 'TABLE\_0' AS source\_table, STUDENT\_NAME

FROM TABLE\_0

UNION

SELECT 'TABLE\_1', STUDENT\_NAME

FROM TABLE\_0

WHERE STUDENT\_NAME NOT LIKE 'A%'

ORDER BY STUDENT\_NAME DESC

**QUERY OUTPUT**

|  |  |
| --- | --- |
| **source\_table** | **student\_name** |
| TABLE\_1 | Sarah Lewis |
| TABLE\_0 | Sam Gomez |
| TABLE\_0 | Ben Smith |

**ROLLBACK**

Sometimes an unintentional transaction (created, updated, or deletion entry, etc.) can be made to a database. SQL has the option to rollback transactions that are not committed, but the database must first be configured to do so.

SQL sets auto-commit to 'on' by default, so this can be turned off with the following command:

SET AUTOCOMMIT = OFF;

This change to the database now requires users to manually commit changes when a transaction is performed. Transactions can be committed with the following command:

COMMIT;

If a transaction needs to be undone before it is committed, a user can return to the last commit point with the following command:

ROLLBACK;

The database will be restored to its last commit point.

**Helpful Resources**

* [SQL in Easy Steps](https://a.co/d/aonx2jc) - Mike McGrath
* [W3Schools](https://www.w3schools.com/sql/default.asp)
* [Practice Exercises](https://www.w3schools.com/sql/exercise.asp?filename=exercise_select1)
* [Free Online Database Sandbox](https://sqlplayground.app/)