Engineer

ETL

🔄 moving data between systems, reshaping it on the way

What it is

ETL, short for extract, transform, load, is the work of copying data out of the systems that create it, putting it into the shape and meaning another system expects, and writing it there, most often into a data warehouse. A close variant, ELT, keeps the three steps but loads the raw data first and transforms it inside the target, and many teams write ETL/ELT to cover both.

Each letter is a separate job. Extract reads from a source: a whole table, an export file, an API, or only the rows that changed since the last run, found through an `updated_at` timestamp or through change data capture (CDC), which reads the database's own record of every change. Transform is where the data is cleaned and reconciled: text turned into dates and numbers, units and time zones made consistent, duplicates removed, codes looked up against reference tables, and the result reshaped into the model the target uses, such as fact and dimension tables. Load writes the result, either as a bulk insert into an empty table or as a merge that updates the rows already there and adds the new ones.

The order of the last two steps decides where the work runs. Classic ETL transforms on a separate server before anything reaches the target, which made sense when warehouse storage and compute were scarce and costly. ELT lands the raw data in staging tables inside the target and runs the transformation there in SQL. The raw copy stays available, so when a rule changes the transformation can be run again over history without extracting it a second time. The price is that everything the source sent, including columns nobody asked for, now lives in the warehouse and has to be governed there.

PostgreSQL covers each step with tools of its own. `COPY` moves rows between a table and a file in text, CSV or binary format and is the fastest way in. Reading a file on the server needs superuser rights or the `pg_read_server_files` role, while

the psql command `\copy` reads the file on the client instead. `INSERT ... ON CONFLICT` and `MERGE` fold new rows into existing ones. Logical decoding turns the write-ahead log into a stream of row changes for CDC, and a PL/pgSQL procedure can keep the transformation next to the data it changes.

Use the concept in a decision

Put “What is ETL?” to work in a real decision. We can help relate the concept to the system you are building or buying. Numbers arriving where they are needed, on a schedule, without anyone copying a spreadsheet at the end of the month.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. One infrastructure plan for 14 departments — 14 research departments on one infrastructure plan

One infrastructure plan for 14 departments Get in touch →

Why it matters

Most questions worth asking a business need data from more than one system. Orders sit in one database, payments in a provider’s export, customer records in a CRM. Until something moves them into one place under one set of rules, every answer is assembled by hand, and the hand changes with the person doing it. ETL is that something, and a warehouse is only as correct as the loads that fill it.

The transform step is also where a business writes down what its data means. Which rows count as a sale, which day a midnight payment belongs to, which record wins when two customers match: decided once in the load, the answer applies to every report downstream. Left to each report, the answers drift apart and two dashboards show two totals.

How a load extracts sets how it grows. A full reload copies everything every time. It is simple and corrects itself, since yesterday’s mistake is overwritten tonight, but its cost rises with the total size of the data. An incremental load moves only what changed, so its cost rises with the rate of change instead. It has to remember where the last run stopped, notice rows deleted at the source, and accept rows that arrive late with old timestamps. Most systems start with full reloads and switch table by table once the night stops being long enough.

The ongoing cost is that loads are code that runs unattended, usually at night, against sources that change without notice. They need logging that says what ran and how many rows moved, alerts when a run stops, and one property above

all: a run repeated with the same input should leave the target exactly as one run would. A load with that property can be retried after any failure without anyone checking first.

What it looks like in practice

Fragmented pipelines were delaying business-critical reporting across several PostgreSQL development and production environments, and 600 ETL/ELT pipelines were written in PL/pgSQL to replace them: ETL at the scale of hundreds of routine loads. Primary and foreign key constraints guarded referential integrity during transformations, unique and composite indexes supported the joins, and exception handling with transactional control rolled a run back when it went wrong. Incremental loads used CDC and timestamp-based deltas, and execution logging and auditing supported tracking and debugging. The delivered work comprises transformation logic, constraints, incremental loading and execution logging; the record establishes no comparative benchmark or error-rate calculation.

A migration is ETL that runs once. Moving a map application from Microsoft SQL Server to PostgreSQL with PostGIS meant carrying over 50 million spatial records through ETL pipelines whose transformation steps converted every coordinate to the EPSG:4326 reference system. Once the records had landed, with GiST and SP-GiST indexes and a tuned configuration on the new side, spatial queries that had averaged 2.5 seconds came back in under 250 milliseconds, ten times faster, and backend CPU load dropped by 65%.

Across a network boundary, six ETL/ELT pipelines show every letter of the name at work. Extraction meant scheduled downloads of compressed archives from a server reachable only through an IPsec VPN, with Bash and Python scripts sorting the Parquet, CSV and .bak files inside by type and column layout, and the .bak backups restored into a local MSSQL Server instance. The data was loaded into staging schemas in MSSQL and PostgreSQL with bcp, psql or SSIS depending on the format, cleaned, normalised and enriched by PL/pgSQL and T-SQL procedures, then compressed with gzip and carried through a Google Cloud Storage bucket into Google BigQuery. Work that had taken at least three hours by hand ran in under 20 minutes end to end, and the sync moved from weekly to daily.

Where it goes wrong

A retried run loads the same rows twice. A job inserts with plain `INSERT`, stops halfway on a network error, and the scheduler runs it again from the start. Rows that made it in the first time are now in twice, and every total built on them is too high. Give the target a key that identifies a row in business terms and fold the batch in with `INSERT ... ON CONFLICT DO UPDATE`, which PostgreSQL documents as an atomic insert-or-update even under high concurrency, or with `MERGE`. One catch: a single `ON CONFLICT DO UPDATE` statement refuses to change the same row twice and raises a cardinality violation, so a batch with repeated keys has to be reduced to one row per key first.

```
INSERT INTO meter_reading (meter_id, read_at, kwh)
SELECT DISTINCT ON (meter_id, read_at) meter_id, read_at, kwh
FROM staging_meter_reading
ORDER BY meter_id, read_at, received_at DESC
ON CONFLICT (meter_id, read_at)
DO UPDATE SET kwh = EXCLUDED.kwh;
```

A bulk load stops near the end and leaves the disk fuller than before. `COPY FROM` writes rows into the table as it goes. When it stops on a bad row, those rows stay behind in a deleted state, invisible to queries but still on disk until `VACUUM` reclaims them, and a failure late in a large file can waste a lot of space. PostgreSQL 17 added `ON_ERROR ignore`, which skips rows in text or CSV input whose values will not convert to their column's type, and PostgreSQL 18 added `REJECT_LIMIT` to cap how many it may skip. That covers type conversion only; a row that breaks a constraint still stops the command. Loading into a staging table first, checking it there, and moving the good rows into the real table in one transaction keeps a bad file away from the data people read.

Change data capture fills the disk. Logical decoding reads changes through a replication slot, and a slot keeps every piece of write-ahead log its consumer still needs, whether or not a consumer is connected. When the reader stops, the log piles up until the disk is full, and in extreme cases the server shuts down to prevent transaction ID wraparound. The PostgreSQL documentation is plain about it: a slot that is no longer required should be dropped. After a crash, a slot can also send recent changes a second time, so the consumer has to tolerate duplicates, which is the same re-run property again.

The morning's first report is slow after a large load. The planner chooses how to run a query from statistics about each table, and a bulk load changes the data under them. Autovacuum may refresh those statistics on its own, but not always before the first report runs. Running `ANALYZE` as the last step of the load fixes it. For very large loads into an existing table, the PostgreSQL guide to populating a database adds that dropping indexes and foreign keys and creating them again afterwards can pay off, because building them in bulk beats updating them row by row. That trades away error checking while they are gone, so a unique index deserves a second thought before it goes.

Further reading

- COPY — the bulk loading command, its formats and options, and what a failed load leaves behind.
- Populating a Database — the PostgreSQL advice for large loads, from one transaction and `COPY` to rebuilding indexes and running `ANALYZE` afterwards.
- MERGE — insert, update and delete in one statement, driven by how each source row matches the target.
- INSERT — the `ON CONFLICT` clause, its atomic upsert and its rule against touching one row twice.
- Logical Decoding Concepts — how changes are read from the write-ahead log and why an unused replication slot has to go.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is ETL?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Printed from https://engineer.company/glossary/etl/

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.