SQL Notes

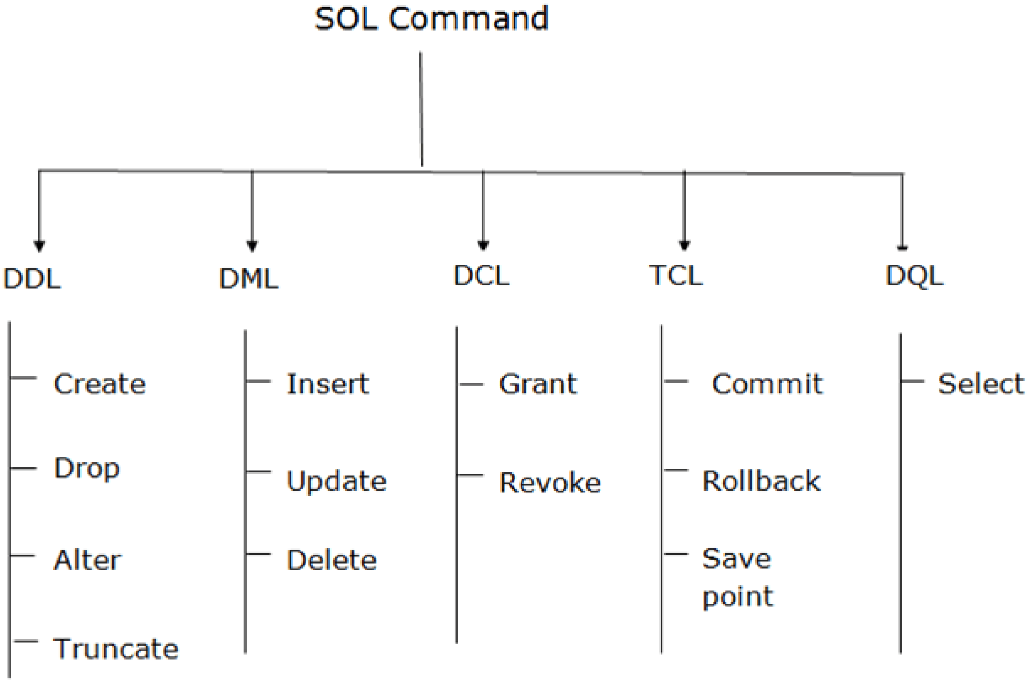
Contents

1 Pre-requsite Concepts

1.1 Types of SQL Commands

SQL commands are mainly categorized into four categories as:

1. DDL – Data Definition Language
2. DQl – Data Query Language
3. DML – Data Manipulation Language
4. DCL – Data Control Language
5. TCL – Transaction Control Language



1.2 ACID (SQL)

1. **Atomicity** – All operations in a transaction succeed or every operation is rolled back.
2. **Consistency** – On the completion of a transaction, the database is structurally sound.
3. **Isolation** – Transactions do not contend with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially.
4. **Durability** – The results of applying a transaction are permanent, even in the presence of failures.

1.3 Primary and Foreign Keys

A primary key is a value (typically a number or a string) that uniquely identifies a record. In many applications, primary keys are generated by the system when a record is created (e.g., sequentially or randomly); they are not usually set by users.

A foreign key is a column or group of columns in a relational database table that provides a link between data in two tables. It acts as a cross-reference between tables because it references the primary key of another table, thereby establishing a link between them

The more ids required to get to a piece of data, the more options you have in partitioning the data. The fewer ids required to get a piece of data, the easier it is to consume your system's output.

Watch out for what kind of information you encode in your ids, explicitly and implicitly. Clients may use the structure of your ids to de-anonymize private data, crawl your system in unexpected ways (auto-incrementing ids are a typical sore point), or perform a host of other attacks.

1.4 Normalized Data

Normalized Data is structured in such a way that there is no redundancy or duplication. In a normalized database, when some piece of data changes, you only need to change it in one place, not many copies in many different places.

1.5 Derived Data, Joins, Materialized Views

A derived dataset is created from some other data through a repeatable process, which you could run again if necessary. Usually, derived data is needed to speed up a particular kind of read access to the data. Indexes, caches, and materialized views are examples of derived data.

A join brings together records that have something in common. Most commonly used in the case where one record has a reference to another (a foreign key, a document reference, an edge in a graph) and a query needs to get the record that the reference points to.

To materialize means to perform a computation preemptively and write out its result, as opposed to calculating it on demand when requested.

1.6 Index and Secondary Index

Indexing is a way to get an unordered table into an order that will maximize the query's efficiency while searching. An index creates a data structure, typically a B-tree, with values from a specific column. It may then use a search algorithm, typically binary search, to find the value and return a pointer to its true index in the unordered table.

A secondary index is an additional data structure that is maintained alongside the primary data storage and which allows you to efficiently search for records that match a certain kind of condition.

1.7 Distributed Databases

See Distributed Systems Notebook[1].

[1] https://github.com/lukepereira/notebooks

2 Reading Data

2.1 SELECT Statement

2.1.1 Select all

```sql
SELECT *
FROM students;
```

2.1.2 Select specific columns

```sql
SELECT name, age
FROM students;
```

2.1.3 De-duplicate rows

```sql
SELECT DISTINCT name
FROM students;
```

2.1.4 Aliasing

In the SELECT block, <expression> AS <alias> provides an alias that can be referred to later in the query. This saves us from having to write out long expressions again, and can clarify the purpose of the expression.

```sql
SELECT name, age, location AS personal_info
FROM students;
```

2.2 WHERE, Conditions and Comparators

2.2.1 Numeric comparison

```sql
=     /* equals */
<     /* less-than */
<=    /* less-than or equals */
>     /* greater-than */
>=    /* greater-than or equals */
!=    /* does not equal */
<>    /* does not equal */
```

2.2.2 String comparison

Note that string values in the condition should be put between single quotes. Also note that any uppercase letter is smaller (i.e. comes before) any lowercase letter.

```
=     /* equals */
<     /* before in dictionary order */
<=    /* before or the same */
>     /* after in dictionary order */
>=    /* after or the same */
!=    /* does not equal */
<>    /* does not equal */
```

2.2.3 AND and OR

Complex clauses can be made out of simple ones using Boolean operators like NOT, AND and OR. SQL gives most precedence to NOT and then AND and finally OR. But if you're too lazy to remember the order of precedence, you can use parenthesis to clarify the order you want.

```
SELECT product
FROM inventory
WHERE amount < 5
OR name = 'paper'
AND price > 1;
```

When using both AND and OR it's important to know that AND has higher precedence.

If we want the OR to be evaluated first, we can use brackets ( ).

```
SELECT product
FROM inventory
WHERE (amount < 5
OR name = 'paper')
AND price > 1;
```

2.2.4 In a given list of values

```
SELECT product
FROM inventory
WHERE amount IN (1, 5, 10);
```

2.2.5 Inclusive ranges

```sql
SELECT product
FROM inventory
WHERE amount BETWEEN 5 AND 9;
```

is the equivalent condition to

```sql
SELECT product
FROM inventory
WHERE amount >= 5 AND amount <= 9
```

2.2.6 Exists, Any, All

```sql
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

```sql
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
(SELECT column_name FROM table_name WHERE condition);
```

```sql
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
(SELECT column_name FROM table_name WHERE condition);
```

2.3 String Pattern Matching

2.3.1 Quotes

In SQL, strings are denoted by single quotes. Backticks can be used to denote column and table names. This is useful when the column or table name is the same as a SQL keyword and when they have a space in them.

2.3.2 LIKE string operators

The most powerful string operators is probably LIKE. It allows us to use wildcards such as % and _ to match various characters. For instance, first_name LIKE ’%roy’ will return true for rows with first names ‘roy’, ‘Troy’, and ‘Deroy’ but not ‘royman’. The wildcard _ will match a single character so first_name LIKE ’_roy’ will only match ‘Troy’.

```sql
SELECT first_name, last_name, ex_number
FROM executions
WHERE first_name = 'Raymond'
  AND last_name LIKE '%Landry%'
```

2.4 Operations on Retrieved Records

2.4.1 Sort rows

```sql
SELECT name, age
FROM friends
ORDER BY name, age DESC, last_name ASC;
```

2.4.2 Limit the number of rows

```sql
SELECT name, grade
FROM course_grades
ORDER BY grade DESC
LIMIT 5;
```

2.4.3 Filter rows

```sql
SELECT product
FROM inventory
WHERE code = 'ABC123';
```

2.5 Aggregate Functions

To aggregate means to combine multiple elements into a whole. Similarly, aggregation functions take multiple rows of data and combine them into one number.

2.5.1 The COUNT Function

COUNT(<column>) returns the number of non-null rows in the column.

```sql
SELECT COUNT(first_name) FROM executions
```

2.5.2 NULLS

In SQL, NULL is the value of an empty entry. This is different from the empty string ‘’ and the integer 0, both of which are not considered NULL. To check if an entry is NULL, use IS and IS NOT instead of = and !=.

What if we don’t know which columns are NULL-free? Worse still, what if none of the columns are NULL-free? The solution is COUNT(). This is reminiscent of SELECT where the × represents all columns. In practice COUNT() counts rows as long as any one of their columns is non-null. This helps us find table lengths because a table shouldn’t have rows that are completely null.

```
SELECT COUNT(*) FROM executions
```

2.5.3 Subsets

Another common variation is to count a subset of the table. For instance, counting Harris county executions.

```
SELECT COUNT(*) FROM executions WHERE county='Harris'
```

2.5.4 Cumulative Sum

Write a query to get cumulative cash flow for each day such that we end up with a table in the form below:

```
/*
| date         | cumulative_cf |
|------------|---------------|
| 2018-01-01 | -1000         |
| 2018-01-03 | -1050         |
| ...          | ...            |
*/
SELECT
    a.date date,
    SUM(b.cash_flow) as cumulative_cf
FROM
    transactions a
JOIN b
    transactions b ON a.date >= b.date
GROUP BY
    a.date
ORDER BY
    date ASC
/*
    Alternate solution using a window function (more efficient!)
```

```
*/

SELECT
    date,
    SUM(cash_flow) OVER (ORDER BY date ASC) as cumulative_cf
FROM
    transactions
ORDER BY
    date ASC
```

2.5.5 CASE WHEN

What if we want to simultaneously find the number of Bexar county executions? The solution is to apply a CASE WHEN block which acts as a big if-else statement.

```
CASE
    WHEN <clause> THEN <result>
    WHEN <clause> THEN <result>
    ...
    ELSE <result>
END
```

A common mistake is to miss out the END command and the ELSE condition which is a catchall in case all the prior clauses are false. Also recall that clauses are expressions that can be evaluated to be true or false. This makes it important to think about the boolean value of whatever you stuff in there.

```
SELECT
    SUM(CASE WHEN county='Harris' THEN 1
        ELSE 0 END),
    SUM(CASE WHEN county='Bexar' THEN 1
        ELSE 0 END)
FROM executions
```

```
SELECT
    COUNT(CASE WHEN county='Harris' THEN 1
        ELSE NULL END),
    COUNT(CASE WHEN county='Bexar' THEN 1
        ELSE NULL END)
FROM executions
```

1. With a WHERE block,
2. With a COUNT and CASE WHEN block,
3. With two COUNT functions.

The WHERE version had it filter down to a small table first before aggregating while in the other two, it had to look through the full table. In the COUNT + CASE WHEN version, it only had to go through once, while the double COUNT version made it go through twice. So even though the output was identical, the performance was probably best in the first and worse in the other versions.

2.5.6 The GROUP BY Block

The GROUP BY block allows us to split up the dataset and apply aggregate functions within each group, resulting in one row per group. Its most basic form is GROUP BY <column>, <column>, ... and comes after the WHERE block.

```sql
SELECT
  county,
  COUNT(*) AS county_executions
FROM executions
GROUP BY county
```

Filtering via the WHERE block happens before grouping and aggregation. This is reflected in the order of syntax. After all, the WHERE block always precedes the GROUP BY block.

We generally do not want to mix aggregated and non-aggregated columns. The difference here is that grouping columns are the only columns allowed to be non-aggregate. After all, all the rows in that group must have the same values on those columns so there's no ambiguity in the value that should be returned.

2.5.7 The HAVING Block

What happens if we want to filter on the result of the grouping and aggregation? Surely we can't jump forward into the future and grab information from there. To solve this problem, we use HAVING.

We need an additional filter—one that uses the result of the aggregation. This means it cannot exist in the WHERE block because those filters are run before aggregation. You can think of it as a post-aggregation WHERE block.

```sql
SELECT county
FROM executions
WHERE ex_age >= 50
GROUP BY county
HAVING COUNT(*) > 2
```

2.5.8 Averages

Note: you can compose functions.

```
SELECT AVG(LENGTH(last_statement)) FROM executions
```

2.5.9 Rolling Average

To write a query to get 7-day rolling (preceding) average of daily sign ups.

```
/*
| date         | sign_ups  |
|------------|----------|
| 2018-01-01 | 10        |
| 2018-01-02 | 20        |
| 2018-01-03 | 50        |
| ...         | ...       |
| 2018-10-01 | 35        |
*/

SELECT
  a.date,
  AVG(b.sign_ups) average_sign_ups
FROM
  signups a
JOIN
  signups b
  ON a.date <= b.date + interval '6␣days'
  AND a.date >= b.date
GROUP BY
  a.date
```

2.5.10 Percentage

To calculate percentages, we need two separate queries, one which aggregates with a GROUP BY (to get the numerator) and another that aggregates without (to get the denominator).

```
SELECT
  county,
  100.0 * COUNT(*) / (SELECT COUNT(*) FROM executions)
    AS percentage
FROM executions
GROUP BY county
ORDER BY percentage DESC
```

2.5.11 MoM Percent Change

Find the month-over-month percentage change for monthly active users (MAU).

```
/*
| user_id | date       |
|---------|------------|
| 1       | 2018-07-01 |
| 3       | 2018-07-02 |
| ...     | ...        |
| 234     | 2018-10-04 |
*/

WITH mau AS
(
  SELECT
    DATE_TRUNC('month', date) month_timestamp,
    COUNT(DISTINCT user_id) mau
  FROM
    logins
  GROUP BY
    DATE_TRUNC('month', date)
  )
 SELECT
    a.month_timestamp previous_month,
    a.mau previous_mau,
    b.month_timestamp current_month,
    b.mau current_mau,
    ROUND(100.0*(b.mau - a.mau)/a.mau,2) AS percent_change
 FROM
    mau a
 JOIN
    mau b ON a.month_timestamp = b.month_timestamp
        - interval '1␣month'
```

2.5.12 Retained Users Per Month

```
SELECT
    DATE_TRUNC('month', a.date) month_timestamp,
    COUNT(DISTINCT a.user_id) retained_users
 FROM
    logins a
 JOIN
    logins b ON a.user_id = b.user_id
        AND DATE_TRUNC('month', a.date) =
        DATE_TRUNC('month', b.date)  + interval '1␣month'
 GROUP BY
    date_trunc('month', a.date)
```

2.6 Joining Tables

2.6.1 Nested Queries

We can combine multiple queries using a technique called “nesting”. The parentheses are important for demarcating the boundary between the inner query and the outer one:

```sql
SELECT first_name, last_name
FROM executions
WHERE LENGTH(last_statement) =
    (SELECT MAX(LENGTH(last_statement))
     FROM executions)
```

2.6.2 MapReduce

An interesting addendum is that we’ve actually just learned to do MapReduce in SQL, i.e. how to map various operations out to all the rows. For example, SELECT LENGTH(last_statement) FROM executions maps the length function out to all the rows.

2.6.3 Union

The UNION operator is used to combine the result-set of two or more SELECT statements. Each SELECT statement within UNION must have the same number of columns. The columns must also have similar data types. The columns in each SELECT statement must also be in the same order. The UNION operator selects only distinct values by default.

To allow duplicate values, use UNION ALL.

```sql
SELECT column_name(s) FROM table_name1
UNION
SELECT column_name(s) FROM table_name2

SELECT column_name(s) FROM table_name1
UNION ALL
SELECT column_name(s) FROM table_name2
```

2.6.4 Joins

If we wish to combine data from multiple queries but our desired table has the same length as the original executions table, we can rule out aggregations which produce a smaller table.

JOIN creates a big combined table which is then fed into the FROM block just like any other table. The big idea behind JOINs is to create an augmented table because the original doesn’t contain the information we need.

This is a powerful concept because it frees us from the limitations of a single table and allows us to combine multiple tables in potentially complex ways. We've also seen that with this extra complexity, meticulous bookkeeping becomes important. Aliasing tables, renaming columns and defining good JOIN ON clauses are all techniques that help us maintain order.

```
SELECT
  last_ex_date AS start,
  ex_date AS end,
  ex_date - last_ex_date AS day_difference
FROM executions
JOIN previous
  ON executions.ex_number = previous.ex_number
ORDER BY day_difference DESC
LIMIT 10
```

```
SELECT
  last_ex_date AS start,
  ex_date AS end,
  JULIANDAY(ex_date) - JULIANDAY(last_ex_date)
    AS day_difference
FROM executions
JOIN previous
  ON executions.ex_number = previous.ex_number
ORDER BY day_difference DESC
LIMIT 5
```

The query above is also notable because the clause executions.ex_number = previous.ex_number uses the format <table>.<column> to specify columns. This is only necessary here because both tables have a column called ex_number.

The JOIN block takes the form of <table1> JOIN <table2> ON <clause>. The clause works the same way as in WHERE <clause>. That is, it is a statement that evaluates to true or false, and anytime a row from the first table and another from the second line up with the clause being true, the two are matched.

The JOIN command defaults to performing what is called an "inner join" in which unmatched rows are dropped.

To preserve all the rows of the left table, we use a LEFT JOIN in in place of the vanilla JOIN. The empty parts of the row are left alone, which means they evaluate to NULL.

The RIGHT JOIN can be used to preserve unmatched rows in the right table, and the OUTER JOIN can be used to preserve unmatched rows in both.

The final subtlety is handling multiple matches. The join creates enough rows of executions so that each matching row of duplicated_previous gets its own partner. In this way, joins can create tables that are larger than the their constituents.

2.6.5 Self Joins

Remember to use aliases to form the column names (ex_number, last_ex_date). Notice that we are using a table alias here, naming the result of the nested query "previous".

```sql
SELECT
  last_ex_date AS start,
  ex_date AS end,
  JULIANDAY(ex_date) - JULIANDAY(last_ex_date) AS day_difference
FROM executions
JOIN (
    SELECT
      ex_number + 1 AS ex_number,
      ex_date AS last_ex_date
    FROM executions
  ) previous
  ON executions.ex_number = previous.ex_number
ORDER BY day_difference DESC
LIMIT 10
```

"previous" is derived from executions, so we're effectively joining executions to itself. This is called a "self join" and is a powerful technique for allowing rows to get information from other parts of the same table.

We've created the previous table to clarify the purpose that it serves. But we can actually write the query more elegantly by joining the executions table directly to itself. Note that we still need to give one copy an alias to ensure that we can refer to it unambiguously.

```sql
SELECT
  previous.ex_date AS start,
  executions.ex_date AS end,
  JULIANDAY(executions.ex_date) - JULIANDAY(previous.ex_date)
    AS day_difference
FROM executions
JOIN executions previous
  ON executions.ex_number = previous.ex_number + 1
ORDER BY day_difference DESC
LIMIT 10
```

2.6.6 Multiple Join Conditions

Write a query to get the response time per email (id) sent to zach@g.com. Do not include ids that did not receive a response from zach@g.com. Assume each email thread has a unique subject. Keep in mind a thread may have multiple responses back-and-forth between zach@g.com and another email address.

```
/*
| id | subject  | from         | to           | timestamp      |
|----|----------|--------------|--------------|----------------|
| 1  | Yosemite | zach@g.com   | thomas@g.com | 01-02 12:45:03 |
| 2  | Yosemite | thomas@g.com | zach@g.com   | 01-02 16:35:04 |
| .. | ..       | ..           | ..           | ..             |
*/
SELECT
    a.id,
    MIN(b.timestamp) - a.timestamp as time_to_respond
FROM
    emails a
JOIN
    emails b
        ON
            b.subject = a.subject
        AND
            a.to = b.from
        AND
            a.from = b.to
        AND
            a.timestamp < b.timestamp
 WHERE
    a.to = 'zach@g.com'
 GROUP BY
    a.id
```

2.6.7 Cross-Joins

The CROSS JOIN is used to combine each row of the first table with each row of the second table. It is also known as the Cartesian join since it returns the Cartesian product of the sets of rows from the joined tables.

Say we have a table state_streams where each row is a state and the total number of hours of streaming from a video hosting service. We want to write a query to get the pairs of states with total streaming amounts within 1000 of each other.

```
SELECT
    a.state as state_a,
    b.state as state_b
 FROM
    state_streams a
 CROSS JOIN
    state_streams b
 WHERE
    ABS(a.total_streams - b.total_streams) < 1000
    AND
    a.state <> b.state
```

2.7 Window Function

A window function performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities.

2.7.1 Get the ID with the highest value

Write a query to get the empno with the highest salary. Make sure your solution can handle ties!

```
/*
  depname   | empno | salary |
-----------+-------+--------+
 develop   |    11 |   5200 |
 ...
*/
WITH max_salary AS (
    SELECT
        MAX(salary) max_salary
    FROM
        salaries
    )
SELECT
    s.empno
FROM
    salaries s
JOIN
    max_salary ms ON s.salary = ms.max_salary
```

2.8 Materialized Views

In a materialized view, data is being persisted into a virtual table which is maintained by the SQL Server. Views can be used to encapsulate and index commonly used queries or precompute aggregations in order to improve read performance.

The database engine recreates the data, using the view's SQL statement, every time a user queries a view. Though this can negatively impact write performance because with each operation, the engine will have to update all of the relevant views.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

3 Writing Data

3.1 Databases

3.1.1 Create database

```
CREATE DATABASE database_name
```

3.1.2 Delete database

```
DROP DATABASE database_name
```

3.2 Tables

3.2.1 Data types

Some common data types for columns of a table:

- CHAR(size)
- VARCHAR(size)
- BINARY(size)
- TEXT(size)
- BLOB(size)
- ENUM(val1, val2, val3, ...)
- SET(val1, val2, val3, ...)
- BIT(size)
- BOOL or BOOLEAN
- INT(size) or INTEGER(size)
- FLOAT(size, d)
- DATE
- DATETIME(fsp)
- TIMESTAMP(fsp)
- YEAR

3.2.2 Constraints

SQL constraints are used to specify rules for data in a table and can be specified either when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

- NOT NULL – Ensures that a column cannot have a NULL value
- UNIQUE – Ensures that all values in a column are different
- PRIMARY KEY – A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY – Uniquely identifies a row/record in another table
- CHECK – Ensures that all values in a column satisfies a specific condition
- DEFAULT – Sets a default value for a column when no value is specified

- INDEX – Used to create and retrieve data from the database very quickly

3.2.3 Creating tables

Each column in a database table is required to have a name and a data type.

```
CREATE TABLE pet (name VARCHAR(20), owner VARCHAR(20),
       species VARCHAR(20), sex CHAR(1), birth DATE, death DATE)
```

3.2.4 Alter table

```
ALTER TABLE table_name
ADD column_name datatype

/* or */

ALTER TABLE table_name
DROP COLUMN column_name
```

3.2.5 Deleting tables

```
DELETE FROM table_name
WHERE some_column=some_value

/* Delete full table */
DELETE FROM table_name
DELETE * FROM table_name
```

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ....
);
```

3.3 Indexes

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries

Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

Some recommendations involving indexes are:

- disable constraints and indexes during bulk loads.
- avoid indexes on columns with low selectivity.
- use partial indexes.
- index columns with high correlation using BRIN (Block Range Index)

3.3.1 B-Trees

A B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. B-trees and B+trees are commonly used to create indexes in relational databases.

See Data Structures and Algorithms Notebook [2] for more details.

3.3.2 Create index

By default, duplicate values are allowed. Use UNIQUE keyword if duplicate values are not allowed.

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

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

3.3.3 Delete index

```
/* SQL Server */
DROP INDEX table_name.index_name;

/* MySQL */
ALTER TABLE table_name
DROP INDEX index_name;
```

3.4 Records

3.4.1 Insert Records

It is possible to write the INSERT INTO statement in two ways.

[2] https://github.com/lukepereira/notebooks

The first way specifies both the column names and the values to be inserted. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query.

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);

INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

The next time you load data into a table, think about how the data is going to be queried, and make sure you sort it in a way that indexes used for range scan can benefit from.

3.4.2 Update Records

UPDATE is a relatively expensive operation. To speed up an UPDATE command it's best to make sure you only update what needs updating.

```
/* Likely slower */
UPDATE users SET email = lower(email);

/* Likely faster */
UPDATE users SET email = lower(email)
WHERE email != lower(email);
```

3.4.3 Delete Records

```
DELETE FROM table_name WHERE condition;
```

3.4.4 Copy with select and insert

Copy all columns into an existing table.

```
INSERT INTO table2
SELECT * FROM table1
WHERE condition;
```

The SELECT INTO statement copies data from one table into a new table.

```
SELECT *
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

4 Transactions

Transactional control commands are only used with the DML Commands such as INSERT, UPDATE and DELETE. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

- COMMIT – to save the changes.
- ROLLBACK – to roll back the changes.
- SAVEPOINT – creates points within the groups of transactions in which to ROLLBACK.
- SET TRANSACTION – Places a name on a transaction.

4.1 COMMIT

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.

```sql
DELETE FROM CUSTOMERS
   WHERE AGE = 25;
COMMIT;
```

4.2 ROLLBACK

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

```sql
DELETE FROM CUSTOMERS
   WHERE AGE = 25;
ROLLBACK;
```

4.3 SAVEPOINT

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

```sql
SAVEPOINT SP1;
/* Savepoint created. */
DELETE FROM CUSTOMERS WHERE ID=1;
/* 1 row deleted. */
SAVEPOINT SP2;
/* Savepoint created. */
DELETE FROM CUSTOMERS WHERE ID=2;
/* 1 row deleted. */
```

```
ROLLBACK TO SP2;
```

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

4.4 SET TRANSACTION

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows. For example, you can specify a transaction to be read only or read write.

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

References

[1] https://selectstarsql.com/

[2] https://www.w3schools.com/sql

[3] https://www.tutorialspoint.com/sql/sql-transactions.htm

[4] https://quip.com/2gwZArKuWk7W

[5] https://hakibenita.com/sql-tricks-application-dba

[6] https://use-the-index-luke.com/
http://docshare01.docshare.tips/files/29375/293750304.pdf