Things You Didn't Know PostgreSQL Could Do!

To say there are many features in PostgreSQL is to make a gross understatement. Which features can help you save time with your project? Which ones can help boost your application performance? Which ones are just cool to play with?

Combine multiple indexes

```
SELECT *
   FROM tab
   WHERE a = ?
         AND b = ?
```

A btree index on <a> and a btree index on <b> can be combined using a bitmap scan.

This isn't as efficient as an index on <a,b> or <b,a> but...

Combine multiple indexes

```
SELECT *
  FROM tab
  WHERE a < ?
        AND b < ?
```

In this case no two column btree index would suffice.

But this still requires scanning two large indexes and combining the results. Postgres will often choose not to do so since the second index may be creating more work than it's saving.

When is one index better than two?

```
SELECT *
   FROM tab
   WHERE a > 10
         AND b < 0
```

An index like:

```
CREATE INDEX magic ON tab (a) WHERE b < 0
```

Partial indexes are like having a hidden second index key. It's like two indexes in one. It's like having a second index on <b> without using any storage space at all.

When are many indexes better than one?

```
SELECT *
   FROM tickets
   WHERE status = ?
         AND last_update > now() - '5 days'

CREATE INDEX disappointing ON tickets (status, last_update)
```

status is probably too low cardinality to be very helpful. And it dramatically increases the size of the index for not much gain in selectivity.

When are many indexes better than one?

```
SELECT *
   FROM tickets
   WHERE status = ?
         AND last_update > now() - '5 days'
```

```
CREATE INDEX neat_1 ON tickets (last_update) WHERE status = 'NEW'
CREATE INDEX neat_2 ON tickets (last_update) WHERE status = 'ASSIGNED'
CREATE INDEX neat_3 ON tickets (last_update) WHERE status = 'PENDING'
CREATE INDEX neat_4 ON tickets (last_update) WHERE status = 'RESOLVED'
```

This takes more or less the same space as a single index on "last_update" and is effectively like having a bitmap index on status for free.

Postgres doesn't have pivot tables does it?

An often asked-for feature, they would let you do the equivalent of GROUP BY but get separate columns for each grouping key.

Postgres can't change the "shape" of the result based on the data so the set of columns has to be fixed.

But Postgres **does** have complex types.... Arrays provide a handy escape hatch:

Postgres doesn't have pivot tables does it?

What we want to do:

```
SELECT sum(revenue)
    FROM revenue_table
    GROUP BY month, country
    PIVOT BY country
```

	US	CA	UK
January	0	0	0
February	0	0	0
March	0	0	0
April	0	0	0

Postgres doesn't have pivot tables does it?

```
SELECT month,
            array_agg(ROW(country,rev))
               AS revenue_list
    FROM (
        SELECT month,country,
                    sum(revenue) as rev
               FROM revenue_table
             GROUP BY month, country
       ) AS row
 GROUP BY month order
```

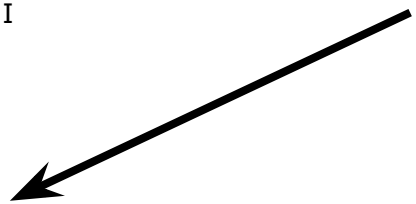
Postgres doesn't have pivot tables does it?

What we get isn't as pretty but from a client-side language which supports composite types it can actually be easier to work with than separate columns:

	revenue_list
January	{"(CA,0)","(US,0)","(UK,0)"}
February	{"(CA,0)","(US,0)","(UK,0)"}
March	{"(CA,0)","(US,0)","(UK,0)"}
April	{"(CA,0)","(US,0)","(UK,0)"}

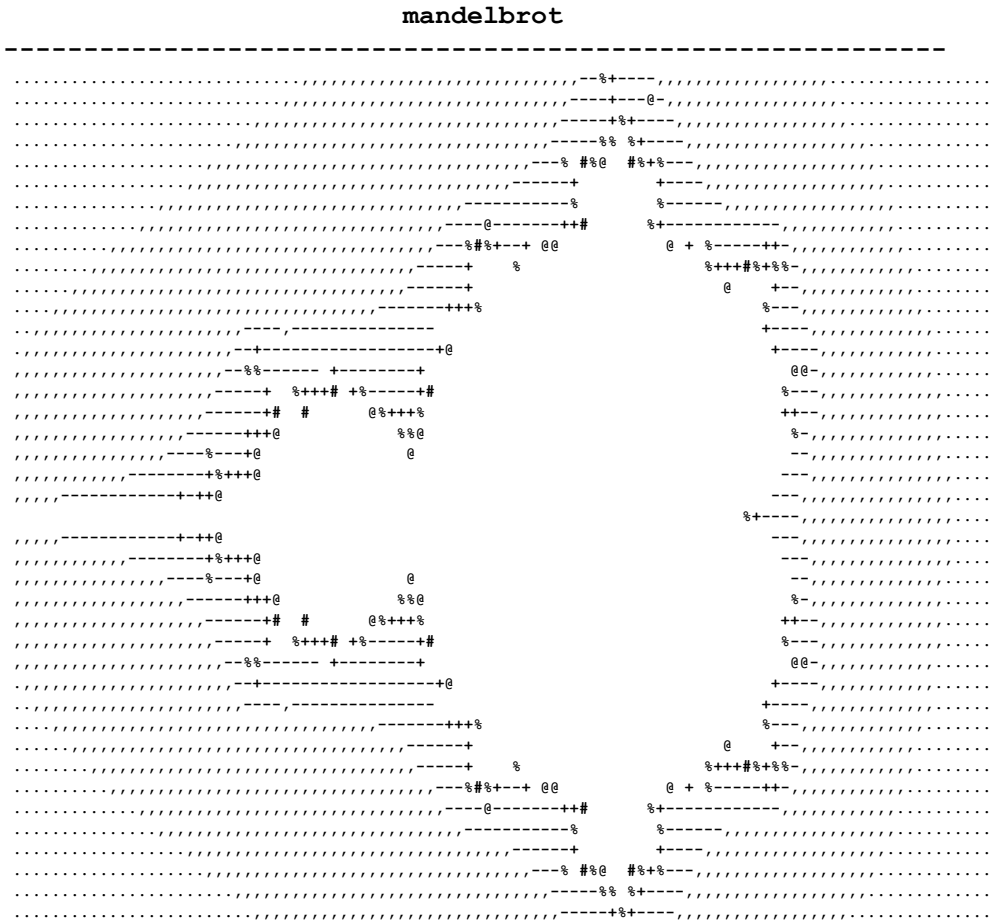
This query was translated from MS-SQL. The original used pivot tables.

```
WITH RECURSIVE
  x(i) AS ( VALUES(0) UNION ALL SELECT i + 1 FROM x WHERE i < 101),
 Z(Ix, Iy, Cx, Cy, X, Y, I) AS (
    SELECT Ix, Iy, X::float, Y::float, X::float, Y::float, 0
    FROM
       (SELECT -2.2 + 0.031 * i, i FROM x) AS xgen(x,ix)
    CROSS JOIN
       (SELECT -1.5 + 0.031 * i, i FROM x) AS ygen(y,iy)
    UNION ALL
    SELECT Ix, Iy, Cx, Cy, X * X - Y * Y + Cx AS X, Y * X * 2 + Cy, I + 1
    FROM Z
    WHERE X * X + Y * Y < 16.0
    AND I < 27
),
Zt (Ix, Iy, I) AS (
    SELECT Ix, Iy, MAX(I) AS I
    FROM Z
    GROUP BY Iy, Ix
    ORDER BY Iy, Ix
)
SELECT array_to_string(
    array_agg(
        SUBSTRING(
            ' .,,,-----++++%%%%@@@@#### ',
            GREATEST(I,1),
            1
        )
    ),''
)
FROM Zt
GROUP BY Iy
ORDER BY Iy;
```



It's actually simpler and resulting query is much shorter using array_agg() and array_to_string() instead.

This query was translated from MS-SQL. The original used pivot tables.



(44 rows)

Finding the median

A perennial question for smarties, Postgres doesn't have a great way to find the median but did you know the OFFSET and LIMIT clauses can take arbitrary sub expressions including subqueries?

```
SELECT *
   FROM tab
  ORDER BY height
 OFFSET (select count(*) from tab)/2
      LIMIT 1
```

Top-k queries and pagination

Similarly if you want to retrieve the top k results or results 11-20 for a page, let the database know it doesn't have to sort the entire result set:

```
SELECT *
    FROM tab
  WHERE ...
  ORDER BY k1,k2,k3
  OFFSET 11
  LIMIT 10
```

The database optimizes this and keeps only the top 20 results in memory at all times. This means it won't have to spill to disk to perform a batch disk sort even if the result set is huge.

Psql features you won't know how you lived without

```
postgres=# select * from pg_views limit 5;
 schemaname | viewname  | viewowner |
------------+-----------+-----------+----------------------------------------
 pg_catalog | pg_roles  | stark     | SELECT pg_authid.rolname, pg_authid.rolsup
 pg_catalog | pg_shadow | stark     | SELECT pg_authid.rolname AS usename, pg_au
 pg_catalog | pg_group  | stark     | SELECT pg_authid.rolname AS groname, pg_au
 pg_catalog | pg_user   | stark     | SELECT pg_shadow.usename, pg_shadow.usesys
 pg_catalog | pg_rules  | stark     | SELECT n.nspname AS schemaname, c.relname
(5 rows)
```

Psql features you won't know how you lived without

```
postgres=# \pset format wrapped
Output format is wrapped.
postgres=# select * from pg_views limit 5;
  schemaname | viewname  | viewowner |                                          definition
-------------+-----------+-----------+-----------------------------------------------------------------------------------
 pg_catalog  | pg_group  | stark     | SELECT pg_authid.rolname AS groname, pg_authid.oid AS grosysid, ARRAY(SELECT p
             |           |           |.embers.member FROM pg_auth_members WHERE (pg_auth_members.roleid = pg_authid.oid
             |           |           |.grolist FROM pg_authid WHERE (NOT pg_authid.rolcanlogin);
 pg_catalog  | pg_user   | stark     | SELECT pg_shadow.usename, pg_shadow.usesysid, pg_shadow.usecreatedb, pg_shadow
             |           |           |.r, pg_shadow.usecatupd, pg_shadow.userepl, '********'::text AS passwd, pg_shadow
             |           |           |.til, pg_shadow.useconfig FROM pg_shadow;
 pg_catalog  | pg_rules  | stark     | SELECT n.nspname AS schemaname, c.relname AS tablename, r.rulename, pg_get_rul
             |           |           |.id) AS definition FROM ((pg_rewrite r JOIN pg_class c ON ((c.oid = r.ev_class)))
             |           |           |.JOIN pg_namespace n ON ((n.oid = c.relnamespace))) WHERE (r.rulename <> '_RETURN
             |           |           |.e);
(5 rows)
```

Interesting uses of transactional DDL

```
postgres=# BEGIN;
BEGIN
postgres=# create index on gianttable (status);
CREATE INDEX
postgres=# analyze gianttable;
ANALYZE
postgres=# explain select * from gianttable where status = 'FOO';
                              QUERY PLAN
-----------------------------------------------------------------------------------
 Bitmap Heap Scan on gianttable  (cost=4.30..13.76 rows=6 width=44)
   Recheck Cond: ((status)::text = 'FOO'::text)
   ->  Bitmap Index Scan on gianttable_status_idx  (cost=0.00..4.30 rows=6 width=0)
         Index Cond: ((status)::text = 'FOO'::text)
(4 rows)

postgres=# rollback;
ROLLBACK
postgres=# explain select * from gianttable where status = 'FOO';
                       QUERY PLAN
-------------------------------------------------------------
 Seq Scan on gianttable  (cost=0.00..23.75 rows=6 width=44)
   Filter: ((status)::text = 'FOO'::text)
(2 rows)
```

Lastly, psql features you won't know how you lived without

```
postgres=# \set AUTOCOMMIT off
postgres=# \set ON_ERROR_ROLLBACK on
```

This lets you perform complex updates, inserts, deletes, then perform selects to verify the data matches expectations, then commit or rollback before the changes are visible to users.

It also lets you hit C-c on a slow operation or correct a typo without losing all the work done in that transaction so far.

Caveat, it doesn't play nicely with some non-transactional commands like VACUUM and CREATE INDEX CONCURRENTLY.