Schema Design Best Practices

Normalization vs. Denormalization: Short Notes

Normalization

- **Definition**: A systematic process of organizing relational database tables to minimize data redundancy and dependency by dividing large tables into smaller, related ones. Follows normal forms (NF): 1NF (atomic values), 2NF (no partial dependencies), 3NF (no transitive dependencies), up to BCNF/6NF for stricter rules.
- **Key Benefits**:
    - Reduces storage waste (e.g., avoids duplicating customer addresses across orders).
    - Enhances data integrity via constraints (e.g., foreign keys prevent orphans).
    - Simplifies updates/deletes (change data in one place).
- **Drawbacks**: Increases join complexity, potentially slowing reads in query-heavy apps.
- **Best For**: OLTP systems with frequent writes (e.g., e-commerce order processing) where accuracy > speed.

Denormalization

- **Definition**: The inverse: Intentionally adding redundant data (e.g., embedding address in order table) to optimize for faster queries by reducing joins.
- **Key Benefits**:
    - Boosts read performance (e.g., single-table lookups for reports).
    - Scales well for analytics/OLAP (e.g., denormalized fact tables in data warehouses).
    - Minimizes network I/O in distributed systems.
- **Drawbacks**: Raises storage needs and update risks (e.g., must sync redundant fields to avoid inconsistencies).
- **Best For**: Read-intensive workloads (e.g., dashboards, search engines) where speed > minimal storage.

Comparison Table

Aspect	**Normalization**	**Denormalization**
Goal	Integrity & efficiency in writes	Performance in reads
Redundancy	Minimized	Introduced strategically
Joins	Many (slower reads)	Few (faster reads)
Maintenance	Easier (centralized updates)	Harder (sync redundancies)
Use Case	Transactional DBs (e.g., PostgreSQL OLTP)	Analytical DBs (e.g., reporting views)

PostgreSQL Table Design Best Practices

- **Define Primary Keys Always**: Every table should have a unique, non-NULL primary key (e.g., SERIAL or BIGINT id PRIMARY KEY) to identify rows efficiently and support relationships; use composite keys for multi-column uniqueness like (order_id, product_id).
- **Enforce Foreign Keys for Integrity**: Use FOREIGN KEY constraints

- **Apply Comprehensive Constraints**: Include NOT NULL for required columns, UNIQUE for duplicates prevention, CHECK for value validation
- **Select Precise Data Types**: Choose the most specific and efficient types (e.g., INTEGER or BIGINT for IDs, TIMESTAMP WITH TIME ZONE for dates, JSONB for semi-structured data) to minimize storage and boost query speed; avoid generic TEXT unless necessary.
- **Column Placement:** For Better performance of queries place all the fixed size columns at left side and variable size at right side
- **Index Strategically**: Create indexes on frequently queried/filtered columns (e.g., WHERE clauses, joins) and high-selectivity fields; prefer B-tree for most cases, and use GIN for JSONB or full-text; monitor with EXPLAIN ANALYZE to avoid over-indexing.
- **Plan for Large Tables with Partitioning**: For tables exceeding 100GB, use declarative partitioning (range, list, hash) from the outset to improve query performance and maintenance, especially with PostgreSQL 17's enhanced logical replication.
- **Start Simple and Iterate**: Begin with a minimal schema focusing on core needs, then expand; use COMMENT ON for documentation and test with realistic data volumes to validate design.

Connection Pooling

PGBouncer

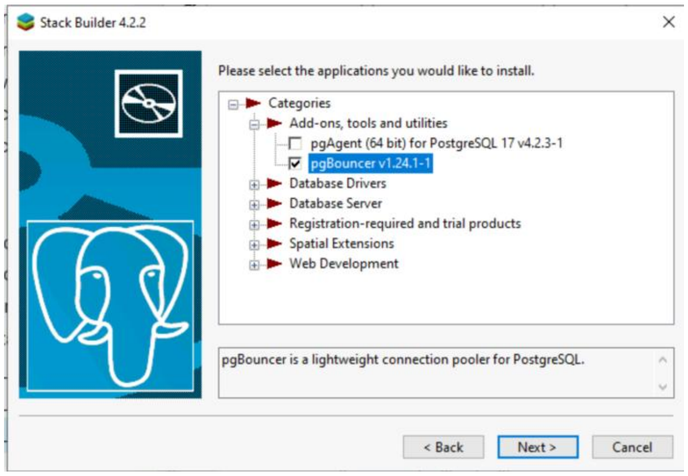
PgBouncer is a lightweight, open-source connection pooler for PostgreSQL that acts as a middleware layer between client applications and the PostgreSQL server. It manages a pool of reusable database connections to minimize the overhead of establishing new connections, which can be resource-intensive in high-concurrency environments. By reusing existing connections, PgBouncer improves performance without altering transaction semantics, supporting pooling modes like session, transaction, and statement.

Use Cases

- High-Concurrency Applications: Ideal for web apps or microservices with many short-lived connections, reducing PostgreSQL's connection load and preventing "too many connections" errors.
- Traffic Burst Handling: Isolates and manages sudden spikes in connections
- Performance Tuning: Enhances throughput in read-heavy or OLTP workloads by efficient pooling.

Install on Windows

Install via Stack Builder or download from EDB website



Please select the applications you would like to install.

- [ ] pgAgent (64 bit) for PostgreSQL 17 v4.2.3-1
- [✓] pgBouncer v1.24.1-1

pgBouncer is a lightweight connection pooler for PostgreSQL.

Install on RHEL

Download from following link
https://dl.fedoraproject.org/pub/epel/9/Everything/x86_64/Packages/p/pgbouncer-1.24.1-2.el9.x86_64.rpm

Install:

```
Sudo dnf -y localinstall  pgbouncer-1.24.1-2.el9.x86_64.rpm

Sudo systemctl enable pgbouncer
```

Configuration

Main configuration file is /etc/pgbouncer/pgbouncer.ini

```
[pgbouncer]
listen_addr = *
listen_port = 6432
pool_mode = session
max_client_conn = 1000
default_pool_size = 20
auth_type = scram-sha-256
auth_file = /etc/pgbouncer/userslist.txt
auth_query = SELECT rolname, rolpassword FROM pg_authid WHERE rolname=$1
logfile = /var/log/pgbouncer/pgbouncer.log

databases]
mydb = host=localhost port=5432 dbname=mydb

[users]
myuser = pool_size=10
```

Start PgBouncer

```
sudo systemctl start pgbouncer
```

Testing

```
psql -h localhost -p 6432 -U user -d mydb
/usr/pgsql-17/bin/pgbench  -h 127.0.0.1 -U postgres -p 6432  -C -c 90 -t 100 tpc
```

PGPool II

Pgpool-II is an open-source middleware for PostgreSQL that sits between client applications and one or more PostgreSQL servers. It provides connection pooling to reduce overhead, load balancing

Use Cases

- Load Balancing & Scaling: Distribute read-heavy workloads across multiple read-only replicas while routing writes to the primary, ideal for web apps or analytics.
- Connection Pooling: Handles high-concurrency scenarios by reusing connections, reducing PostgreSQL's resource strain.
- Parallel Querying: Speeds up complex queries by executing them across

ProxySQL

ProxySQL is a high-performance, open-source, protocol-aware proxy and load balancer designed primarily for MySQL , MariaDB. PostgreSQL support introduced in version 3.0 (June 2025). It acts as a middleware layer between clients and database servers, enabling intelligent query routing, connection multiplexing, and traffic management without requiring application changes. ProxySQL uses a modular architecture with runtime configuration via SQL-like commands, making it lightweight (under 10MB memory) yet feature-rich for optimizing database infrastructures.

Use Cases

- **Read/Write Splitting:** Automatically routes write queries to the primary server and reads to replicas, improving scalability
- **Load Balancing & Sharding:** Distributes traffic across multiple backends based on rules, weights, or query patterns, ideal for horizontal scaling in high-traffic apps.
- **High Availability:** Provides failover with zero downtime by monitoring server health and seamlessly switching endpoints.
- **Query Optimization:** Implements caching, firewalling, and transformation (e.g., rewriting slow queries) to reduce latency and enhance security in multi-tenant environments.
- **Multi-Database Management:** Handles traffic for multiple applications or databases from a single proxy, simplifying complex cloud-native deployments.

Linux Optimizations For PostgreSQL

Linux / Unix provides various kernel parameters to control kernel behavior which will benefit Postgres.

Kernel Memory Parameters

`kernel.shmmax` : Maximum size of shared Memory Segment size. Default size on latest Linux system is very large enough to take care any workload. Not recommended to change but if decided to change set atleast 50% of Physical RAM installed on server.

`kernel.shmall` : It limits the available shared memory on the system. It is set in pages rather bytes. The default on latest Linux is good enough for all types of workloads

`kernel.shmmni` : The default of 4096 is good enough.

`vm.swappiness` : This parameter controls the swappiness on a Linux system. Valid values from 0 to 100. 0 being no swappiness and 100 being very aggressive swappiness. Setting to high value degrades the performance as system considers swapping more often and a value of 0 may lead to OOM (Out Of Memory). We can set it to a smaller value of 10 or less. default is 60.

`vm.overcommit_memory` : Postgres (or any other application) memory allocation request to OS require the requested amount of free memory on system, if not, there is a risk of getting killed by OOM. We can control this by allowing OS to over commit memory (allocate memory beyond installed mem). Three values possible

- 0 means let kernel decides on overcommit
- 1 Allocate memory without any checks
- 2 Don't over commit more than vm.overcommit_ratio

A value of 2 works best for PostgreSQL as it is limits the use of SWAP and better utilization of RAM.

`vm.overcommit_ratio` : If the **vm.overcommit_memory** set 2 then system will over commit this much % RAM. Default is 50% of RAM

Linux Huge Pages

Default page size in linux is 4k which may not be very efficient in handling a large shared_buffer. From linux kernel version 2.6 onward linux integrated the support for huge pages where it can support page sizes unto 1GB.

Using this feature in postgres requires enabling and setting huge page size on Linux and configuring PostgreSQL to use huge pages.

Configuring huge pages on Linux for PostgreSQL

Step 1: Find the postgres Postmaster pid. Run the following command and note down the pid

```
head -1  /var/lib/pgsql/13/data/postmaster.pid
```

or alternatively we can use **`ps -ef|grep postgres`** to find PID.

Step 2: Find the memory status from proc filesystem for this PID

```
grep  -i vmpeak  /proc/<postmaster pid>/status
```

Step 3: find the huge-age size on the system

```
grep -i hugepagesize /proc/meminfo
```

Step 4: Calculate the appropriate hugepage size for system to support Postgres

```
<vmpeak size from step 2> / <hugepage size in Step 3>
```

Step 5: Set huge-age size in Linux to the result derived in step 4

```
sysctl -w vm.nr_hugepages= <result from step 4>
```

and also make entry in **`/etc/sysctl.conf`**

Enabling Hugepage support in PostgreSQL

One the huge pages are enabled in Linux we can enable huge page support in postgresql by following setting is **`postgresql.conf`**

```
huge_pages = on
```

It requires service restart.

Hardware Optimization

1. Always use RAID 10 for postgreSQL
2. Invest in ECC RAM
3. More cores gives more performance make sure at least 4 cores are available for Postgres
4. Invest in faster disk and SSD storage
5. Have multiple filesystem available on system to improve IO
6. More spindles (DISKs) translates to more performance.
7. User GBPS network cards as it will speed up data transfers to clients
8. Use more than one network interface as it provides network redundancy