How PostgreSQL Can Help You Enforce Best Practices

Jimmy Angelakos
Senior Solutions Architect

PGConf NYC 2023-10-05
What is this talk?

- IT systems can have commonalities and share similar best practices
- We will discuss PostgreSQL best practices
- How these translate to best practices in general

- Not all-inclusive or in-depth!
- May be preachy (for a reason)
We will go over:

- Proper data types
- Locking
- High concurrency & transaction rate
- Home-brewing distributed systems (don’t)
- Tracking resource usage
- Security
- High Availability

... and some other stuff
Using the proper data types
Data types and keys

- Use the correct data type for each thing you’re storing
- e.g. don’t store datetime as text
  - Waste of space, not indexable, no calculations
- Be aware of the data type storage requirements
- Don’t use more storage than you need
  - e.g. 'open'/''closed' vs boolean true/false
  - It adds up!
# Data type sizes

<table>
<thead>
<tr>
<th>Data type</th>
<th>Size in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>bigint</td>
<td>8</td>
</tr>
<tr>
<td>timestamptz</td>
<td>8</td>
</tr>
<tr>
<td>double precision</td>
<td>8</td>
</tr>
<tr>
<td>uuid</td>
<td>16</td>
</tr>
<tr>
<td>text</td>
<td>1 + string bytes (+4 if &gt; 127 bytes)</td>
</tr>
</tbody>
</table>
Using the right PK data type (i)

CREATE TABLE test (id bigint, content text);
CREATE \
timing
Timing is on.
INSERT INTO test SELECT generate_series(1,100000000), 'test';
INSERT 0 100000000
Time: 90202.739 ms (01:30.203)
ALTER TABLE test ADD PRIMARY KEY (id);
ALTER TABLE
Time: 38123.742 ms (00:38.124)
Using the right PK data type (ii)

```sql
SELECT pg_column_size(id) FROM TEST LIMIT 1;
pg_column_size
----------------
     8
\di+ test_pkey
```

List of relations

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Table</th>
<th>Persistence</th>
<th>Access method</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>test_pkey</td>
<td>index</td>
<td>foo</td>
<td>test</td>
<td>permanent</td>
<td>btree</td>
<td>2142 MB</td>
<td></td>
</tr>
</tbody>
</table>
Using the right PK data type (iii)

CREATE TABLE test (id uuid, content text);
CREATE
\timing
Timing is on.
INSERT INTO test
SELECT gen_random_uuid, 'test' FROM generate_series(1,100000000);
INSERT 0 100000000
Time: 387838.234 ms (06:27.838) +330%
ALTER TABLE test ADD PRIMARY KEY (id);
ALTER TABLE
Time: 67710.091 ms (01:07.710) +78%
Using the right PK data type  (iv)

SELECT pg_column_size(id) FROM TEST LIMIT 1;

```
pg_column_size
----------------
  16
\di+ test_pkey
```

List of relations

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Table</th>
<th>Persistence</th>
<th>Access method</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>test_pkey</td>
<td>index</td>
<td>foo</td>
<td>test</td>
<td>permanent</td>
<td>btree</td>
<td>3008 MB</td>
<td>+40%</td>
</tr>
</tbody>
</table>

(1 row)
Use TIMESTAMPTZ

- Default is TIMESTAMP (WITHOUT TIME ZONE)
  - a.k.a. naïve timestamps, no time zone information
  - Arithmetic between timestamps entered at diff time zones is meaningless, gives wrong results
  - Don’t use to store UTC, DB doesn’t know it’s UTC

- TIMESTAMP WITH TIME ZONE
  - Stores a moment in time
  - Arithmetic works correctly
  - Displays in your time zone, or AT TIME ZONE
Use TIMESTAMPTZ as PK

- Natural primary key for time series data
- Do you need a surrogate (artificial) key?
- Really compact storage
- Partitions and indexes wonderfully
  - Also: Block range indexes (BRIN)
    
    For 106308001 records:
    - btree index is 2277 MB
    - brin index is 192 kb
“Relational JSON”

- Anti-pattern

```sql
SELECT json_account -> 'id'
FROM accounts, sales
WHERE json_account ->> balance::int < 20000
AND json_sale ->> 'account_id' = json_account ->> 'id'
AND json_sale ->> 'amount'::int > 10000;
```

- NoSQL / “schemaless” was meant to eliminate the need for JOINs
Choosing the right encoding
SQL_ASCII

- Is not a database encoding
- No encoding conversion or validation!
  - Byte values 0–127 interpreted as ASCII
  - Byte values 128–255 uninterpreted
- Setting behaves differently from other character sets
- Can end up storing a mixture of encodings
  - With no way to recover original strings
UTF8

● Your safest bet
● If you’re migrating, convert to UTF8
● Postgres has conversion functions available
● Mind your collations
  - Sort order
  - Character classification
Locking and how it affects performance
Locks in PostgreSQL

- MVCC: Multi-Version Concurrency Control
- Rather than locking for high concurrency and high performance
  - Reading never waits
  - Writing doesn’t block reading, reading doesn’t block writing
  - Each write creates a new version of tuple
- Snapshot isolation: Timestamps & Transaction IDs (XIDs)
Explicit locks

- Table-level (e.g. SHARE) or row-level (e.g. FOR UPDATE)
- Conflict with other lock modes (e.g. ACCESS EXCLUSIVE with ROW EXCLUSIVE)
- Block read/write access totally leading to waits
- Disastrous for performance
  - Unless your application is exquisitely crafted
  - Hint: it isn’t
Lightweight Locks (LWLocks)

- Protect data in shared memory
  - Multi-process system
  - Ensure consistent reads/writes
  - Shared, Exclusive modes
- Enable fast MVCC
  - Generally held briefly
  - Sometimes protect I/O
To lock or not to lock?

- Avoid explicit locking!
- Use SSI (Serializable Snapshot Isolation: SERIALIZABLE isolation level)
- Make application tolerant
  - Allow it to fail and retry
- Slightly reduced concurrency, but:
  - No blocking, no explicit locks needed (SIReadLocks, rw-conflicts)
  - Best performance choice for some applications
Controlling concurrency & transaction rate
Concurrency: Connections

- Don’t overload your server for no reason
  - `max_connections = 5000`
- Every client connection spawns a separate backend process
  - IPC via semaphores & shared memory
  - Risk: CPU context switching
- Accessing the same objects from multiple connections may incur many LWLocks
  - Lots of lockers slow each other down
Controlling concurrency

- Pre-PG 13: Snapshot contention
  - Each transaction has an MVCC snapshot – even if idle!
- Parallelization
  - Count your cores!
  - `max_parallel_workers(_per_gather)`
- Monitoring: `pg_stat_activity` (look for `wait_event_type: LWLock`)
Connection pooling

- Rule of thumb: No more than 4 connections per core
- e.g. PgBouncer between application & DB
  - Allow fewer connections in, make the rest queue for their turn
  - “Throttle” or introduce latency on the application side, to save your server performance
- Sounds counter-intuitive!
  - Doesn’t necessarily slow anything down
  - Queries may execute faster
High transaction rate

- Postgres assigns an identifier to each transaction
  - Unsigned 32-bit int (4.2B values), circular space
  - XID wraparound
- Heavy OLTP workloads can go through 2.1B transactions quickly
  - Autovacuum
  - Can batching help? Does application really need to commit everything atomically?
  - Batch size 1000 will have 1/1000th the burn rate
Tracking resource usage
PostgreSQL statistics

• Cumulative Statistics System (FKA Statistics Collector)
  - Postgres subsystem that collects info about system activity
• Dynamic statistics (right now)
• Cumulative statistics, but can be reset
• Table/index information on row & disk block levels
• This info can be reported via views
Track over time

- For causal analysis and making predictions
  - Troubleshooting
  - Projections / futureproofing
- Log with monitoring tools
- Export with Prometheus
- Minimalist: pg_statviz extension
Home-brewing distributed systems (don’t)
Home-brewing multi-master

• Using native logical replication or pglogical 2
• Just establish a connection in each direction right?
  – Problem solved!
• Replication origins
  – Ping-pong
• Concurrency
  – Data conflicts
Conflicts

- Communication is not at light speed
- Synchronous replication or explicit locking kill performance
- Data integrity / consistency
  - Are all nodes consistent?
  - Updating a row you didn’t know was there
  - Deleting a deleted row, etc.
- Sequence management!
Serialization anomalies

• Application needs to be multi-master aware
• Write on one node, read from another
  – Inside the same application-level transaction
  – Global transaction manager
• Successful SQL operations may well be a business logic error
  – Atomicity violation
Craft the distributed system inside your application

Use standard facilities like:
- Serializable isolation level
- Two-phase commits

Why do you really need multi-master?

Use a tool that was designed for this
- Not replicators / change data capture
Configuring for production usage
Defaults are safe

- Very conservative, safest choices
- `postgresql.conf`:
  
  ```
  # WRITE-AHEAD LOG
  # - Settings -
  wal_level = replica
  fsync = on
  synchronous_commit = on
  full_page_writes = on
  ```
Defaults are (too) safe

- Safe for running on any (small) system
- For production, may be woefully inadequate

  # - Memory -
  shared_buffers = 128MB
  work_mem = 4MB

  # - Cost-Based Vacuum Delay -
  vacuum_cost_limit = 200

- Autovacuum will not be aggressive enough
Don’t log to PGDATA

- Run the risk of disk space exhaustion
- e.g. application endless loop
- This will crash Postgres
- Ideally place log files on a different filesystem
- And monitor disk usage
Applying Security best practices
Security by default  (i)

- No cleartext passwords, no access by remote hosts, SSL used if available
- `pg_hba.conf`:

```plaintext
# TYPE  DATABASE        USER            ADDRESS                 METHOD
# "local" is for Unix domain socket connections only
local   all             all                                     peer
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```
Host-Based Authentication

trust is a Very Bad Idea™
- Even for local e.g. improper user can connect to the DB
- Postgres might be fine, but other software on the same server could be compromised

Default to giving access only where strictly necessary (better safe...)

pg_hba.conf
Security by default (ii)

- No cleartext passwords, no access by remote hosts, SSL used if available
- `postgresql.conf`:
  
  ```
  # - Connection Settings -
  listen_addresses = 'localhost'

  # - Authentication -
  password_encryption = scram-sha-256

  # - SSL -
  ssl = on
  ```
listen_addresses = 'localhost'

- Listening for connections from clients
- There's a reason the default is 'localhost' (only TCP/IP loopback)
  - Make sure you only enable the interfaces and networks which you actually want to have access to the DB server
  - e.g. Internet connection on one network & private network on another interface
- Don’t advertise your presence:
  - 3,600,000 MySQL/MariaDB servers (port 3306) found exposed on the Internet in May 2022
Only give access where needed

- Use superuser only for management of global objects
  - Such as users
  - Superuser bypasses a lot of checks
- (Bad) code that’s normally harmless could be exploited in harmful way with superuser access
- Restrict database ownership to standard users
- New in PG 16: Client-side requirements, Kerberos delegation
Applying High Availability best practices
Back! Up!

- pg_dump is not a backup
- A backup that is not tested is not a backup
- A backup that is not automated is not a backup
- Use a specialized backup tool
  - Preferably one created for Postgres
  - Barman, pgBackRest, etc...
- Point in time recovery (PITR) is a great tool
High Availability

- Practice redundancy
- Use standbys with a HA tool
- e.g. RepMgr, Patroni, EFM
- Kubernetes: CloudNativePG
- Pay close attention to your architecture
  - Data centers
  - Witnesses
  - Quorum
Upgrading is important
Which version of Postgres are you on?
NEVER UPGRADE.
NEVER SURRENDER!
Why people avoid upgrading

- “It works fine now” – what about tomorrow?
- “Don’t touch it, you might break it”
  “Touch it, you can make it better – Seth Godin
- How well do you know your system?
  - Breaking is learning
- False sense of stability
- Upgrade procedure not well defined
Upgrade regularly

- Open source: updates issued rapidly
- Security updates known to roll out in a matter of hours
- Long-standing bugs undetected for years
- Triggering of unexpected behaviors in software
- Have a QA system to test upgrades regularly
- No license fees for test systems!
You may be missing out

- Stayed on PG13, didn’t get:
  - Throughput improvement for large numbers of connections
  - Streaming of large transactions
  - libpq pipelining

- Stayed on PG14, didn’t get:
  - Improved sort speed & WAL compression
  - SQL MERGE
  - Logical Replication improvements
  - JSON logging
You may be missing out

- Stay on PG15, and you won’t get:
  - Significant query performance improvements
  - Logical replication from standby servers
  - New SQL/JSON functionality
  - `pg_stat_io`
  - `pg_hba.conf` regular expressions