Create Index

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON [ ONLY ]
table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass [ (
opclass_parameter = value [, ... ] ) ] ] [ ASC | DESC ] [ NULLS { FIRST | LAST }
] [, ...] )
    [ INCLUDE ( column_name [, ...] ) ]
    [ NULLS [ NOT ] DISTINCT ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

Experiment 1: the size of index

1. Import data my_flight from `cs307_public_flights.sql`
2. Create an index

flights

```
select * from my_flight where duration = 505;
```

```
create index fight_index on my_flight (duration);
```

3. You can query the size of this table or you can query the size of the table's index

```
select pg_size_pretty(pg_table_size('my_flight'));
select pg_size_pretty(pg_indexes_size('my_flght'));
select pg_size_pretty(pg_total_relation_size('my_flight'));
```

Tips: Consider the storage size of index, create suitable index.

4. Use `DROP INDEX` to remove an index.

EXPLAIN

EXPLAIN — show the execution plan of a statement

This command displays the execution plan that the PostgreSQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc. — and if multiple tables are referenced, what join algorithms will be used to bring together the required rows from each input table.

The most critical part of the display is the estimated statement execution cost, which is the planner's guess at how long it will take to run the statement (measured in cost units that are arbitrary, but conventionally mean disk page fetches). 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.

The structure of a query plan is a tree of *plan nodes*. Nodes at the bottom level of the tree are scan nodes: they return raw rows from a table. There are different types of scan nodes for different table access methods: sequential scans, index scans, and bitmap index scans.

The `ANALYZE` option causes the statement to be actually executed, not only planned. Then actual run time statistics are added to the display, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned. This is useful for seeing whether the planner's estimates are close to reality.

Basic Search

Import `film.sql`

Experiment 2: Sequence Scan

```
explain select * from movies;
```

Result:

	QUERY PLAN
1	Seq Scan on movies (cost=0.00..169.38 rows=9538 width=31)

Total evaluation time cost = seq_page_cost + cpu_tuple_cost. The time cost in seq is about 1.0, while time cost in cpu is about 0.01(millisecond). Suppose the data from 9538 rows in table movies are distributed in 74 disk page, and the total time cost would be **74 * 1.0 + 9538 * 0.01 = 169.38**

Adding where condition

```
explain select * from movies where movieid<6000;
```

Result:

	QUERY PLAN
1	Seq Scan on movies (cost=0.00..193.23 rows=6000 width=31)
2	Filter: (movieid < 6000)

<6000没有使用index scan:
数据库会自动判断使用哪种查询方式更快
movies共9204行 6000是一个相对很大的范围 不如选择直接遍历

Experiment 3: Index Scan

Using index in where condition

```
explain select * from movies where movieid<200;
```

Result:

0.29 使用index开启耗时 12.79 查询耗时

	QUERY PLAN
1	Index Scan using movies_pkey on movies (cost=0.29..12.79 rows=200 width=31)
2	Index Cond: (movieid < 200)

Experiment 4: Bitmap Index scan

```
explain select * from my_flight where duration = 105;
```

Result:

```
QUERY PLAN
Bitmap Heap Scan on my_flight  (cost=32.55..917.53 rows=1581 width=59)
  Recheck Cond: (duration = 105)
  ->  Bitmap Index Scan on fight_index  (cost=0.00..32.15 rows=1581 width=0)
        Index Cond: (duration = 105)
```

Join

Experiment 5： Nest Loop

```
explain select *
        from movies m
          join
          countries c2 on m.country = c2.country_code
        where c2.country_code = 'cn';
```

Result:

```
QUERY PLAN
Nested Loop  (cost=0.00..199.68 rows=214 width=49)
  ->  Seq Scan on countries c2  (cost=0.00..4.31 rows=1 width=18)
        Filter: (country_code = 'cn'::bpchar)
  ->  Seq Scan on movies m  (cost=0.00..193.23 rows=214 width=31)
        Filter: (country = 'cn'::bpchar)
```

Experiment 6: Hash Join

```
explain select *
        from movies m
          join credits c
            on m.movieid = c.movieid
        where c.credited_as = 'D' and m.movieid < 200;
```

Result:

```
QUERY PLAN
Hash Join  (cost=15.29..901.94 rows=188 width=41)
  Hash Cond: (c.movieid = m.movieid)
  ->  Seq Scan on credits c  (cost=0.00..863.08 rows=8980 width=10)
        Filter: (credited_as = 'D'::bpchar)
  ->  Hash  (cost=12.79..12.79 rows=200 width=31)
        ->  Index Scan using movies_pkey on movies m  (cost=0.29..12.79 rows=200 width=31)
              Index Cond: (movieid < 200)
```

Multi-key indexes

Experiment 7:

Step 1

```
create table my_movies
  as
    select *
   from movies;
```

Step 2

```
create index movies_multi_index on my_movies(movieid,year_released, runtime);
```

Step 3 Compare with following queries

```
explain select * from my_movies where movieid=20;
explain select * from my_movies where movieid<100 and year_released=2000;
explain select * from my_movies where year_released=2000;  没有使用index
explain select * from my_movies where runtime=200; 没有使用index
```

Create index on function

Experiment 8:

Step 1

```
create index movies_title_index on my_movies(title);
```

Step 2 Compare following two queries, whether index is effected.

```
explain select * from my_movies where title = 'Armaan';
explain select * from my_movies where upper(title) = 'ARMAAN';
                                                    upper()之后不匹配了
```

Step 3 Create index on upper() function

```
create index movies_upper_title_index on my_movies(upper(title));
```

Step 4 whether index is effected in follow query

```
explain select * from my_movies where upper(title) = 'ARMAAN';
```