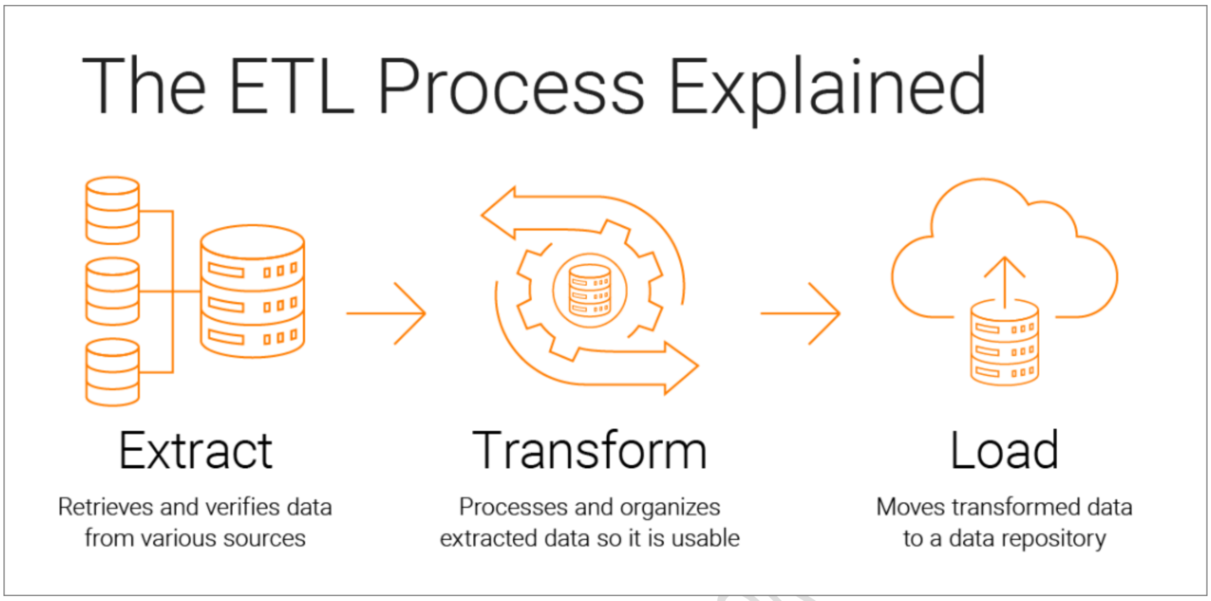
Mukesh Chaurasia – SQL DBA

COURSE: POSTGRESQL DATABASE ADMINISTRATOR

DAY – 10

TOPIC: Postgres Integration with Other Tools (BI, ETL, Cloud)


The ETL Process Explained
Extract
Retrieves and verifies data from various sources
Transform
Processes and organizes extracted data so it is usable
Load
Moves transformed data to a data repository

Mind Map

Power Bi & Postgres SQL Analytical Report

- **Purpose**
  - Connect Power BI to Postgres SQL
  - Pull Data
  - Design Simple Analytical Report
  - Not about Postgre SQL DB
- **Power BI Desktop Download & Install**
  - Free Software
  - Windows Platform Only
  - Download Steps
    - Search google ; Powerbi desktop
    - Go to Microsoft download site
    - Select Language
    - Check details (version, space, OS)
    - Click Download
    - Select 64 bit Installer
  - Installation Steps
    - Run installer
    - Language

        - Accept License Agreement
        - Default path : C drive
        - Create desktop shortcut
        - Finish (Launch powerbi automatically)

- **Connect to Postgres SQL Database**
    - Required Information
        - DNC/IP Address
        - Port (5432 default)
        - Database Name
        - Credentials (username, pass)
    - Connection Steps
        - Click 'Get Data' menu
        - Select More option
        - Select Database Category
        - Select PostgreSQL Database
        - Click Connect
        - Enter Server\Port\Database Name
        - Click OK
        - Enter username\Password
        - Click Connect
    - Load Data
        - Connection Established (schema/Table displayed)
        - Select employees table (sample data)
        - Click Load
        - Data Loaded
        - Data View (optional Check)
        - Model View (for relation, not converted in depth)
- **Design Analytical Report**
    - Go to Report view
    - Add table visual
    - Add pie chart visual
    - Add slicer visual
    - Add ribbon chart visual
    - Solve duplicate last name (calculated Column)
    - Enhance ribbon chart (conditional formatting)
- **Final Report Touches**
    - Change Canvas background
    - Rename ribbon chart title
    - Add report header
- **Save & Publish**

  - Save report: employ data from postgres
  - Publish option
  - Desktop version: Free for learning

PostgreSQL is an adaptable open-source relational database that integrates seamlessly with a wide range of tools for Business Intelligence (BI), Extract, Transform, Load (ETL), and cloud platforms. Its robustness and flexibility make it a central component in modern data stacks.

Business Intelligence (BI) Tools

PostgreSQL serves as a reliable data source for numerous BI tools, enabling powerful data visualization, reporting, and analysis. Direct connectors or standard **ODBC/JDBC drivers** facilitate these connections, allowing users to leverage PostgreSQL's structured data.

- **Tableau**: A popular tool for creating interactive dashboards and performing in-depth data exploration. It connects directly to PostgreSQL, enabling users to build visualizations from their data.
- **Power BI**: Microsoft's business analytics service that provides interactive visualizations and business intelligence capabilities. It connects to PostgreSQL to create comprehensive reports and analytical models.
- **Looker**: A BI platform with a strong focus on data modeling and self-service analytics. It uses LookML (a data modeling language) to define data relationships and business logic on top of PostgreSQL.
- **Qlik Sense/View**: Known for their associative data modeling, these tools allow users to explore data freely without predefined hierarchies, connecting to PostgreSQL for data extraction.

ETL Tools

ETL tools are essential for building data pipelines to move and transform data into and out of PostgreSQL. These tools automate workflows, ensuring data consistency and preparing data for analysis.

- **Cloud-based ETL**: These are managed services that simplify data replication and transformation.
  - **Fivetran, Stitch Data, Hevo Data**: These tools specialize in automated data replication from various sources into PostgreSQL, handling schema changes and incremental updates.

- **Open-source ETL**: These tools offer greater flexibility and control over the data pipeline.
    - **Apache Airflow**: A platform to programmatically author, schedule, and monitor workflows. It's often used to manage complex ETL pipelines with PostgreSQL.
    - **Talend Open Studio**: Provides a graphical interface for building data integration jobs, connecting to PostgreSQL as both a source and a destination.
    - **Apache NiFi**: A system for moving data between various sources and destinations, supporting a wide range of processors for data routing and transformation with PostgreSQL.
- **Data Build Tool (dbt)**: This tool focuses specifically on the **T** in ETL, allowing data analysts and engineers to transform data within the data warehouse (PostgreSQL) using SQL. It promotes software engineering best practices like version control and testing for data transformations.

Cloud Platforms

PostgreSQL is widely supported across all major cloud providers, which offer managed database services that simplify deployment, scaling, and maintenance. These services automate backups, security patches, and high availability, making PostgreSQL easier to manage.

- **Amazon Web Services (AWS)**: Offers **Amazon RDS for PostgreSQL**, a managed service for relational databases, and **Amazon Aurora PostgreSQL**, a cloud-native relational database that is compatible with PostgreSQL and offers higher performance and availability.
- **Microsoft Azure**: Provides **Azure Database for PostgreSQL**, a fully managed database service with options for single or flexible servers.
- **Google Cloud Platform (GCP)**: Supports PostgreSQL through **Cloud SQL for PostgreSQL**, a fully managed database service that automates database management tasks.
- **Heroku**: A platform-as-a-service (PaaS) that includes **Heroku Postgres** as a managed database offering, widely used in application development.

Other Integrations

PostgreSQL's extensibility further enhances its integration capabilities with other systems.

- **Foreign Data Wrappers (FDW)**: This feature allows PostgreSQL to connect to and query external data sources as if they were local tables. For example, a postgresql_fdw can connect to another PostgreSQL database, while other FDWs exist for **MySQL**, **Oracle**, and even **CSV files**. This enables a unified data access layer.
- **Extensions**: PostgreSQL's rich extension ecosystem adds specialized functionality. For example, pg_partman helps with automatic table partitioning for large datasets, and pg_cron allows for scheduling database tasks directly within PostgreSQL. This functionality minimizes the need for external tools for certain automation tasks.

ETL, which stands for **Extract, Transform, and Load**, is a fundamental three-step process in data management, primarily used to move data from one or more source systems into a target database or data warehouse for analysis and reporting. This process is a cornerstone of business intelligence and data warehousing.

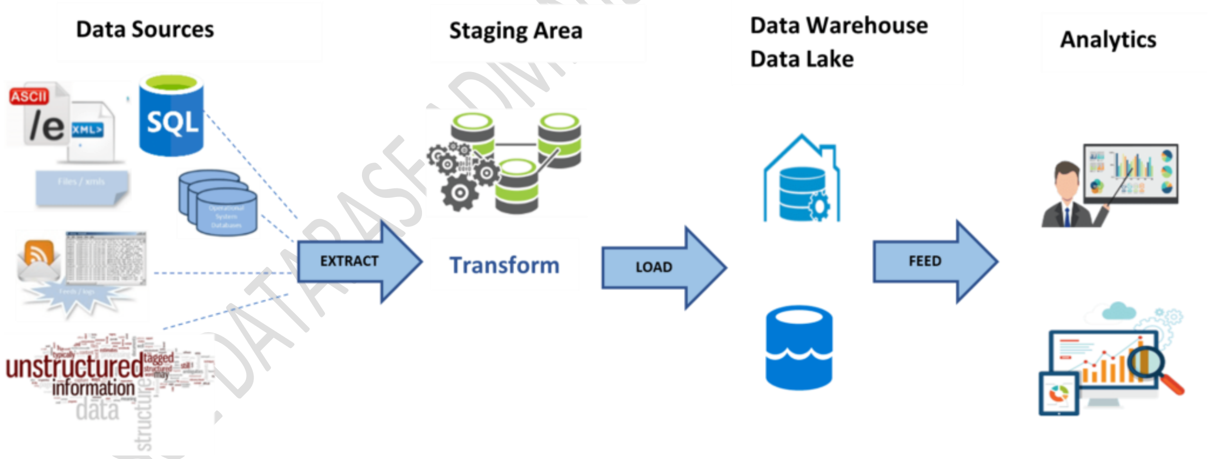
Here is a breakdown of the three phases:

1. **Extract**: This is the first step where raw data is gathered from various source systems. These sources can be diverse, including relational databases (like PostgreSQL, Oracle, MySQL), flat files (such as CSV or JSON), cloud applications (like Salesforce or HubSpot), and APIs. The extraction process must be efficient to handle different data formats and volumes without disrupting the source systems.
2. **Transform**: Once extracted, the data is not yet ready for the target system. This phase involves cleansing and structuring the data to meet the business requirements of the destination database. Common transformation tasks include:
   - **Data Cleansing**: Removing duplicate records, correcting errors, and handling missing values.
   - **Standardization**: Converting data into a consistent format (e.g., standardizing date formats or country codes).
   - **Aggregation**: Summarizing data from a detailed level to a higher, more manageable level.
   - **Derivation**: Creating new data fields or values based on existing ones.

- **Integration**: Merging data from multiple sources to create a unified view.

3. **Load**: In the final step, the transformed data is moved into the target database or data warehouse. This can be done in various ways:
    - **Full Load**: The entire dataset is loaded into the target, often used for initial data migration.
    - **Incremental Load**: Only new or changed data is loaded, which is more efficient for ongoing updates and is the most common approach for live data pipelines.

While many commercial and open-source ETL tools exist to automate this process, such as **Talend**, **Informatica**, and **Apache Airflow**, businesses often face unique data migration challenges that these tools cannot fully address out-of-the-box. In such cases, database administrators (DBAs) and developers often resort to building custom ETL scripts using programming languages like Python or SQL to handle complex logic, ensuring that the data is migrated and transformed precisely according to their specific business needs.



The Strategic Imperative of ETL

In the modern data landscape, organizations are flooded with data from a multitude of sources, including transactional databases, flat files, spreadsheets, and cloud applications. The challenge isn't just collecting this data but making it usable for business intelligence and analytics. This is where ETL (Extract, Transform, Load) becomes indispensable.

ETL is the process of taking data from its disparate origins, cleaning and standardizing it, and then moving it to a central location, typically a data warehouse. Without an efficient

ETL process, organizations would be forced to manually manage data inconsistencies, leading to significant time delays, high costs, and inaccurate insights. Using dedicated ETL tools, whether commercial or open-source, automates this complex process, ensuring data is always clean, consistent, and ready for analysis.

The Two Primary Drivers for Data Migration with ETL

While ETL is a continuous process for ongoing data integration, it is particularly critical for large-scale data migration projects. Two common and compelling reasons drive these migrations:

1. **Consolidating Data for a Data Warehouse**: Businesses often need to bring together data from various operational systems (e.g., sales, inventory, CRM) into a single, unified database for holistic analysis. This is the foundation of a data warehousing environment. ETL tools are essential here because they can handle the complexity of extracting data from different formats and schemas, transforming it to a common model, and loading it into the data warehouse. This approach allows real-time applications to continue using their existing databases while the data warehouse serves as the dedicated environment for reporting and analytics.
2. **Migrating from Commercial to Open-Source Databases**: Many organizations are looking to reduce their total cost of ownership (TCO) by moving away from expensive proprietary databases like Oracle or SQL Server to cost-effective, open-source alternatives. **PostgreSQL** is a prime candidate for this transition. An ETL process is a non-negotiable part of this migration, as it handles the critical task of moving all data—including schemas, tables, and records—from the old commercial system to the new open-source one.

The Case for Migrating to PostgreSQL

PostgreSQL has emerged as a top choice for both new projects and database migrations for several compelling reasons:

- **Cost Efficiency**: As a robust open-source database, PostgreSQL is free to use. This eliminates the significant licensing fees associated with commercial databases, leading to a substantial reduction in IT costs. This cost-effectiveness allows organizations to allocate their budget to other strategic initiatives.
- **Feature-Rich and Enterprise-Class**: PostgreSQL is not just a low-cost alternative; it is a powerful, enterprise-grade database. It is highly compliant with SQL standards and offers advanced features like support for complex data types (JSONB), advanced indexing, and Foreign Data Wrappers (FDW), which allow it to interact with other data sources.

- **Reliability and Performance**: Known for its data integrity (ACID compliance), PostgreSQL is trusted for mission-critical applications. Its robust architecture and features ensure high performance and reliability, even under heavy and complex workloads.
- **Community and Support**: The active and vibrant open-source community provides a wealth of resources, regular updates, security patches, and a collaborative environment for innovation. This continuous development ensures that PostgreSQL remains a modern and competitive database solution.

The combination of the need for data consolidation and the financial and technical benefits of migrating to PostgreSQL makes the use of an efficient ETL tool a necessity. Luckily, the vast ecosystem of ETL tools, both commercial and open-source, fully supports PostgreSQL, making the migration process more streamlined and manageable.

Both Ora2pg and Talend are powerful tools for data migration, particularly for moving data to PostgreSQL, but they cater to different needs and user profiles. Ora2pg is a specialized, command-line utility designed specifically for Oracle-to-PostgreSQL migrations, while Talend is a general-purpose, GUI-based ETL tool for a wide range of data integration tasks.

Ora2pg

Ora2pg is a highly specialized, Perl-based command-line tool that is the go-to option for migrating Oracle databases to PostgreSQL. It is renowned for its accuracy and efficiency in handling the nuances of both database systems.

Key Features and Process:

1. **Specialization**: Ora2pg is purpose-built for Oracle-to-PostgreSQL migrations. It understands specific Oracle data types, schemas, and objects, enabling it to handle the migration with high fidelity.
2. **Schema and Data Migration**: The tool can migrate the entire database schema, including tables, indexes, views, sequences, and triggers. It then efficiently migrates the data, ensuring data integrity.

3. **Command-Line Interface**: Ora2pg operates through a textual interface, making it ideal for automation via shell scripts. This allows DBAs and developers to easily schedule and run migration jobs in the background.

Step-by-Step Migration Process (Conceptual Diagram)

1. **Setup**: Install Perl and the required modules, along with the Oracle and PostgreSQL client libraries on the migration server. This can be complex depending on the operating system.
2. **Configuration**: Create a configuration file (ora2pg.conf) to specify the Oracle source database credentials, the PostgreSQL target database details, and various migration options (e.g., what to migrate, data types to handle).
3. **Schema Export**: Run Ora2pg to generate the PostgreSQL schema. The tool analyzes the Oracle schema and creates a compatible SQL script that can be executed on the target PostgreSQL database.
4. **Data Migration**: After the schema is created, run Ora2pg again to migrate the data. The tool reads data from Oracle and writes it to the corresponding PostgreSQL tables, handling data type conversions as needed.

To elaborate on the Ora2pg migration process, let's break down each step with more detail on the procedure, actions, and example scripts.

1. Setup: Installation and Prerequisites 💻

This is the most crucial, and often the most complex, initial step. You can't run Ora2pg without getting all its dependencies in place.

- **Action**: Install Ora2pg, Perl, and the necessary database client libraries on your migration server.
- **Procedure**:
    - **Perl**: Ensure Perl is installed. Most Linux distributions come with it pre-installed.
    - **Ora2pg**: Download the latest version of Ora2pg and install it. A common method is using CPAN (Comprehensive Perl Archive Network) with the command cpan Ora2pg or installing via your OS package manager like sudo apt-get install ora2pg on Debian/Ubuntu.
    - **Oracle Client**: Install the Oracle Instant Client and the Perl DBD::Oracle module. The Oracle client is necessary for Ora2pg to connect to the

source Oracle database. The DBD::Oracle module is the Perl driver for that connection.

  - **PostgreSQL Client**: Install the PostgreSQL client libraries and the Perl DBD::Pg module. This allows Ora2pg to connect to the target PostgreSQL database. The DBD::Pg module is the corresponding Perl driver.

- **Script Example**: (For a Debian/Ubuntu system)

```
# Install Ora2pg and PostgreSQL client
sudo apt-get update
sudo apt-get install ora2pg libdbd-pg-perl


# Install Oracle Instant Client (manual download from Oracle's website)
# This involves downloading and unzipping the files, then setting environment variables.
# For example:
export LD_LIBRARY_PATH=/path/to/instantclient_19_3:$LD_LIBRARY_PATH
export ORACLE_HOME=/path/to/instantclient_19_3


# Install the Perl Oracle driver via CPAN
sudo cpan install DBD::Oracle
```

2. Configuration: The ora2pg.conf File 🛠

This file acts as a blueprint for the entire migration process, telling Ora2pg exactly what to do.

- **Action**: Create and edit a configuration file, typically named ora2pg.conf.
- **Procedure**:
  - **Source and Target Connections**: Define the Oracle source database (ORACLE_DSN, ORACLE_USER, ORACLE_PWD) and the PostgreSQL target database (PG_DSN, PG_USER, PG_PWD).
  - **Migration Scope**: Specify which schemas, tables, or objects to migrate using parameters like SCHEMA, TABLES, EXCLUDE_SCHEMA, EXCLUDE_TABLES.

- **Behavioral Parameters**: Fine-tune the migration by setting options for data types, parallelism, and other specific behaviors. For example, LONGREADLEN to handle large objects (LOBs) or PARALLEL_TABLES for concurrent data migration.

- **Script Example**: (Excerpt from ora2pg.conf)

```
# Oracle Source Database
ORACLE_DSN dbi:Oracle:host=192.168.1.10;sid=ORCL;port=1521
ORACLE_USER ora_user
ORACLE_PWD ora_password

# PostgreSQL Target Database
PG_DSN dbi:Pg:dbname=my_pg_db;host=192.168.1.20;port=5432
PG_USER pg_user
PG_PWD pg_password

# Migration Options
TYPE TABLE,TRIGGER,VIEW,SEQUENCE,FUNCTION
SCHEMA "HR"
```

---

3. Schema Export: Generating the SQL Script 📜

This step is all about translation—Ora2pg reads the Oracle schema and writes it out as a compatible PostgreSQL schema definition.

- **Action**: Run the ora2pg command with the --type SCHEMA parameter to generate a DDL (Data Definition Language) script.
- **Procedure**:
  - Ora2pg connects to the Oracle database and queries its metadata tables to understand the schema structure.
  - It then translates Oracle-specific syntax and data types (e.g., NUMBER to NUMERIC, VARCHAR2 to VARCHAR) and generates a .sql file.

   - This script includes CREATE TABLE, CREATE INDEX, CREATE SEQUENCE, and other commands needed to replicate the schema in PostgreSQL.

- **Script Example**:

```
# Command to generate schema migration script
ora2pg -c /path/to/ora2pg.conf -t SCHEMA --output schema.sql
# After generation, execute the script on the target PostgreSQL database
psql -h 192.168.1.20 -U pg_user -d my_pg_db -f schema.sql
```

---

4. Data Migration: The Final Transfer ➡

Once the schema is in place, the tool moves the actual data. This is often the longest step for large databases.

- **Action**: Run the ora2pg command with the --type COPY parameter to migrate the data.
- **Procedure**:
   - Ora2pg reads data from the specified Oracle tables.
   - For performance, it uses PostgreSQL's COPY command, which is a highly efficient bulk-loading mechanism, to write the data into the corresponding tables on the target database.
   - This process handles data type conversions and can be parallelized for faster performance on large tables.
- **Script Example**:

```
# Command to run data migration
ora2pg -c /path/to/ora2pg.conf -t COPY --output data.sql
# You can also use -t INSERT but COPY is much faster for bulk data
ora2pg -c /path/to/ora2pg.conf -t INSERT --output data.sql
# After generation, execute the script on the target PostgreSQL database
psql -h 192.168.1.20 -U pg_user -d my_pg_db -f data.sql
```

This structured approach ensures that the migration is systematic, repeatable, and verifiable, minimizing the risk of data loss or corruption.

Talend

Talend is a comprehensive, Java-based ETL (Extract, Transform, Load) tool with a graphical user interface (GUI) that supports a wide array of data sources and destinations. It is a general-purpose solution for data integration, not limited to a specific database migration.

Key Features and Process:

1. **General-Purpose ETL**: Talend's strength lies in its ability to connect to virtually any data source (databases, files, cloud services) and integrate the data into any target system.
2. **Graphical Interface**: It provides a visual development environment where users can drag-and-drop components to design complex data pipelines. This makes it accessible to developers and DBAs who are less comfortable with command-line scripting.
3. **Extensibility**: As a Java-based tool, Talend allows developers to incorporate custom Java code into their jobs, providing immense flexibility to handle specific business logic or data transformation requirements.

Step-by-Step Migration Process (Conceptual Diagram)

1. **Setup**: Install the Talend Open Studio, which is a straightforward process on any platform that supports Java.
2. **Job Design**: In the Talend GUI, create a new "job." This involves:
    - **Defining Source**: Drag a component for the source database (e.g., an Oracle connection) and configure its credentials.
    - **Defining Target**: Drag a component for the target database (PostgreSQL) and configure its connection details.
    - **Adding Transformation**: Use various built-in components to perform data cleansing, filtering, or transformation between the source and target.
3. **Execution**: Run the job from the GUI. Talend executes the underlying Java code to extract data, apply transformations, and load it into the PostgreSQL database.

4. **Automation**: While direct scheduling can be a challenge with the open-source version, jobs can be exported as executable files and scheduled using external tools like cron jobs (on Linux) or Task Scheduler (on Windows).

Comparison and Use Cases

Feature	**Ora2pg**	**Talend**
Purpose	Specialized Oracle-to-PostgreSQL migration	General-purpose ETL and data integration
Interface	Command-line (textual)	Graphical User Interface (GUI)
Complexity	Complex installation, simple execution	Simple installation, complex job design for non-trivial tasks
Performance	Optimized for large, bulk data migration	Efficient for medium-sized tables; may struggle with very large tables and LOBs
Extensibility	Limited; relies on Perl scripting	High; allows custom Java code integration
Best for...	DBAs and developers needing a reliable, fast, and automated migration from Oracle to PostgreSQL.	Developers and data professionals handling diverse data sources and complex transformations.

Below is a practical, DBA-friendly guide you can use to choose—and successfully run—an ETL/ELT migration to **PostgreSQL**. I've kept it tool-agnostic where possible, with concrete procedures for both one-time bulk loads and near-zero-downtime CDC. I also included a downloadable **evaluation matrix template** so you can score tools against your requirements.

1) What to evaluate (and what "good" looks like)

1. Understand your data

- **Shape & complexity**: number of tables, relationships, cycles, implicit dependencies, nested JSON/XML, LOBs.

- **Data types**: SQL Server → PostgreSQL mapping (see §8). Watch datetimeoffset, money, uniqueidentifier, varbinary, geometry/geography, nvarchar(max).
- **Cardinality & size**: per-table row counts, largest LOBs, daily change rate, skew.
- **Data quality**: nullability violations, invalid dates, orphan FKs, code-page/encoding issues, duplicate keys.
- **Source location & access**: on-prem, cloud, firewalls/VPN, TLS, throughput limits, throttling.

2. Migration expectations & workload profile

- **One-time vs. continuous**: cutover window vs. near-zero downtime (CDC).
- **Batch vs. streaming**: large nightly drops vs. real-time events.
- **Latency SLOs**: e.g., <5 min for changes.
- **Transform style**: pushdown ELT in Postgres vs. ETL in a middle tier.

3. Platform & operability

- **OS support**: Linux/Windows/macOS/Kubernetes.
- **Deployment**: on-prem VM, containers, managed (PaaS/SaaS).
- **Security**: TLS, encryption at rest, key management, masking/row-level filtering.
- **Observability**: logs, metrics, lineage, alerting, retries, back-pressure.
- **Extensibility**: CLI/SDK, scriptability, templates, GitOps/CI-CD.
- **Cost**: license, infra, team skills.

4. PostgreSQL-specific capability

- Uses **COPY** (or equivalent bulk path) for speed.
- Handles **types** correctly (UUID/JSONB/bytea/arrays).
- Supports **upsert** (INSERT ... ON CONFLICT) for CDC merges.
- Can control **parallelism**, **order of operations**, and **idempotency**.

2) Tool landscape (quick map)

- **Postgres-focused bulk**: pgloader (fast, scriptable), psql + COPY, pg_bulkload.
- **Enterprise ETL**: SSIS (use ODBC or stage + COPY), Talend, Pentaho/Kettle, Informatica.

- **Cloud integration**: Azure Data Factory (ADF), AWS DMS (great for heterogeneous CDC), GCP Dataflow.
- **Streaming/CDC**: Debezium (SQL Server CDC) → Kafka → Postgres sink; Striim; Qlik Replicate.
- **Orchestration**: Airflow/Prefect/Azure DevOps/SQL Agent/cron.
- **ELT after load**: dbt for Postgres-native transforms/tests/docs.
- **GUI one-click**: fine for small/medium data; less control at scale.

Tip: for very large datasets, tools that **stage to files** and then invoke Postgres **COPY** consistently outperform row-by-row JDBC inserts.

3) Two proven reference architectures

A) One-time bulk (fast cutover)

- **Extract** from SQL Server (bcp/sqlcmd/ADF/SSIS) → **stage files** (CSV/Parquet).
- **Load** with psql \copy or server-side COPY into **staging tables**.
- **Transform/merge** inside Postgres (ELT) and build indexes after bulk load.
- **Validate** (row counts/checksums) → cutover.

Best when: a maintenance window is available and you want maximum throughput.

B) Near-zero downtime (CDC)

- **Enable SQL Server CDC** → stream with **Debezium** or **AWS DMS** → land to Kafka or directly to Postgres.
- **Initial snapshot** + ongoing change stream.
- **Apply** changes with **upserts**; ensure **idempotency** and ordering.
- **Validate** and switch traffic when lag ≈ 0.

Best when: the app can't afford downtime; you need continuous sync until cutover.

4) Step-by-step: Procedure A (Bulk migration)

Phase 0 — Discovery & profiling

1. Inventory sources (schemas, tables, PK/FK, indexes).
2. Capture row counts, sizes, and daily change rate.
3. Profile for data quality: nulls, invalid dates, bad encodings.

Phase 1 — Target design (PostgreSQL)

1. Map data types (see §8).
2. Decide on **partitions** and **table/PK** design.
3. Plan **indexes** and **constraints** (create after load).
4. Choose **staging** vs. direct load strategy.

Phase 2 — Choose bulk path

- Preferred order: COPY from local files on server > \copy via psql > batched inserts via JDBC/ODBC.

Phase 3 — Create schema

- Create **staging** tables without FKs/triggers, minimal indexes.
- Create **final** tables (without secondary indexes; define PKs).

Phase 4 — Extract

- From SQL Server:
  - **bcp** to CSV with proper encoding (-w Unicode), delimiter, null representation.
  - Or **ADF/SSIS** to land files (Blob/NFS). Avoid transformations here.

Phase 5 — Load (very fast)

- On Postgres, session-level tuning (only for the session doing the load):
  - SET synchronous_commit = OFF;
  - SET maintenance_work_mem = '2GB'; (fit to server)
  - SET wal_compression = ON;
- Load to staging:
- psql -v ON_ERROR_STOP=1 -d targetdb -c "\copy staging.my_table from '/data/my_table.csv' csv header quote '\"' escape '\"' delimiter ',' null ''"
- For very large tables, **split files** and run loads **in parallel** by key ranges (but avoid disk thrash).

Phase 6 — Post-load optimize

- Create required **indexes** (consider CREATE INDEX CONCURRENTLY if the table must stay writable).
- Run ANALYZE (or VACUUM ANALYZE) on loaded tables.

- Re-enable constraints/triggers if disabled.

Phase 7 — Transform & merge

- Use **ELT**: INSERT ... SELECT, MERGE (v15+), or INSERT ... ON CONFLICT.
- For SCD2, use staged diff tables.

Phase 8 — Validation

- **Counts** per table: source vs. target.
- **Checksums**: MD5 across canonical column order.
- Spot-check **business keys** and totals (SUM(amount)).

Phase 9 — Cutover

- Freeze source writes (short outage), capture delta, load delta, validate, switch connection strings.
- Keep rollback plan: retain source in read-only until sign-off.

5) Step-by-step: Procedure B (CDC migration)

Phase 0 — Prepare source

1. Ensure SQL Server has primary keys on replicated tables.
2. Enable CDC at DB and table level:
3. EXEC sys.sp_cdc_enable_db;
4. EXEC sys.sp_cdc_enable_table
5. @source_schema = N'dbo',
6. @source_name = N'Orders',
7. @role_name = NULL,
8. @supports_net_changes = 1;

Phase 1 — Deploy CDC pipeline

- **Debezium** SQL Server connector (or **AWS DMS** task) to capture inserts/updates/deletes.
- Topic or stream per table.

Phase 2 — Initial snapshot

- Take a consistent snapshot (Debezium/DMS do this) and load to Postgres.

- Apply schema evolution rules (map types, ON CONFLICT targets).

Phase 3 — Apply changes

- PostgreSQL sink does **upsert**:
- INSERT INTO public.orders(id, col1, col2, updated_at)
- VALUES ($1,$2,$3,$4)
- ON CONFLICT (id) DO UPDATE
- SET col1 = EXCLUDED.col1, col2 = EXCLUDED.col2, updated_at = EXCLUDED.updated_at;

Phase 4 — Monitor & tune

- Track **replication lag**, error rates, dead letters, and apply retries/back-pressure.

Phase 5 — Cutover

- When lag ≈ 0, pause source writes, drain last events, validate, flip traffic, then retire pipeline.

6) Performance tuning checklist (PostgreSQL)

- Prefer **COPY** over single-row inserts.
- **Load without secondary indexes/FKs**, add them after.
- Consider **UNLOGGED** staging tables (faster; not crash-safe).
- Increase maintenance_work_mem for index builds.
- Use reasonable max_wal_size and checkpoint_timeout to reduce checkpoints during bulk.
- synchronous_commit=off **only** during bulk sessions.
- Partition very large tables to enable parallel loads and simplify maintenance.
- After bulk loads, ANALYZE so the planner gets good stats.

7) Validation & reconciliation

- Row counts per table (tolerance = 0).
- Business aggregates (e.g., SUM(amount), COUNT(DISTINCT key)).
- Random sampling and key-by-key diffs for high-value tables.
- Encode a **repeatable checksum** (order columns, coalesce nulls).

- Document anomalies and remediation (e.g., values out of range).

8) Common SQL Server → PostgreSQL type mappings (cheat sheet)

SQL Server	PostgreSQL	Notes
bit	boolean	0/1 → false/true
tinyint/smallint/int	smallint/integer	Use bigint for large IDs
bigint	bigint	
decimal(p,s)/numeric	numeric(p,s)	Match precision/scale
float	double precision	
real	real	
money/smallmoney	numeric(19,4)	Avoid money type differences
datetime/smalldatetime	timestamp without time zone	Normalize time zone handling
datetimeoffset	timestamp with time zone	Beware semantics; test
date/time	date/time	
uniqueidentifier	uuid	Requires uuid-ossp or generate in app
varbinary/image	bytea	Use hex/escape format
varchar/nvarchar(max)	text or varchar(n)	Use UTF-8; check max sizes
xml	xml	
sql_variant	(model-specific)	Typically JSONB or separate columns
geography/geometry	PostGIS geography/geometry	Install PostGIS extension
identity	generated ... as identity	Prefer modern identity over serial

9) Patterns by tool (quick how-to)

pgloader (one-time, fast)

```
load database
  from mssql://user:pass@sqlserver-host/dbname
  into postgresql://user:pass@pg-host/targetdb
with
  workers = 8, concurrency = 4, batch rows = 50000,
  on error stop
set work_mem to '512MB', maintenance_work_mem to '2GB'
cast type uniqueidentifier to uuid drop default,
     type datetimeoffset to timestamptz,
     type varbinary to bytea using byte-vector-to-bytea
including only table names like 'dbo.Orders', 'dbo.Customers';
```

SSIS (Windows shops)

- Data Flow: OLE DB Source (SQL Server) → **Flat File Destination** (UTF-8, quoted) → run psql \copy via Execute Process Task.
- Or ODBC Destination to Postgres (slower); tune batch/commit sizes.

Azure Data Factory (Azure-first)

- Linked Services: SQL Server (self-hosted IR if on-prem), PostgreSQL.
- Copy Activity: partition SQL source (e.g., by date or ID ranges), increase parallel copies, land to files if you need COPY-level speed, then load with a custom psql activity (Custom/Execute) on a compute target.
- Orchestrate transform in Postgres with Stored Procedure activity or dbt jobs.

AWS DMS (heterogeneous CDC)

- Create replication instance, endpoints (SQL Server source, Postgres target).
- Task: full load + CDC, table mappings, LOB handling (Limited LOB with size or Full LOB).
- Validate with DMS Table Statistics; reconcile in Postgres.

Debezium + Kafka

- Enable SQL Server CDC; deploy Debezium connector JSON.

- Use Kafka Connect **PostgreSQL sink** with insert.mode=upsert and pk.mode=record_key.
- Add SMTs for type fixes; use schema registry; monitor via Prometheus/JMX.

10) Operational checklists

Security

- Enforce TLS on both ends, rotate creds, least-privilege service accounts.
- If staging files: encrypt at rest; sanitize PII when appropriate.

Reliability

- Retries with exponential backoff, dead-letter queues for bad records.
- Checkpointing & exactly-once (or at-least-once + idempotent upsert).

Observability

- Centralized logs; per-table throughput & error dashboards; alert on lag.

Governance

- Keep a **mapping spec** (source→target types, transforms), versioned in Git.
- Data validation plan and sign-off criteria before cutover.

---

etl_tool_evaluation_template.csv

Tailored Runbook — SQL Server → PostgreSQL (Azure)

Scope & Assumptions

- Source: SQL Server (on-prem or IaaS).
- Target: Azure Database for PostgreSQL (Flexible Server) or self-managed Postgres on Azure VM.
- Constraints: strong throughput, controlled cost, infra already on Azure; acceptable to use either **bulk cutover** or **near-zero downtime (CDC)**.

Phase 0 — Discovery & Profiling (2–5 days)

1. **Inventory**

- List DBs/schemas/tables, PK/FK, indexes, triggers, views, procs.
- Export row counts & size (top 20 largest tables).
- Identify daily change rate for CDC planning.

2. **Type & feature mapping**

- Highlight risky types: datetimeoffset, uniqueidentifier, varbinary(max), money, sql_variant, geography/geography.
- Note features needing redesign: sequences/identity, triggers, T-SQL logic, collation.

3. **Data quality**

- Check nullable violations, orphan FKs, invalid dates (0000-00-00, 1900-01-01 placeholders), non-UTF-8 text, duplicates on business keys.

Deliverables: Inventory spreadsheet, type mapping sheet, DQ findings.

Phase 1 — Target Design (1–2 days)

- Choose Postgres versions & extensions (e.g., uuid-ossp, pg_stat_statements, PostGIS if spatial).
- Define **staging schema** and **final schema**.
- Partition very large tables (range on date or hash on key).
- Decide index set (create **after** bulk loads).

Phase 2 — Tooling Strategy (choose track)

Track A — Bulk (fastest throughput; short maintenance window)

Preferred: bcp/ADF → files → psql \copy (COPY path).
Alternative: **pgloader** for end-to-end from SQL Server.

Track B — CDC (near-zero downtime)

- **Debezium (SQL Server)** → Kafka → Postgres sink (upsert), **or**
- **AWS DMS** source=SQL Server → target=Postgres (even if hosted in Azure), then cutover.

Why both? ADF's native CDC coverage to Postgres is limited for heterogeneous CDC; Debezium/DMS excel here.

Phase 3 — Schema Creation (staging first)

- Create **staging tables** (no FKs/triggers, minimal constraints).
- Create **final tables** with only **PK** (defer secondary indexes until after load).

Phase 4 — Extraction

Option 1: bcp (fast, scriptable)

```
REM Unicode, widechar export with headers (use a format file for precision if needed)

bcp "SELECT * FROM dbo.Orders WITH (NOLOCK)" queryout "D:\stage\Orders.csv" -S
MYSERVER -d MyDb -T -w
```

Tips:

- Use -w for Unicode; avoid data loss.
- Split by key ranges/date to parallelize.
- For **varbinary**, export hex or Base64 (ensure matching Postgres loader function).

Option 2: Azure Data Factory

- Linked Services: SQL Server (Self-hosted IR for on-prem), Azure Storage, PostgreSQL (for direct copy if used).
- Copy Activity: partitioned queries (e.g., WHERE OrderDate >= @rangeStart AND < @rangeEnd).
- Prefer **landing to files** (Blob/NFS) and then COPY on Postgres for best speed.

Option 3: pgloader

- Direct SQL Server → Postgres with explicit **CAST** rules (see §Type Mapping).

Phase 5 — Loading to PostgreSQL (high throughput)

Session-level settings for the bulk loader session only:

```
SET synchronous_commit = OFF;

SET maintenance_work_mem = '2GB';   -- adjust to ~10–20% of RAM

SET wal_compression = ON;         -- if available
```

Fast path using \copy:

```
psql -d targetdb -v ON_ERROR_STOP=1 -c "\copy staging.orders from '/data/Orders.csv' csv header quote '\"' escape '\"' delimiter ',' null ''"
```

For huge tables

- Split CSV by key ranges; run N parallel \copy processes (ensure disk IO and WAL can keep up).
- Consider **UNLOGGED** staging tables during load (faster, but not crash-safe).

Post-load:

```
ANALYZE staging.orders;
```

Phase 6 — Transform & Merge (ELT inside Postgres)

Identity/sequence alignment (Postgres 10+):

```
ALTER TABLE public.orders ALTER COLUMN id ADD GENERATED BY DEFAULT AS IDENTITY;

SELECT setval(pg_get_serial_sequence('public.orders','id'),

       (SELECT COALESCE(MAX(id),0) FROM public.orders), true);
```

Merge (PG 15+)

```
MERGE INTO public.orders t
USING staging.orders s
ON (t.id = s.id)
WHEN MATCHED THEN UPDATE SET
 col1 = s.col1, col2 = s.col2, updated_at = s.updated_at
WHEN NOT MATCHED THEN
 INSERT (id,col1,col2,updated_at) VALUES (s.id,s.col1,s.col2,s.updated_at);
```

Or upsert pattern (PG 9.5+):

```
INSERT INTO public.orders (id, col1, col2, updated_at)
SELECT id, col1, col2, updated_at
FROM staging.orders
```

ON CONFLICT (id) DO UPDATE

SET col1 = EXCLUDED.col1, col2 = EXCLUDED.col2, updated_at = EXCLUDED.updated_at;

Create secondary indexes **after** data lands:

CREATE INDEX CONCURRENTLY idx_orders_dt ON public.orders(order_date);

ANALYZE public.orders;

Phase 7 — Validation

Counts

-- SQL Server

SELECT COUNT(*) FROM dbo.Orders;

-- PostgreSQL

SELECT COUNT(*) FROM public.orders;

Checksums (canonicalized)

-- Example hash on key columns in Postgres

SELECT md5(string_agg(coalesce(col1::text,'') || '|' || coalesce(col2::text,''), '|' ORDER BY id))

FROM public.orders;

Business totals

- SUM(amount), COUNT(DISTINCT customer_id), spot sampling by keys.

Record a **Validation Report** for sign-off.

Phase 8 — Cutover

Bulk track

1. Freeze writes on source (short outage).
2. Export delta since snapshot; load delta; re-run validation.
3. Flip app connection strings.
4. Keep source read-only for rollback window.

CDC track

1. Run CDC until **replication lag ~ 0**.
2. Pause app writes briefly, drain last events, validate.
3. Switch traffic; monitor closely; retire pipeline.

CDC Track — Implementation Details

Option 1: Debezium (AKS/VM)

- Enable SQL Server CDC:

```
EXEC sys.sp_cdc_enable_db;
EXEC sys.sp_cdc_enable_table
 @source_schema=N'dbo', @source_name=N'Orders',
 @role_name=NULL, @supports_net_changes=1;
```

- Deploy Debezium SQL Server connector → Kafka topics.
- Kafka Connect **Postgres sink** with insert.mode=upsert, pk.mode=record_key, pk.fields=id.
- Use SMTs for type conversions; ensure **idempotent upserts**.

Option 2: AWS DMS (even in Azure)

- Create replication instance + endpoints (SQL Server → Postgres).
- Task: **Full load + ongoing replication**, tune LOB mode and batch commit.
- Monitor Table statistics & latency; validate in Postgres.

SQL Server → PostgreSQL Type Mapping (practical deltas)

- uniqueidentifier → uuid (generate server-side or cast; install uuid-ossp if needed).
- varbinary(max)/image → bytea (export hex/base64; use correct loader).
- datetimeoffset → timestamptz (normalize to UTC, confirm semantics).
- money → numeric(19,4).
- nvarchar(max) → text (UTF-8).
- geography/geometry → PostGIS types; validate SRIDs.

Operability & Tuning (Postgres)

- Bulk: disable secondary indexes/FKs during load; re-create after.
- Increase maintenance_work_mem for index builds.
- max_wal_size, checkpoint_timeout sized to reduce checkpoint churn during loads.
- After bulk, ANALYZE every large table.
- For huge merges, use temp tables + MERGE and work in key ranges.

What to Use (Recommendation Matrix — Azure-first)

Top picks by scenario (from the attached scoring):

- **Bulk one-off at maximum speed**: **bcp → files → psql \copy** (or **pgloader**) orchestrated by **ADF** or **SQL Agent/Azure DevOps**.
- **Near-zero downtime**: **AWS DMS** or **Debezium+Kafka** for CDC, followed by a brief controlled cutover.
- **Modeling & tests after load**: **dbt** on Postgres (ELT, data tests, docs).

Download the matrix with weighted scores:

- etl_tool_scoring_azure_sqlserver_to_postgres.csv

If you share a sample schema (top 10 largest tables with types and row counts) and the intended Postgres flavor (Flexible Server vs. VM), I'll generate:

- A **bcp/psql** export+load script (parallelized by ranges).
- ADF **pipeline JSON** skeleton with partitioned copy + custom psql activity.
- A **pgloader** cast file with exact type rules for your columns.

Mukesh Chaurasia – SQL DBA

COURSE: POSTGRESQL DATABASE ADMINISTRATOR

Tool	Performan	CDC/Low L	Azure Integ	Type Fideli	Scriptabilit	Operationa	Cost Efficie	Ease of Us	Licensing/	Weighted S
Debezium	4	5	3	4	4	5	3	2	5	4.01
dbt (ELT tra	5	1	4	5	5	4	5	3	5	3.96
Azure Data	4	3	5	4	4	4	4	4	4	3.94
bcp+psql C	5	2	4	4	5	3	5	2	5	3.87
AWS DMS (	4	5	3	4	4	4	3	3	3	3.86
Fivetran/St	4	4	4	4	3	4	3	5	3	3.8
pgloader	5	1	3	4	5	3	5	2	5	3.57
SSIS (stagi	4	3	3	3	4	4	3	4	3	3.43
Talend/Pe	4	3	3	3	4	4	3	4	3	3.43
Apache Ni	3	2	3	3	3	4	4	4	5	3.17

etl_tool_scoring_azure_sqlserver_to_postgr