Define a new and remove a view

```
CREATE VIEW venue_view_2017 AS
        SELECT *
        FROM venue
        WHERE year = 2017;
```

```
DROP VIEW venue_view_2017;
```

For more information:

- https://www.postgresql.org/docs/current/static/sql-createview.html
- https://www.postgresql.org/docs/current/static/sql-dropview.html

View

The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

SELECT * **FROM** venue_view_2017;

```
id | name | year | school | volume | number | type
----+------+------+--------+--------+--------+------
(0 rows)
```

INSERT INTO venue **VALUES** (3150859, 'VLDB', 2017, '', '9', '12', 0);

SELECT * **FROM** venue_view_2017;

```
id      | name | year | school | volume | number | type
--------+------+------+--------+--------+--------+------
3150859 | VLDB | 2017 |        | 9      | 12     |    0
(1 row)
```

Materialized view

The query is executed and used to populate the view at the time the command is issued and may be refreshed later using REFRESH MATERIALIZED VIEW.

CREATE MATERIALIZED VIEW is similar to CREATE TABLE AS, except that it also remembers the query used to initialize the view, so that it can be refreshed later upon demand.

A materialized view has many of the same properties as a table, but there is no support for temporary materialized views or automatic generation of OIDs.

Define a new, remove and refresh a materialized view

```sql
CREATE MATERIALIZED VIEW coauthors_per_paper AS
        SELECT paperid AS paper, COUNT(authid) AS coauthors
        FROM paperauths
        GROUP BY paperid;
```

```sql
REFRESH MATERIALIZED VIEW coauthors_per_paper;
```

```sql
DROP MATERIALIZED VIEW coauthors_per_paper;
```

For more information:

- https://www.postgresql.org/docs/current/static/sql-creatematerializedview.html
- https://www.postgresql.org/docs/current/static/sql-refreshmaterializedview.html
- https://www.postgresql.org/docs/current/static/sql-dropmaterializedview.html

Materialized view

SELECT * **FROM** coauthors_per_paper **WHERE** paper = 342;

```
paper | coauthors
------+-----------
 342  |     2
(1 row)
```

INSERT INTO paperauths **VALUES** (342, 43);

SELECT * **FROM** coauthors_per_paper **WHERE** paper = 342;

```
paper | coauthors
------+-----------
 342  |     2
(1 row)
```

Materialized view (cont)

REFRESH MATERIALIZED **VIEW** coauthors_per_paper;

```
REFRESH MATERIALIZED VIEW
```

SELECT $*$ **FROM** coauthors_per_paper **WHERE** paper $=$ 342;

```
 paper | coauthors
-------+-----------
  342  |     3
(1 row)
```

Define a new table as

CREATE TABLE AS creates a table and fills it with data computed by a SELECT command.

The table columns have the names and data types associated with the output columns of the SELECT.

```
CREATE TABLE journal_articles AS
    SELECT p.*
    FROM papers p, venue v
    WHERE p.venue = v.id AND v.type = 0;
```

For more information:

- https://www.postgresql.org/docs/current/static/sql-createtableas.html

Temporary tables and views

Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see ON COMMIT below).

```
CREATE TEMP TABLE conference_and_workshop_papers AS
    SELECT p.*
    FROM papers p, venue v
    WHERE p.venue = v.id AND v.type = 1;
```

Temporary views are automatically dropped at the end of the current session. If any of the tables referenced by the view are temporary, the view is created as a temporary view.

```
CREATE TEMP VIEW books_and_thesis AS
    SELECT p.*
    FROM papers p, venue v
    WHERE p.venue = v.id AND v.type = 3;
```

For more information:

- https://www.postgresql.org/docs/current/static/sql-createtable.html
- https://www.postgresql.org/docs/current/static/sql-createview.html

Full-text search

Full Text Search provides the capability to identify natural-language documents that satisfy a query, and optionally to sort them by relevance to the query.

Differently from textual data types operators such as LIKE, full text indexing allows documents to be preprocessed and an index saved for later rapid searching. Preprocessing includes:

- Parsing documents into tokens.
- Converting tokens into lexemes.
- Storing preprocessed documents optimized for searching.

It's possible to perform full text search without an index.

For more information:

- `https://www.postgresql.org/docs/current/static/textsearch-intro.html`

Full-text search

SELECT id, name
FROM papers
WHERE to_tsvector('english', name) @@ to_tsquery('english', 'solve & problem')
LIMIT 7;

id	name
762788	Declaratively solving tricky Google Code Jam problems with Prolog–based ECLiPSe CLP system.
763157	Quantum Algorithms for many–to–one Functions to Solve the Regulator and the Principal Ideal Problem
763402	A Critique of "Solving the P/NP Problem Under Intrinsic Uncertainty".
763780	Solving the Parity Problem with Rule 60 in Array Size of the Power of Two.
764236	Towards Solving the Inverse Protein Folding Problem
764710	Induction of High–level Behaviors from Problem–solving Traces using Machine Learning Tools
764822	Abstract flows over time: A first step towards solving dynamic packing problems

For more information:

- https://www.postgresql.org/docs/current/static/textsearch-controls.html

Full-text search

SELECT to_tsquery('english', 'the & solve & problem');

```
      to_tsquery
---------------------
 'solv' & 'problem'
```

SELECT id, name
FROM papers
WHERE to_tsvector('english', name) @@ to_tsquery('english', 'the & solve & problem')
LIMIT 4;

```
  id    |                              name
--------+-----------------------------------------------------------------------------
767312 | A priori estimation of a time step for numerically solving parabolic problems.
767510 | A novel approach of solving the CNF-SAT problem.
767645 | Solving reviewer assignment problem in software peer review: An approach based on preference matrix
and asymmetric TSP model.
767967 | Ensuring Trust in One Time Exchanges: Solving the QoS Problem
```

For more information:

- https://www.postgresql.org/docs/current/static/textsearch-controls.html

Full-text indexes

GIN (Generalized Inverted Index)-based index, contain an index entry for each word (lexeme), with a compressed list of matching locations.

The column must be of tsvector type.

CREATE INDEX ON papers **USING** GIN (to_tsvector('english', name));

CREATE INDEX ON venue **USING** GIN (to_tsvector('english', name || ' ' || school));

For more information:

- `https://www.postgresql.org/docs/current/static/textsearch-indexes.html`
- `https://www.postgresql.org/docs/current/static/textsearch-tables.html#textsearch-tables-index`

PostgreSQL functions

SELECT replace('abcdefabcdef', 'cd', 'XX');

```
         replace
---------------
abXXefabXXef
(1 row)
```

SELECT unnest(ARRAY[1,2]);

```
unnest
--------
     1
     2
(2 rows)
```

For more information:

- `https://www.postgresql.org/docs/current/static/functions-string.html`
- `https://www.postgresql.org/docs/current/static/functions-array.html`

PostgreSQL functions (cont)

SELECT plainto_tsquery('english', 'The Fat Rats');

```
plainto_tsquery
-----------------
'fat' & 'rat'
(1 row)
```

SELECT ∗ **FROM** generate_series(2,4);

```
 generate_series
-----------------
               2
               3
               4
(3 rows)
```

For more information:

- https://www.postgresql.org/docs/current/static/functions-textsearch.html
- https://www.postgresql.org/docs/current/static/functions-srf.html

Table functions: crosstab

Produces a "pivot table" (that is, multiple rows) with the value columns specified by a second query.

crosstab function is part of a PostgreSQL extension called tablefunc. To call the crosstab function, you must first enable the tablefunc extension by executing the following SQL command: **CREATE extension tablefunc;**

```
SELECT *
FROM crosstab(
        'SELECT v.name, v.year, COUNT(p.id) AS papers
        FROM venue v, papers p
        WHERE v.id = p.venue
        GROUP BY v.name, v.year
        ORDER BY 1, 2',
        'SELECT DISTINCT year
        FROM venue
        ORDER BY 1 DESC
        LIMIT 6')
AS final_result(venue TEXT, "2016" INTEGER, "2015" INTEGER, "2014" INTEGER, "2013" INTEGER, "2012" INTEGER, "2011" INTEGER);
```

For more information:

- https://www.postgresql.org/docs/current/static/tablefunc.html

Table functions: crosstab

The first query parameter must be compliant with the following restrictions:

- The SELECT must return 3 columns.
- The first column in the SELECT will be the identifier of every row in the pivot table or final result. In our example, this is the venue's name.
- The second column in the SELECT represents the categories in the pivot table. In our example, these categories are the venue years.
- The third column in the SELECT represents the value to be assigned to each cell of the pivot table. In our example, these are the number of papers.

```
SELECT v.name, v.year, COUNT(p.id) AS papers
FROM venue v, papers p
WHERE v.id = p.venue
GROUP BY v.name, v.year
ORDER BY 1, 2
```

For more information:

- https://www.postgresql.org/docs/current/static/tablefunc.html

Table functions: crosstab

The second query parameter represents the complete list of categories, in this case the last 6 years:

```
SELECT DISTINCT year
FROM venue
ORDER BY 1 DESC
LIMIT 6
```

The crosstab function is invoked in the SELECT statement's FROM clause. We must define the names of the columns and data types that will go into the final result.

```
venue TEXT, "2016" INTEGER, "2015" INTEGER, "2014" INTEGER, "2013"
INTEGER, "2012" INTEGER, "2011" INTEGER
```

For more information:

- https://www.postgresql.org/docs/current/static/tablefunc.html

Table expressions: GROUPING SETS

Enables complex grouping operations.

The data selected by the FROM and WHERE clauses is grouped separately by each specified grouping set, aggregates computed for each group just as for simple GROUP BY clauses, and then the results returned.

SELECT v.name, v.**year**, **COUNT**(p.id) **AS** papers
FROM venue v, papers p **WHERE** v.id = p.venue
GROUP BY GROUPING SETS ((v.name), (v.**year**), ());

Each sublist of GROUPING SETS may specify zero or more columns or expressions and is interpreted the same way as though it were directly in the GROUP BY clause.

An empty grouping set means that all rows are aggregated down to a single group.

For more information:

- https://www.postgresql.org/docs/current/static/queries-table-expressions.html#queries-grouping-sets

Table expressions: ROLLUP

A shorthand notation for a common type of grouping set.

ROLLUP (e1, e2, e3, ...)

represents the given list of expressions and all prefixes of the list including the empty list. Thus it is equivalent to:

```
GROUPING SETS (
        (e1, e2, e3, ...),
        ...
        (e1, e2),
        (e1),
        ()
)
```

Commonly used for analysis over hierarchical data; e.g. total population by city, state, and country.
For more information:

- `https://www.postgresql.org/docs/current/static/queries-table-expressions.html#queries-grouping-sets`

Table expressions: CUBE

A shorthand notation for a common type of grouping set.

CUBE(e1, e2, e3)

represents the given list and all of its possible subsets (i.e. the power set). Thus it is equivalent to:

```
GROUPING SETS (
    ( e1, e2, e3),
    ( e1, e2 ),
    ( e1, e3),
    ( e1 ),
    ( e2, e3),
    ( e2 ),
    ( e3 ),
    ( )
)
```

For more information:

- https://www.postgresql.org/docs/current/static/queries-table-expressions.html#queries-grouping-sets