psql essentials

```
\l                    list databases        \c db     connect
\dt  \di  \dv  \df    tables / indexes / views / functions
\d+ name              describe, incl. indexes & storage
\du  roles            \dn  schemas          \dx  extensions
\dp name              privileges of a table
\x                    expanded output (toggle)
\timing               show per-query time
\e                    edit the query in $EDITOR
\i file.sql           run a SQL file
\copy t TO 'out.csv' CSV HEADER    client-side export
\set VERBOSITY verbose show SQLSTATE with every error
\watch 2              re-run the last query every 2s
```

What's running

```
SELECT pid, state, now() - query_start AS runtime,
       wait_event_type, left(query, 80) AS query
FROM pg_stat_activity
WHERE state <> 'idle'
ORDER BY query_start;
```

```
-- Who blocks whom:
SELECT w.pid AS waiting, b.pid AS blocking,
       left(b.query, 60) AS blocking_query
FROM pg_stat_activity w
JOIN pg_stat_activity b
  ON b.pid = ANY (pg_blocking_pids(w.pid));
```

`state = 'idle in transaction'` + old `xact_start` = trouble. More: show running queries.

Cancel & kill

```
SELECT pg_cancel_backend(pid);     -- cancel the query, keep
the session
SELECT pg_terminate_backend(pid);  -- close the whole
connection
```

Escalate in that order. Never `kill -9` a backend from the shell: the whole cluster restarts. Details: kill a query safely.

Timeouts

```
SET statement_timeout = '30s';   -- total runtime cap (incl.
lock waits)
SET lock_timeout = '3s';         -- max wait for a lock (great
for DDL)
SET idle_in_transaction_session_timeout = '5min';

BEGIN;
SET LOCAL statement_timeout = '30min';  -- this transaction
only
-- ... long migration or report ...
COMMIT;
```

When they fire: statement timeout · lock timeout.

Sizes

```
SELECT pg_size_pretty(pg_total_relation_size('t')); --
heap+TOAST+indexes
SELECT pg_size_pretty(pg_table_size('t'));          -- without
indexes
SELECT pg_size_pretty(pg_indexes_size('t'));        -- indexes
only
SELECT pg_size_pretty(pg_database_size(current_database()));
```

Top-20 tables query and the fine print: table & index sizes. Size ≠ live data — see bloat.

Maintenance

```
ANALYZE t;                          -- refresh planner
statistics
VACUUM (VERBOSE, ANALYZE) t;        -- reclaim dead rows + stats
CREATE INDEX CONCURRENTLY idx ON t (col);  -- no write lock
REINDEX INDEX CONCURRENTLY idx;

-- wraparound watch (alert long before ~2 billion):
SELECT datname, age(datfrozenxid)
FROM pg_database ORDER BY 2 DESC;
```

Background: VACUUM, autovacuum and bloat · emergency: wraparound.

EXPLAIN

```
EXPLAIN SELECT ...;                    -- plan only, does not
run
EXPLAIN (ANALYZE, BUFFERS) SELECT ...; -- RUNS the query, real
timings

-- writes, safely:
BEGIN; EXPLAIN (ANALYZE) UPDATE ...; ROLLBACK;
```

Paste the output (text or JSON) into the EXPLAIN visualizer for a flame graph and automatic warnings.

Backup & restore

```
pg_dump -Fc dbname > db.dump        # custom format: compressed,
pg_restore -j4 -d dbname db.dump    #   parallel restore with -j
pg_dump -Fc -t mytable dbname > table.dump
pg_dumpall --globals-only > roles.sql   # roles: NOT in pg_dump
psql dbname < plain.sql             # restore plain-SQL dumps
```

`pg_dump` is consistent without blocking writes (it runs in a Repeatable Read transaction).

Privileges (typical app role)

```
GRANT USAGE ON SCHEMA app TO app_user;
GRANT SELECT, INSERT, UPDATE, DELETE
  ON ALL TABLES IN SCHEMA app TO app_user;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA app TO app_user;

-- future tables too (created by the migration role):
ALTER DEFAULT PRIVILEGES FOR ROLE migrator IN SCHEMA app
  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO app_user;
```

Grants are not retroactive-forward — the classic trap. More: permission denied.

Configuration

```
SHOW work_mem;
SELECT name, setting, source FROM pg_settings
WHERE source <> 'default';        -- what has been customized

ALTER SYSTEM SET work_mem = '64MB';
SELECT pg_reload_conf();          -- some params need a restart instead
```

`work_mem` is per sort/hash node, per query — it multiplies. `max_connections` and `shared_buffers` need a restart.

Replication quick checks

```
SELECT pg_is_in_recovery();            -- am I a replica?
SELECT client_addr, state, replay_lag
FROM pg_stat_replication;              -- on the primary

-- slots pinning WAL (active = f is a red flag):
SELECT slot_name, active, pg_size_pretty(
  pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)) AS retained
FROM pg_replication_slots;
```

Forgotten slots fill disks: No space left on device · replica query cancels: conflict with recovery.

Locking & transactions

```
BEGIN ISOLATION LEVEL SERIALIZABLE;  -- retry on SQLSTATE 40001
SELECT ... FOR UPDATE;               -- lock the rows you will update
SELECT ... FOR UPDATE SKIP LOCKED;   -- job queues without contention
SELECT ... FOR UPDATE NOWAIT;        -- fail fast instead of waiting
SHOW default_transaction_isolation;  -- what you actually run at
```

Which level guarantees what, and how to retry correctly: isolation levels in practice · deadlock detected.

Connections

```
SHOW max_connections;
SELECT count(*) FROM pg_stat_activity;
SELECT usename, application_name, count(*)
FROM pg_stat_activity GROUP BY 1, 2 ORDER BY 3 DESC;

ALTER ROLE app_user CONNECTION LIMIT 50;   -- cap per role
```

Slots exhausted? The durable fix is a pooler, not a bigger limit: too many clients.

Common SQLSTATEs

`40P01` deadlock detected · `40001` serialization failure · `23505` duplicate key · `53300` too many clients · `57014` query canceled · `55P03` lock timeout · `42P01` relation does not exist · `42501` permission denied · `25P02` transaction aborted · `22P02` invalid input syntax

All 30, with causes and fixes: error reference. In psql, `\set VERBOSITY verbose` shows the code with each error.