PostgreSQL Syntax: A Concise but Complete Practical Introduction

(for study use)

September 28, 2025

Contents

1 Mindset & Scope **2**

2 Data Types (core + PG extras) **2**

3 Schemas, Databases, Search Path **2**

4 DDL: Tables, Constraints, Identity, Generated Columns **3**
- 4.1 Create tables . . . 3
- 4.2 Altering & dropping . . . 3
- 4.3 Indexes (btree default) and advanced types . . . 3

5 DML: INSERT, UPDATE, DELETE, MERGE, UPSERT **3**

6 DQL: SELECT Core **4**
- 6.1 Projection, filtering, ordering, limits . . . 4
- 6.2 Joins . . . 4
- 6.3 Aggregates & GROUP BY . . . 4
- 6.4 Window functions (analytic) . . . 4
- 6.5 Subqueries & CTEs . . . 5
- 6.6 Recursive CTE . . . 5

7 Arrays & JSON (Postgres superpowers) **5**
- 7.1 Arrays . . . 5
- 7.2 JSONB . . . 5

8 Range Types & Exclusion Constraints **5**

9 Full-Text Search (FTS) **6**

10 Views & Materialized Views **6**

11 Sequences & Identity **6**

12 Constraints: DEFERRABLE & Triggers **6**

13 Partitioning (range, list, hash) **6**

14 Transactions (TCL), Isolation, Locks **7**
- 14.1 Basics . . . 7
- 14.2 Isolation levels . . . 7
- 14.3 Explicit locks (use sparingly) . . . 7

15 Security (DCL): Roles, Privileges, Row-Level Security **7**

16 Query Planning & Performance Hygiene **7**

17 Common Table Patterns & Idioms **8**
17.1 Upsert with conflict target expression . . . 8
17.2 Keyset pagination . . . 8
17.3 Pivot-ish aggregates . . . 8
17.4 Distinct-on (PG feature) . . . 8

18 Advanced Joins & Set Operations **8**

19 Error Handling & Constraints in Practice **8**

20 Triggers & PL/pgSQL (sketch) **8**

21 Extensions & FDW **9**

22 Temporal Patterns **9**

23 Admin Odds & Ends (psql meta, safe defaults) **9**

24 Common Pitfalls (Nitpicks) **9**

25 Minimal End-to-End Example **9**

26 Where this fits in a curriculum **10**

27 Further Study Pointers (conceptual) **10**

1 Mindset & Scope

This guide targets the high–value subset of PostgreSQL syntax you will actually use. It groups syntax by responsibility:

- **DDL** (define structure): `CREATE/ALTER/DROP`, types, constraints, indexes, partitioning, schemas.
- **DML** (change data): `INSERT/UPDATE/DELETE/MERGE`.
- **DQL** (query data): `SELECT`, joins, subqueries, CTE, window functions, aggregates.
- **TCL** (transactions): `BEGIN/COMMIT/ROLLBACK`, savepoints, isolation.
- **DCL** (control): roles, privileges, `GRANT/REVOKE`.

Postgres–specifics: **arrays**, **JSON**, **range types**, **full–text search**, **generated columns**, **identity/sequence**, **UPSERT**, **recursive CTEs**, **indexes** (btree/hash/brin/gin/gist), **partitioning**, **locks**, **PL/pgSQL**, **extensions**.
Caveat: "Complete" SQL is unbounded (extensions, planner hints, FDWs). This is the core, with idiomatic examples and pitfalls.

2 Data Types (core + PG extras)

- Numeric: `smallint`, `int`, `bigint`, `decimal(p,s)`, `numeric`, `real`, `double precision`, `serial/bigserial (legacy)`.
- Text/byte: `text`, `varchar(n)`, `char(n)`, `bytea`.
- Temporal: `date`, `time`, `timestamp [with(out) time zone]`, `interval`.
- Boolean: `boolean`.
- UUID, `json`, `jsonb`, `xml`, `cidr`, `inet`, `macaddr`, `money`.
- Arrays: `int[]`, `text[]`, multidimensional `int[][]`.
- Ranges: `int4range`, `int8range`, `numrange`, `tsrange`, `tstzrange`, `daterange`.
- Composite & enum: `CREATE TYPE`.

3 Schemas, Databases, Search Path

```
-- create and use a schema
CREATE SCHEMA app AUTHORIZATION app_user;
SET search_path TO app, public; -- resolves unqualified names
```

Pitfall: if you rely on `public`, lock down `CREATE` privileges for security.

4 DDL: Tables, Constraints, Identity, Generated Columns

4.1 Create tables

```
CREATE TABLE app.users (
  user_id bigserial PRIMARY KEY, -- legacy; prefer identity
  email text NOT NULL UNIQUE,
  full_name text,
  is_active boolean NOT NULL DEFAULT true,
  created_at timestamptz NOT NULL DEFAULT now(),
  -- identity (SQL standard)
  v2_id bigint GENERATED ALWAYS AS IDENTITY,
  -- generated column (stored)
  email_domain text GENERATED ALWAYS AS (split_part(email, '@', 2)) STORED,
  -- FK with actions
  org_id bigint REFERENCES app.orgs(org_id)
              ON UPDATE CASCADE ON DELETE SET NULL,
  -- check constraint
  CONSTRAINT email_ok CHECK (position('@' IN email) > 1)
);
```

Notes: prefer `IDENTITY` over `serial`; generated columns require `STORED`.

4.2 Altering & dropping

```
ALTER TABLE app.users ADD COLUMN tags text[] DEFAULT '{}'::text[];
ALTER TABLE app.users ALTER COLUMN full_name SET NOT NULL;
ALTER TABLE app.users DROP COLUMN tags;
DROP TABLE IF EXISTS app.users CASCADE;
```

4.3 Indexes (btree default) and advanced types

```
-- btree for equality/range
CREATE INDEX ON app.users (email);
-- expression index: matches WHERE lower(email)=...
CREATE INDEX users_email_lower_idx ON app.users ((lower(email)));
-- partial index: sparse filtered
CREATE INDEX users_active_idx ON app.users (created_at) WHERE is_active;
-- unique multi-column
CREATE UNIQUE INDEX users_org_email_uniq ON app.users (org_id, email);

-- GIN for arrays|jsonb|full-text
CREATE INDEX users_tags_gin ON app.users USING GIN (tags);
-- JSONB path ops
CREATE INDEX users_profile_gin ON app.users USING GIN ((profile jsonb_path_ops));
-- GiST example (ranges, geo)
CREATE INDEX events_time_gist ON app.events USING GIST (when_range);
-- BRIN for very large, naturally clustered tables
CREATE INDEX logs_brin ON app.logs USING BRIN (ts);
```

Pitfall: `LIKE 'prefix%'` benefits from `btree` only with `text_pattern_ops` or `citext` / expression indexes (e.g., `lower()`).

5 DML: INSERT, UPDATE, DELETE, MERGE, UPSERT

```
-- INSERT
INSERT INTO app.users (email, full_name) VALUES
  ('a@x.com','A'), ('b@x.com','B')
RETURNING user_id;

-- INSERT .. SELECT
```

```sql
INSERT INTO app.audit(email, at) SELECT email, now() FROM app.users WHERE is_active;

-- UPSERT
INSERT INTO app.users (email, full_name)
VALUES ('a@x.com','New Name')
ON CONFLICT (email) DO UPDATE
  SET full_name = EXCLUDED.full_name, updated_at = now();

-- MERGE (PG15+)
MERGE INTO app.inventory AS t
USING (VALUES (1,'USB-C',10),(2,'HDMI',5)) AS s(id,name,qty)
ON t.id = s.id
WHEN MATCHED THEN UPDATE SET qty = t.qty + s.qty
WHEN NOT MATCHED THEN INSERT (id,name,qty) VALUES (s.id,s.name,s.qty);

-- UPDATE
UPDATE app.users SET is_active = false WHERE last_login < now() - interval '180 days'
RETURNING user_id;

-- DELETE
DELETE FROM app.users WHERE is_active = false AND created_at < now() - interval '1 year';
```

6 DQL: SELECT Core

6.1 Projection, filtering, ordering, limits

```sql
SELECT user_id, email, created_at
FROM app.users
WHERE is_active AND created_at >= date_trunc('month', now())
ORDER BY created_at DESC
LIMIT 20 OFFSET 0; -- prefer keyset pagination for scale
```

6.2 Joins

```sql
SELECT u.user_id, u.email, o.name AS org
FROM app.users u
JOIN app.orgs o ON o.org_id = u.org_id
LEFT JOIN app.subscriptions s ON s.user_id = u.user_id
WHERE s.active IS DISTINCT FROM false;
```

Nits: use `IS DISTINCT FROM` to handle NULL-safe equality.

6.3 Aggregates & GROUP BY

```sql
SELECT org_id, count(*) AS users, avg(age) AS avg_age
FROM app.users
GROUP BY org_id
HAVING count(*) > 10
ORDER BY users DESC;
```

6.4 Window functions (analytic)

```sql
SELECT
  user_id, email,
  row_number() OVER (PARTITION BY org_id ORDER BY created_at) AS rn,
  avg(age) OVER (PARTITION BY org_id) AS org_avg_age
FROM app.users;
```

Concept link: aggregation collapses rows; windows compute per-row analytics without collapsing.

6.5 Subqueries & CTEs

```
WITH active AS (
  SELECT user_id, org_id FROM app.users WHERE is_active
)
SELECT o.name, count(a.user_id)
FROM active a JOIN app.orgs o USING(org_id)
GROUP BY o.name;
```

Nitpick: CTEs are *optimization fences* before PG12; from PG12 they can inline. Avoid unnecessary CTEs in hot paths.

6.6 Recursive CTE

```
WITH RECURSIVE org_tree AS (
  SELECT org_id, parent_id, name, 1 AS lvl
  FROM app.orgs WHERE parent_id IS NULL
  UNION ALL
  SELECT c.org_id, c.parent_id, c.name, p.lvl+1
  FROM app.orgs c JOIN org_tree p ON c.parent_id = p.org_id
)
SELECT * FROM org_tree ORDER BY lvl, name;
```

7 Arrays & JSON (Postgres superpowers)

7.1 Arrays

```
-- literals and operators
SELECT ARRAY[1,2,3] @> ARRAY[2]; -- contains
SELECT 2 = ANY(ARRAY[1,2,3]); -- membership
-- unnest with ordinality for position-aware explode
SELECT email, tag, ord
FROM app.users u,
LATERAL unnest(u.tags) WITH ORDINALITY AS t(tag, ord);
```

Indexing: GIN on arrays supports `@>`, `&&`, etc.

7.2 JSONB

```
-- access
SELECT profile->>'lang' AS lang
FROM app.users
WHERE profile @> '{"marketing": {"opt_in": true}}';

-- jsonpath (PG12+)
SELECT jsonb_path_query_array(profile, '$.addresses[*].city')
FROM app.users;

-- GIN indexing for containment queries
CREATE INDEX users_profile_gin ON app.users USING GIN (profile jsonb_path_ops);
```

Tradeoff: prefer normalized schema for frequently filtered fields; use JSONB for sparse or evolving attributes.

8 Range Types & Exclusion Constraints

```
CREATE TABLE room_bookings(
  room_id int, during tstzrange NOT NULL,
  EXCLUDE USING GIST (room_id WITH =, during WITH &&) -- disallow overlaps
);
-- include/exclude bounds
```

```
INSERT INTO room_bookings VALUES (101, tstzrange(now(), now()+interval '2h','[)'));
```

9 Full-Text Search (FTS)

```
ALTER TABLE docs ADD COLUMN tsv tsvector;
UPDATE docs SET tsv = to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''));
CREATE INDEX docs_tsv_idx ON docs USING GIN (tsv);
SELECT * FROM docs WHERE tsv @@ plainto_tsquery('english', 'quick fox');
```

Note: maintain `tsv` via trigger for write paths.

10 Views & Materialized Views

```
CREATE VIEW app.active_users AS
  SELECT * FROM app.users WHERE is_active;

CREATE MATERIALIZED VIEW app.monthly_summary AS
  SELECT date_trunc('month', created_at) AS m, count(*) c
  FROM app.users GROUP BY 1;

REFRESH MATERIALIZED VIEW CONCURRENTLY app.monthly_summary; -- needs unique index
```

11 Sequences & Identity

```
CREATE SEQUENCE app.seq START 100 INCREMENT 10 OWNED BY app.users.user_id;
SELECT nextval('app.seq');

-- identity columns (preferred)
CREATE TABLE t(i int GENERATED BY DEFAULT AS IDENTITY);
```

12 Constraints: DEFERRABLE & Triggers

```
ALTER TABLE app.users
  ADD CONSTRAINT org_fk FOREIGN KEY (org_id) REFERENCES app.orgs(org_id)
  DEFERRABLE INITIALLY DEFERRED;

BEGIN;
SET CONSTRAINTS org_fk DEFERRED;
-- batch of rows becoming valid only at txn end
COMMIT;
```

13 Partitioning (range, list, hash)

```
CREATE TABLE events(
  id bigserial, ts timestamptz NOT NULL, payload jsonb
) PARTITION BY RANGE (ts);

CREATE TABLE events_2025_09 PARTITION OF events
  FOR VALUES FROM ('2025-09-01') TO ('2025-10-01');

-- indexing on parent propagates to partitions in PG11+
CREATE INDEX ON events (ts);
```

Use-case: time-series/logs; enables pruning and lighter maintenance.

14 Transactions (TCL), Isolation, Locks

14.1 Basics

```
BEGIN;
-- ... DML ...
SAVEPOINT s1;
-- ... attempt risky step ...
ROLLBACK TO SAVEPOINT s1;
COMMIT;
```

14.2 Isolation levels

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; -- default
-- REPEATABLE READ (no non-repeatable reads; phantom-free in PG)
-- SERIALIZABLE (true serializability; may abort on conflicts)
```

Pitfall: `SERIALIZABLE` can abort; retry logic required.

14.3 Explicit locks (use sparingly)

```
SELECT * FROM app.users WHERE user_id=42 FOR UPDATE; -- row lock
LOCK TABLE app.users IN SHARE MODE; -- table lock
```

15 Security (DCL): Roles, Privileges, Row-Level Security

```
-- roles
CREATE ROLE app_user LOGIN PASSWORD '...';
GRANT USAGE ON SCHEMA app TO app_user;
GRANT SELECT, INSERT, UPDATE ON ALL TABLES IN SCHEMA app TO app_user;

-- future-proof default privileges
ALTER DEFAULT PRIVILEGES IN SCHEMA app
  GRANT SELECT, INSERT, UPDATE ON TABLES TO app_user;

-- Row-Level Security
ALTER TABLE app.users ENABLE ROW LEVEL SECURITY;
CREATE POLICY same_org ON app.users
USING (org_id = current_setting('app.current_org')::int);
```

Link: RLS complements application auth; be explicit about `ALTER TABLE ... FORCE ROW LEVEL SECURITY`.

16 Query Planning & Performance Hygiene

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT ... ;

-- maintenance
VACUUM (ANALYZE); -- autovacuum usually handles this
SET work_mem='64MB'; -- per-sort/hash; beware overcommit
SET enable_seqscan=off; -- debugging only (do not use in prod)
```

Mental model: Pick *right data types*, *right indexes*, *right predicates*. Prefer keyset pagination over `OFFSET` for large pages.

17 Common Table Patterns & Idioms

17.1 Upsert with conflict target expression

```
CREATE UNIQUE INDEX users_email_lower ON app.users ((lower(email)));
INSERT INTO app.users(email) VALUES('A@X.com')
ON CONFLICT ((lower(email))) DO NOTHING;
```

17.2 Keyset pagination

```
SELECT * FROM app.users
WHERE created_at < $last_seen
ORDER BY created_at DESC
LIMIT 50;
```

17.3 Pivot-ish aggregates

```
SELECT org_id,
  count(*) FILTER (WHERE is_active) AS active,
  count(*) FILTER (WHERE NOT is_active) AS inactive
FROM app.users GROUP BY org_id;
```

17.4 Distinct-on (PG feature)

```
SELECT DISTINCT ON (org_id) org_id, user_id, created_at
FROM app.users
ORDER BY org_id, created_at DESC;
```

18 Advanced Joins & Set Operations

```
-- set ops: UNION [ALL], INTERSECT, EXCEPT
SELECT email FROM a
UNION
SELECT email FROM b;

-- lateral joins: use outputs of a subquery per-row
SELECT u.user_id, f.friend_id
FROM app.users u
LEFT JOIN LATERAL (
  SELECT friend_id FROM app.friends WHERE user_id=u.user_id ORDER BY since DESC LIMIT 5
) f ON true;
```

19 Error Handling & Constraints in Practice

- Prefer `NOT NULL` + defaults; `CHECK` for domain rules.
- Use `DEFERRABLE` when intra-batch temporary violations are expected.
- Validate emails/phones at app layer; use light DB checks only.

20 Triggers & PL/pgSQL (sketch)

```sql
CREATE OR REPLACE FUNCTION app.users_tsv_update()
RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN
  NEW.tsv := to_tsvector('english', coalesce(NEW.full_name,'')||' '||coalesce(NEW.email,''));
  RETURN NEW;
END $$;

CREATE TRIGGER users_tsv_trg
BEFORE INSERT OR UPDATE OF full_name, email ON app.users
FOR EACH ROW EXECUTE FUNCTION app.users_tsv_update();
```

Rule of thumb: keep business logic in app layer unless DB-centric.

21 Extensions & FDW

```sql
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS pg_trgm; -- trigram search
CREATE EXTENSION IF NOT EXISTS postgres_fdw;
```

22 Temporal Patterns

```sql
-- system-versioning pattern (manual):
valid_from timestamptz NOT NULL DEFAULT now(),
valid_to timestamptz,
EXCLUDE USING GIST (id WITH =, tstzrange(valid_from, coalesce(valid_to,'infinity')) WITH &&)
```

23 Admin Odds & Ends (psql meta, safe defaults)

```sql
-- psql (client) meta-commands (not SQL):
-- \d app.users describe
-- \dt app.* list tables
-- \di list indexes
-- \df list functions
```

Tip: in teaching labs, align with NUS/CMU-style DB courses: write `EXPLAIN ANALYZE` for each nontrivial query and justify index choice.

24 Common Pitfalls (Nitpicks)

- **NULL traps**: = vs `IS NULL`; use `IS DISTINCT FROM`.
- **Text & collation**: cross-locale comparisons may disable index usage; normalize with `lower()` + expression index or `citext`.
- **LIKE indexing**: *`%pattern`* (leading %) cannot use btree; consider trigram (`pg_trgm`).
- **Offset pagination**: avoid large `OFFSET`; use keyset.
- **CTE misuse**: avoid as "style"; prefer inline subqueries for performance unless readability & reuse dominate.
- **Serial vs identity**: prefer `GENERATED {ALWAYS|BY DEFAULT} AS IDENTITY`.
- **Transactions**: remember that `EXCEPTION` in PL/pgSQL subtransaction rolls back only to block.

25 Minimal End-to-End Example

```sql
BEGIN;
CREATE SCHEMA app;
CREATE TABLE app.orgs (
  org_id bigserial PRIMARY KEY,
  name text NOT NULL UNIQUE
```

```
);
CREATE TABLE app.users (
  user_id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  org_id bigint NOT NULL REFERENCES app.orgs(org_id),
  email text NOT NULL,
  created_at timestamptz NOT NULL DEFAULT now(),
  CONSTRAINT email_uniq_per_org UNIQUE (org_id, email)
);
CREATE INDEX ON app.users (created_at);
INSERT INTO app.orgs(name) VALUES ('NUS'),('KTH') RETURNING org_id;

WITH ins AS (
  INSERT INTO app.users (org_id, email)
  SELECT org_id, concat('user', g, '@x.com')
  FROM app.orgs, generate_series(1,3) AS g
  RETURNING *
)
SELECT * FROM ins;

-- query: last 2 users per org
SELECT * FROM (
  SELECT org_id, email, created_at,
         row_number() OVER (PARTITION BY org_id ORDER BY created_at DESC) AS rn
  FROM app.users
) t WHERE rn <= 2;

COMMIT;
```

26 Where this fits in a curriculum

This mirrors typical university DB courses (e.g., Berkeley CS186, CMU 15-415, NUS CS2102): master relational algebra in practice (`SELECT/FROM/WHERE/GROUP BY/HAVING`), then PG-specific power features (arrays/JSON/windows/CTEs/indexes), and finally transactional semantics (isolation/locking/RLS).

27 Further Study Pointers (conceptual)

Normalization vs. JSONB tradeoffs; logical vs. physical design; cost-based planning; index selectivity; MVCC internals; vacuum and freeze; partition pruning; `EXPLAIN` reading; concurrency anomalies & serializable retries; FDW performance; `pg_trgm`/`btree_gin`/`btree_gist` nuances.