Performance issues

Create index

- CREATE INDEX test1_id_index ON test1 (id);
- CREATE INDEX test1_id_index ON test1 USING btree (id);

- CREATE INDEX test1_id_index ON test1 [USING btree] (id) WHERE <condition>;

➔ Partial index

Index types in PostgreSQL

- B-Tree (default)
  - handle equality and range queries on data that can be sorted into some ordering.
  - Operators: <, ≤, = , ≥, > , LIKE (col LIKE **'foo%'** but not col LIKE '%bar')
  - Sorted output
- Hash index: can only handle simple equality comparisons
- GiST index: for several two-dimensional geometric data types,
  - not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented
- GIN index
  - inverted indexes which can handle values that contain more than one key, arrays for example

Multicolumn index

- CREATE INDEX test2_mm_idx ON test2 (major, minor);

- B-Tree
- GiST index
- GIN index

file:///C:/Program%20Files/PostgreSQL/9.4/doc/postgresql/html/sql-createindex.html

file:///C:/Program%20Files/PostgreSQL/9.4/doc/postgresql/html/indexes.html

Examining index usage

- EXPLAIN [ANALYZE] [VERBOSE] *statement*
  - EXPLAIN *statement:* displays the execution plan that the PostgreSQL planner generates for the supplied statement.

    Actually two numbers are shown: the start-up cost before the first row can be returned, and the total cost to return all the rows.
  - **VERBOSE** option: displays additional information regarding the plan (output column list, table and function names, …)

Examining index usage

- EXPLAIN [ANALYZE] [VERBOSE] *statement*
  - **ANALYZE** option: causes the statement to be actually executed, not only planned, actual runtime statistics are added to the display
- **Important:** If you wish to use EXPLAIN ANALYZE on an INSERT, UPDATE, DELETE, CREATE TABLE AS, or EXECUTE statement without letting the command affect your data, use this approach:

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

View table indexes

- \d table_name
- Ex.: \d customers

Tips

- Select fewer columns to improve hash join performance
- Index the *independent* **where** predicates to improve hash join performance

Tips

- Having a **WHERE / HAVING** clause in your queries does not necessarily means that it is a bad query
- Only retrieve the data you need
  - Inner join vs exists (with subqueries)
  - Select **DISTINCT** : try to avoid if you can
  - **LIKE** operator: the index isn't used if the pattern starts with % or _

- Limit your results : LIMIT, TOP
- Don’t Make Queries More Complex Than They Need To Be
  - OR / IN / UNION ?
  - OR operator : index is not used except composite index ➔ IN/UNION/OUTER JOIN
  - NOT operator: index is not used => avoid
  - AND vs BETWEEN
  - ANY / ALL: index not used => max , min ,…
  - Isolate columns in Condition : age + 7 < 20 ➔ age < 13

- No Brute force
  - JOIN clause:
    - Order of tables => biggest table: placed last in join
    - No redundant conditions on joins
  - Having clause:
    - Used only if needed
    - Not to replace WHERE => WHERE help to limit the intermediate number of records

➔ Need smart indexing, smart using