02 Logical Backups and SQL Dump

01 What a SQL Dump Is and How It Works Internally in PostgreSQL

In simple words

- A SQL dump is a logical backup where PostgreSQL writes **SQL commands** that can rebuild the database later.
- **Think of it as instructions to recreate:**
  - database structure
  - data
  - permissions

When I restore a SQL dump, PostgreSQL simply **replays those instructions**.

Why SQL dumps exist

- SQL dumps solve portability and flexibility problems.
- **They are designed for:**
  - database migrations
  - PostgreSQL version upgrades
  - moving data across servers
  - partial restores (tables or schemas)

They are not the fastest, but they are the most flexible.

Tool used: `pg_dump`

- `pg_dump` is the standard tool used to create SQL dumps.
- **Important truth:**
  - `pg_dump` is just a normal PostgreSQL client.
  - It connects to the database like any other application and follows all role permissions.

What `pg_dump` actually reads

- `pg_dump` reads the database **logically**, not physically.
- **It reads:**
  - schemas
  - tables
  - indexes
  - sequences
  - views
  - functions
  - data
- It does not copy disk files. It reads data through SQL queries.

How `pg_dump` works internally (step-by-step)

1. `pg_dump` connects to the database
2. PGSQL creates a transaction snapshot
3. `pg_dump` reads metadata (tables, schemas, objects)
4. `pg_dump` reads table data row by row
5. SQL commands are written to the dump file

The snapshot guarantees consistency across all objects.

Why `pg_dump` does not block users

- `pg_dump` uses a snapshot-based read.
- This means:
  - no exclusive locks
  - no write blocking
  - normal queries continue

Users can keep inserting and updating data while the dump runs.

What consistency means in a SQL dump

- All tables in the dump represent the **same point in time**.
- Data committed after the dump starts is ignored.
- Uncommitted data is never included.
- This prevents broken foreign keys or partial data.

What a SQL dump contains

- **A SQL dump usually includes:**
  - CREATE DATABASE (optional)
  - CREATE TABLE statements
  - CREATE INDEX statements
  - INSERT data
  - GRANT and ownership statements
- It may also include comments and extensions if requested.

What a SQL dump does NOT contain

- **SQL dumps do not include:**
  - server configuration files
  - running transactions
  - OS-level settings
  - WAL history
- They only capture logical database objects.

Why SQL dumps are slow for large databases

- **SQL dumps:**
  - write data as INSERT statements
  - rebuild indexes during restore
  - execute commands one by one
- This makes them slower for very large databases compared to physical backups.

Restore behavior of SQL dumps

- **Restore means:**
  - create a clean database
  - run the SQL file using psql
  - PostgreSQL executes commands sequentially
- Indexes are rebuilt, not copied.
- Statistics must be regenerated after restore.

When I prefer SQL dumps

- I use SQL dumps when:
  - upgrading PostgreSQL versions
  - migrating between platforms
  - restoring individual tables
  - creating test or dev environments
- They give control, not speed.

Common misunderstanding

- A SQL dump is not a snapshot of disk files.
- It is a **logical reconstruction recipe**.
- That is why it works across versions and architectures.

Final mental model

- SQL dump = instructions
- `pg_dump` = reader and writer
- snapshot = consistency guarantee
- restore = replay SQL

One-line explanation

A SQL dump is a logical backup that stores SQL commands generated from a consistent snapshot to recreate a PostgreSQL database.

02 pg_dump Command – Deep Dive (PostgreSQL Logical Backup)

In simple words

- `pg_dump` is the tool PostgreSQL provides to take **logical backups** of a single database.
- It reads database objects through SQL and writes instructions that can rebuild the database later.
- It is safe, online, and transaction-consistent.

Important truth about `pg_dump`

- `pg_dump` is **not a server-side tool**.
- **It is a normal client program:**
  - it connects like any application
  - it follows role permissions
  - it can run from any machine with network access
- If permissions are wrong, `pg_dump` fails.

Basic `pg_dump` syntax

```
pg_dump dbname > backup.sql
```

- **What happens:**
  - `pg_dump` connects to `dbname`
  - takes a snapshot
  - reads schema and data
  - writes SQL into `backup.sql`
- The database stays online.

Connecting to a specific server

```
pg_dump -h server_ip -p 5432 -U postgres dbname > backup.sql
```

- **Meaning**:
  - `-h` → server hostname or IP
  - `-p` → port (default 5432)
  - `-U` → database role
- This works locally or remotely.

Authentication behavior

- **`pg_dump` uses the same authentication as any client:**
  - password
  - .pgpass
  - environment variables
  - peer or trust (OS-based)
- There is no special authentication bypass.

Dumping schema only or data only

Schema only:

```
pg_dump -s dbname > schema.sql
```

Data only:

```
pg_dump -a dbname > data.sql
```

- **Useful when:**
  - rebuilding structures first
  - loading data separately

Dumping specific objects

Only one table:

```
pg_dump -t customers dbname > customers.sql
```

- Only one schema:

```
pg_dump -n public dbname > public_schema.sql
```

These are common in partial restores and debugging.

Excluding objects

Exclude a table:

```
pg_dump --exclude-table=logs dbname > backup.sql
```

Exclude schema:

```
pg_dump --exclude-schema=test dbname > backup.sql
```

- Used to skip temporary or irrelevant data.

Dump formats (overview)

- **`pg_dump` supports multiple formats:**
  - `-Fp` → plain SQL (default)
  - `-Fc` → custom (compressed)
  - `-Fd` → directory (parallel)
  - `-Ft` → tar
- Format choice affects restore method and speed.

Why custom and directory formats matter

- **Custom and directory formats:**
  - are compressed
  - restore faster
  - allow selective restore
  - support parallel restore
- Plain SQL does not support parallelism.

Compression with `pg_dump`

```
pg_dump dbname | gzip > backup.sql.gz
```

- This reduces disk usage but adds CPU cost.
- Custom format has built-in compression.

Performance impact during dump

- **`pg_dump`:**
  - reads data sequentially
  - uses MVCC snapshots
  - does not block writers
- **But:**
  - large dumps consume I/O
  - CPU usage increases

Scheduling matters in production.

Common `pg_dump` failures

- **`pg_dump` often fails because:**
  - role lacks permission on one object
  - view references missing table
  - extension dependency issues
  - network interruptions
- Always read the error message carefully.

Best practices for `pg_dump`

- run backups as a dedicated role
- store dumps on separate storage
- monitor dump duration and size
- always test restore

`pg_dump` success is measured at restore time.

When `pg_dump` is the wrong tool

- **Avoid `pg_dump` when:**
  - database is extremely large
  - fast restore is critical
  - point-in-time recovery is required
- Physical backups are better in those cases.

Final mental model

- `pg_dump` reads logically
- snapshot guarantees consistency
- permissions decide success
- format decides restore strategy

One-line explanation

`pg_dump` is a PostgreSQL client tool that creates transaction-consistent logical backups by exporting database objects as SQL instructions.

03 Restore Using psql – Complete Flow (PostgreSQL Logical Restore)

In simple words

- Restoring a SQL dump means running SQL commands again to rebuild the database.
- PostgreSQL does not have a special “restore mode” for SQL dumps.
- Restore is simply **executing the dump file using psql**.

What restore actually does

- A SQL dump contains:
  - CREATE statements
  - INSERT statements
  - GRANT and ownership commands
- When restoring:
  - PostgreSQL executes these commands one by one
  - objects are rebuilt logically
  - indexes are recreated, not copied

Restore is a **replay process**, not a file copy.

Pre-restore checklist (very important)

- Before restoring, I always check:
  - target server is correct
  - PostgreSQL version is compatible
  - sufficient disk space exists
  - required roles already exist
- Most restore failures happen due to missing preparation.

Step 1: Create an empty database

```
createdb newdb
```

The target database must exist before restore.

Alternatively:

```
createdb -O app_user newdb

# This command creates a new database named newdb and sets app_user as the owner. In
simple terms, app_user becomes the main user who controls this database and has full
rights over it.
```

This sets correct ownership upfront.

Step 2: Restore using `psql`

```
psql -d newdb -f backup.sql
```

- What happens internally:
  - `psql` reads SQL line by line
  - PostgreSQL executes each statement
  - errors are reported immediately
- Restore speed depends on dump size and indexes.

Restore directly from compressed dump

```
gunzip -c backup.sql.gz | psql -d newdb
```

- This avoids extracting the file to disk.
- Useful when storage space is limited.

Restore from a remote server

```
psql -h server_ip -U postgres -d newdb < backup.sql
```

Restore works over network just like local execution.

Common restore errors and causes

Role does not exist

Error:

```
ERROR: role "app_user" does not exist
```

- **Fix:**
  - create role first
  - or restore with `--no-owner`

Permission denied

- Occurs when restore role lacks privileges.
- **Fix:**
  - restore as superuser
  - or adjust ownership and grants

Object already exists

- Occurs when restoring into a non-empty database.
- **Fix**:
  - drop and recreate database
  - or clean objects manually

Useful restore options

Skip ownership:

```
psql -d newdb -f backup.sql --set ON_ERROR_STOP=on

# This command connects to the newdb database and runs all SQL commands from the
backup.sql file. If any error occurs while executing the file, PostgreSQL immediately
stops instead of continuing, which helps avoid ending up with a half-restored or broken
database.
```

- **For controlled restores**:
  - run schema first
  - then data

Post-restore tasks (often forgotten)

- **After restore, I always run:**

```
ANALYZE;

# This command tells PostgreSQL to scan the tables and update statistics about the data.
These statistics help the query planner choose better and faster execution plans for
future queries.
```

This regenerates statistics and improves performance.

- **I also verify:**
  - row counts
  - application connectivity
  - basic queries
- Restoring without verification is incomplete.

Why restore takes time

- **SQL restore:**
  - executes millions of INSERTs
  - rebuilds indexes
  - processes constraints
- This is slower than physical restore by design.

When `psql` restore is the right choice

- **I use `psql` restore when:**
  - restoring plain SQL dumps
  - migrating data
  - doing partial or selective rebuilds
  - debugging schema issues
- It gives full visibility and control.

Common DBA mistake

- Assuming backup success means restore success.
- A backup is only valid if restore completes cleanly.

Final mental model

- Restore = replay SQL
- psql = execution engine
- errors must be fixed immediately
- verification is mandatory

One-line explanation

Restoring a SQL dump means executing the exported SQL commands using psql to rebuild the database logically.

04 `pg_restore` and Selective Restore in PostgreSQL

In simple words

- `pg_restore` is used to restore logical backups that were created in **custom**, **directory**, or **tar** format.
- **Unlike plain SQL restore, `pg_restore` gives control**:
  - what to restore
  - how to restore
  - how fast to restore
- This makes it very powerful for real DBA work.

Why `pg_restore` exists

- **Plain SQL dumps:**
  - must be restored fully
  - execute line by line
  - cannot skip objects easily

`pg_restore` exists to solve these problems.

- **It works only with non-plain dump formats:**
  - `-Fc` (custom)
  - `-Fd` (directory)
  - `-Ft` (tar)

How `pg_restore` works internally

- **`pg_restore`:**
  - reads dump metadata
  - understands database objects
  - decides restore order
  - executes commands selectively
- It does **not** blindly replay SQL like `psql`.

Basic `pg_restore` syntax

```
pg_restore -d target_db backup.dump
```

This restores everything from the dump into `target_db`.

Creating a compatible dump for `pg_restore`

```
pg_dump -Fc dbname > dbname.dump
```

Without this format, `pg_restore` cannot be used.

Listing dump contents (very important)

Before restoring, I always inspect the dump:

```
pg_restore -l dbname.dump

# This command lists the contents of the dbname.dump backup file. It lets you see what objects are inside the dump—like tables, schemas, functions—before you decide what or how to restore.
```

- **This shows:**
  - tables
  - schemas
  - indexes
  - functions
  - extensions
- It helps decide what to restore.

Restoring specific objects

Only one table:

```
pg_restore -t customers -d target_db dbname.dump
```

Only one schema:

```
pg_restore -n sales -d target_db dbname.dump
```

Selective restore is not possible with plain SQL dumps.

Excluding objects during restore

Exclude table:

```
pg_restore --exclude-table=logs -d target_db dbname.dump
```

Exclude schema:

```
pg_restore --exclude-schema=test -d target_db dbname.dump
```

This is useful in debugging and migrations.

Restoring schema and data separately

Schema only:

```
pg_restore -s -d target_db dbname.dump
```

- **Data only:**

```
pg_restore -a -d target_db dbname.dump
```

This allows controlled restore sequences.

Parallel restore (big performance boost)

```
pg_restore -j 4 -d target_db dbname.dump

# This command restores the dbname.dump file into the target_db database using 4 parallel
jobs. It speeds up the restore process by loading multiple objects at the same time,
which is especially useful for large databases.
```

- **Meaning:**
  - `-j 4` = use 4 parallel jobs
- Parallel restore:
  - speeds up large restores
  - requires directory or custom format

Handling ownership and permissions

Skip ownership:

```
pg_restore --no-owner -d target_db dbname.dump

# This command restores the dump into target_db without trying to set original object
owners. It’s useful when restoring into a database where the original roles don’t exist
or when you want all objects to belong to the current user.
```

Skip privileges:

```
pg_restore --no-privileges -d target_db dbname.dump

# This command restores the dump into target_db without restoring GRANT and REVOKE
permissions. It’s useful when you want to handle access control separately or avoid
permission errors during restore.
```

Very common when restoring to test or staging.

Common `pg_restore` failures

- **pg_restore fails when:**
  - roles do not exist
  - target database is missing
  - permissions are insufficient
  - objects already exist
- Most issues are environment-related, not tool-related.

When `pg_restore` is the right tool

- **I use pg_restore when:**
  - restoring large databases
  - restoring selective objects
  - doing migrations
  - minimizing restore time
- It offers control and speed.

When `pg_restore` is NOT useful

- **pg_restore cannot:**
  - restore plain SQL dumps
  - restore physical backups
  - bypass permission rules
- Tool choice must match dump format.

Final mental model

- `pg_dump` creates structured dumps
- `pg_restore` understands dump structure
- selective restore saves time
- parallel restore improves performance

One-line explanation

`pg_restore` restores custom-format logical backups with fine-grained control over objects, order, and performance.

05 PostgreSQL Dump Formats: -Fp, -Fc, -Fd, -Ft (Explainable Guide)

In simple words

- When I take a logical backup with `pg_dump`, I must choose **how the backup is stored**.
- That choice is called the **dump format**.
- The format decides:
  - file structure
  - restore speed
  - flexibility
  - whether selective and parallel restore is possible
- Choosing the wrong format is a common DBA mistake.

Overview of available dump formats

- PostgreSQL supports four main dump formats:
  - `-Fp` → Plain SQL (default)
  - `-Fc` → Custom format
  - `-Fd` → Directory format
  - `-Ft` → Tar format
- Each format has a specific purpose.

Plain format (`-Fp`)

What it is

A human-readable SQL file containing CREATE, INSERT, and GRANT statements.

```
pg_dump -Fp mydb > mydb.sql
```

Characteristics

- text file
- readable and editable
- restored using `psql`

Pros

- very simple
- easy to inspect or modify
- no special restore tool needed

Cons

- largest file size
- slow restore
- no selective restore
- no parallel restore

When I use it

- small databases
- learning and debugging
- manual inspection needed

Custom format (`-Fc`)

What it is

A compressed binary dump designed specifically for PostgreSQL.

```
pg_dump -Fc mydb > mydb.dump
```

Characteristics

- binary format
- requires `pg_restore`
- internally structured

Pros

- smaller size
- faster restore
- supports selective restore
- supports parallel restore

Cons

- not human-readable
- cannot be edited manually

When I use it

- production backups
- medium to large databases
- when restore speed matters

Directory format (`-Fd`)

What it is

A folder containing separate files for database objects.

```
pg_dump -Fd mydb -f mydb_dir
```

Characteristics

- one directory, many files
- best for parallel restore

Pros

- fastest restore
- highest flexibility
- ideal for very large databases

Cons

- not a single file
- harder to move manually

When I use it

- very large databases
- enterprise systems
- time-critical restores

Tar format (`-Ft`)

What it is

A `tar` archive containing dump contents.

```
pg_dump -Ft mydb > mydb.tar
```

Characteristics

- single archive file
- intermediate flexibility

Pros

- single file
- supports `pg_restore`

Cons

- slower than custom and directory formats
- less commonly used

When I use it

- when I need a single file but want `pg_restore` features

Restore tool comparison

Dump Format	Restore Tool	Selective Restore	Parallel Restore
-Fp	psql	No	No
-Fc	pg_restore	Yes	Yes
-Fd	pg_restore	Yes	Yes (Best)
-Ft	pg_restore	Yes	Limited

Performance reality

- Backup speed is similar across formats
- Restore speed varies significantly
- Parallel restore makes the biggest difference

Format choice matters most during restore, not backup.

DBA recommendation (real world)

- Small DB → `-Fp`
- Medium / Large DB → `-Fc`
- Very Large / Mission-critical DB → `-Fd`

Avoid default plain format in production unless you know why you are using it.

Common mistakes to avoid

- Using plain format for huge databases
- Not planning restore strategy
- Choosing format without testing restore

Backup format must match recovery expectations.

Final mental model

- Dump format defines restore power
- pg_restore needs structured formats
- Parallel restore saves hours
- Production ≠ plain SQL

One-line explanation

PostgreSQL dump formats define how backups are stored and restored, directly affecting flexibility, restore speed, and recovery options.

06 Streaming Backups Between Servers in PostgreSQL

In simple words

- Streaming backup means copying data from one PostgreSQL server to another **without creating an intermediate dump file**.
- Data flows directly from source to target using a pipe.
- This is fast, clean, and very useful for migrations.

Why streaming backups exist

- **Creating dump files:**
  - needs disk space
  - takes extra time
  - creates cleanup work
- **Streaming avoids this by:**
  - reading from source
  - writing to target immediately

No file sits in the middle.

Most common streaming pattern

```
pg_dump source_db | psql -d target_db
```

What happens internally:

- `pg_dump` reads data from source
- output is sent through pipe
- psql receives and executes SQL
- target database is rebuilt live

Streaming between two different servers

```
pg_dump -h source_ip -U src_user source_db \
| psql -h target_ip -U tgt_user -d target_db
```

This works across:

- different machines
- different data centers
- different PostgreSQL versions

Why this works safely

- pg_dump uses a consistent snapshot
- psql executes commands in order
- data integrity is preserved

Users can keep working on source during streaming.

When streaming is a good choice

I use streaming when:

- migrating databases
- cloning production to staging
- disk space is limited
- one-time transfers are needed

It is fast and simple.

Limitations of streaming backups

Streaming backups:

- cannot be resumed if interrupted
- provide no backup file for reuse
- depend heavily on network stability

If network drops, restore fails.

Streaming with compression

```
pg_dump source_db | gzip | gunzip | psql -d target_db

# This command takes a live backup of source_db, compresses it, immediately decompresses
it, and pipes it straight into target_db. In simple words, it copies data from one
database to another in a single flow without creating any dump file on disk.
```

- Used when network bandwidth is limited.
- CPU cost increases.

Handling errors during streaming

If error occurs:

- streaming stops immediately
- partial data may exist

Best practice:

- restore into empty database
- drop and retry on failure

Streaming vs file-based backups

Streaming:

- faster
- less disk usage
- single-use

File-based:

- reusable
- resumable
- safer for long-term storage

Choose based on situation.

DBA checklist before streaming

Before streaming I ensure:

- target DB is empty
- required roles exist
- permissions are correct
- network is stable

Preparation prevents failures.

Final mental model

- Streaming = pipe + live restore
- No files in between
- Fast but fragile
- Best for migrations

One-line explanation

Streaming backup transfers a logical dump directly from one PostgreSQL server to another using pipes, avoiding intermediate files.

07 pg_dumpall and Cluster-Level Backups in PostgreSQL

In simple words

`pg_dumpall` is used to back up **the entire PostgreSQL cluster**, not just one database.

It captures:

- all databases
- all roles and users
- role memberships
- global objects

If `pg_dump` backs up *data*, `pg_dumpall` backs up the *identity of the cluster*.

Why `pg_dumpall` exists

Backing up only databases is not enough.

Real restores fail because:

- roles do not exist
- ownership is missing
- permissions break

`pg_dumpall` exists to solve this by capturing **global objects**.

What `pg_dumpall` actually backs up

`pg_dumpall` includes:

- CREATE ROLE statements
- role passwords (hashed)
- role memberships
- CREATE DATABASE statements
- all databases (as SQL)

It does NOT back up:

- server configuration files
- physical WAL or data files

How `pg_dumpall` works internally

- connects as a superuser
- reads global system catalogs
- dumps roles and privileges first
- dumps each database sequentially

All output is written as **plain SQL**.

Basic `pg_dumpall` usage

```
pg_dumpall > cluster_backup.sql
```

This creates one large SQL file containing everything.

Role requirements (very important)

`pg_dumpall` **must run as a superuser**.

Reason:

- global catalogs are restricted
- role passwords and memberships require superuser access

Non-superuser runs will fail.

Restoring from `pg_dumpall`

Restore is done using psql:

```
psql -f cluster_backup.sql postgres
```

What happens:

- roles are created first
- databases are created
- database contents are restored

Restoring should be done on a clean cluster.

Common `pg_dumpall` problems

- extremely large output files
- slow restore
- no parallel restore support
- hard to debug failures

Because everything is in one file, recovery is all-or-nothing.

`pg_dumpall` vs `pg_dump` (real difference)

pg_dump:

- single database
- flexible formats
- selective restore

`pg_dumpall`:

- entire cluster
- plain SQL only
- no selective restore

They serve different purposes.

When I actually use `pg_dumpall`

I use `pg_dumpall` mainly for:

- backing up roles and global objects
- disaster recovery documentation
- rebuilding a cluster from scratch

For data backups, I still prefer pg_dump.

Best practice (important)

Instead of using `pg_dumpall` alone:

- back up databases with `pg_dump`
- back up roles separately with `pg_dumpall --globals-only`

Example:

```
pg_dumpall --globals-only > globals.sql

# This command backs up only the global objects of the PostgreSQL cluster, such as roles and tablespaces. It does not include any database data, making it useful when you want to preserve users and permissions separately from databases.
```

This gives more control during restore.

Security warning

`pg_dumpall` output contains:

- role definitions
- password hashes

Backup files must be:

- protected
- access-controlled
- stored securely

Final mental model

- `pg_dump` = database-level backup
- `pg_dumpall` = cluster identity backup
- roles matter as much as data

One-line explanation

`pg_dumpall` creates a logical backup of all databases and global objects in a PostgreSQL cluster using plain SQL.

08 Post-Restore Tasks: ANALYZE, VACUUM, and Verification

In simple words

After a restore, the database *looks* fine but it usually **does not perform fine**.

Post-restore tasks exist to:

- fix planner statistics
- clean internal states
- verify data correctness
- make the database production-ready

Restore without post-restore work is incomplete.

Why performance is bad after restore

During logical restore:

- data is inserted in bulk
- indexes are rebuilt
- planner statistics are **empty or outdated**
- Without fresh statistics, PostgreSQL guesses wrong plans.
- That is why queries feel slow even though data is present.

`ANALYZE` (most important step)

What `ANALYZE` does

`ANALYZE` scans tables and builds statistics about:

- row counts
- data distribution
- column selectivity

The query planner depends on these stats to choose indexes.

When I run it

Immediately after restore.

```
ANALYZE;

# This command tells PostgreSQL to scan the tables and update statistics about the data.
These statistics help the query planner choose better and faster execution plans for
future queries.
```

For large systems, this single command fixes most post-restore issues.

`VACUUM` after restore

What `VACUUM` does

- cleans dead tuples
- updates visibility map (*The visibility map is an internal PostgreSQL structure that tracks which data pages contain only visible rows. It helps PostgreSQL skip unnecessary table scans during VACUUM and SELECT queries, making reads faster and reducing extra work.*)
- helps index-only scans

After a fresh restore, heavy `VACUUM` is usually **not required**, but a light `vacuum` helps internal bookkeeping.

```
VACUUM;

# This command cleans up dead rows left behind by updates and deletes, freeing space and
keeping the database healthy. It also helps PostgreSQL maintain good performance by
preventing tables from becoming bloated.
```

Do **not** run aggressive `VACUUM FULL` right after restore.

Why VACUUM FULL is dangerous

```
VACUUM FULL;

# This command completely rewrites the table to remove dead rows and reclaim disk space
back to the operating system. It locks the table while running, so it’s used only when
space recovery is more important than availability.
```

- locks tables
- rewrites data
- blocks concurrent access

After restoration, it usually adds risk without benefit. Use it only when space reclaim is required.

Refreshing sequence values

After restore, sequences may become out of sync.

Check:

```
SELECT last_value FROM my_table_id_seq;
```

Fix if needed:

```
SELECT setval('my_table_id_seq', MAX(id)) FROM my_table;
```

This prevents duplicate key errors.

- After a restore, sequence values can be out of sync with table data.
- The first query checks the current sequence value, and the second one resets the sequence to the highest `id` present in the table, so future inserts don’t fail with duplicate key errors.

Validating data correctness

I always verify:

- table row counts
- critical business tables
- foreign key integrity

Example:

```
SELECT count(*) FROM important_table;
```

Never assume restore was perfect.

Checking application connectivity

Before declaring success:

- connect application users
- run basic queries
- confirm permissions

Restore is successful only if applications work.

Autovacuum considerations

Autovacuum may:

- start running after restore
- consume I/O unexpectedly

In large restores:

- monitor autovacuum
- avoid tuning changes immediately

Let the system stabilize first.

Logging and monitoring

After restore, I check:

- PostgreSQL logs
- error messages
- slow queries

Hidden issues appear only in logs.

Common DBA mistake

Declaring restore complete after SQL finishes.

Correct mindset:

> Restore ends only after performance and correctness are verified.

Final mental model

- Restore builds data
- ANALYZE builds intelligence
- VACUUM maintains health
- Verification builds confidence

One-line explanation

After restore, a DBA must run ANALYZE, verify data, and check system health to ensure correct performance and consistency.

09 Common SQL Dump Mistakes and Failure Scenarios in PostgreSQL

In simple words

- Most backup failures are not tool problems.
- They are **human and process mistakes**.

Mistake 1: Assuming backup success means restore success

Many DBAs run:

```
pg_dump mydb > backup.sql
```

If the command finishes, they assume everything is fine.

Reality:

- backup file may be incomplete
- restore may fail due to roles, permissions, or dependencies

Correct approach:

> A backup is valid only after a successful restore test.

Mistake 2: Not checking permissions before `pg_dump`

`pg_dump` fails if it cannot read **any single object**.

Common causes:

- missing access to one schema
- view referencing inaccessible table
- extension privilege issue

Correct approach:

- run `pg_dump` as database owner or superuser
- verify permissions in advance

Mistake 3: Using plain SQL for very large databases

Plain format:

- generates huge files
- restores slowly
- cannot run in parallel

Using it for multi-GB databases leads to:

- long downtime
- restore failures

Correct approach:

- use custom or directory formats
- enable parallel restore

Mistake 4: Forgetting roles and global objects

Database restore fails silently when:

- roles are missing
- ownership cannot be assigned

Symptoms:

- restore completes with warnings
- application fails later

Correct approach:

- restore roles first
- use `pg_dumpall --globals-only`

Mistake 5: Restoring into a dirty database

Restoring into a database that already contains objects leads to:

- object already exists errors
- partial restore
- inconsistent state

Correct approach:

- always restore into a clean database
- drop and recreate if unsure

Mistake 6: Ignoring restore errors

During restore, errors scroll quickly.

Ignoring them results in:

- missing tables
- broken foreign keys
- silent data loss

Correct approach:

- stop on error
- fix root cause
- restart restore

Mistake 7: Skipping post-restore steps

After restore:

- statistics are missing
- sequences may be wrong

Skipping `ANALYZE` causes:

- slow queries
- wrong plans

Correct approach:

- always run `ANALYZE`
- verify sequences and counts

Mistake 8: Backups stored on same server

Storing backups on the same server means:

- disk failure = data + backup lost

Correct approach:

- store backups off-host
- use separate storage or remote systems

Mistake 9: No monitoring of backup jobs

Backups may:

- silently fail
- stop due to disk full
- hang for hours

Correct approach:

- log backup output
- monitor duration and size
- alert on failures

Mistake 10: Never testing restore under pressure

Real disaster recoveries fail because:

- restore steps were never practiced
- documentation is missing
- decisions are made in panic

Correct approach:

- schedule restore drills
- document recovery steps

Final mental model

- Tools rarely fail
- Process failures cause data loss
- Restore testing is non-negotiable
- Preparation beats panic

One-line explanation

Most SQL dump failures happen due to permission issues, wrong formats, missing roles, or untested restore processes rather than tool limitations.