

# PostgreSQL Proficiency for Python People

Christophe Pettus  
PostgreSQL Experts, Inc.

[thebuild.com](http://thebuild.com)  
[pgexperts.com](http://pgexperts.com)

# Welcome!

- Christophe Pettus
- Consultant with PostgreSQL Experts, Inc.
- Based in sunny San Francisco, California.
- Technical blog: **thebuild.com**
- Twitter: **@xof**
- **christophe.pettus@pgexperts.com**

# My background.

- PostgreSQL person since 1998.
- Came to databases as an application developer and architect.
- Python/Django person since 2008.

# What is this?

- “Just enough” PostgreSQL for a Python developer.
- PostgreSQL is a rich environment.
- Far too much to learn in a single tutorial.
- But enough to be dangerous!

# The DevOps World

- “Integration between development and operations.”
- “Cross-functional skill sharing.”
- “Maximum automation of development and deployment processes.”
- “We’re way too cheap to hire real operations staff. Anyway: **Cloud!**”

# This means...

- No experienced DBA on staff.
  - Have you seen how much those people *cost, anyway?*
- Development staff pressed into duty as database administrators.
- But it's OK... it's **PostgreSQL!**

# Everyone Loves PostgreSQL!

- Fully ACID-compliant relational database management system.
- Richest set of features of any modern production RDMS.
- Relentless focus on quality, security, and spec compliance.
- Capable of very high performance.

# PostgreSQL Can Do It.

- Tens of thousands of transactions per second.
- Enormous databases (into the petabyte range).
- Support by all Python ORMs and web frameworks.



# Cross-Platform.

---

- Operates natively on all modern operating systems.
- Plus Windows.
- Scales from development laptops to huge enterprise clusters.

# Installation

# If you have packages...

- ... use them!
  - Provides platform-specific scripting, etc.
- RedHat-flavor and Debian-flavor have their own repositories.
- Other OSes have a variety of packaging systems.

# Or you can build from source.

- Works on any platform.
- Maximum control.
- Requires development tools.
- Does not come with platform-specific utility scripts (/etc/init.d, etc.).

# Other OSes.

---

- Windows: One-click installer available.
- OS X: One-click installer, MacPorts, Fink and Postgres.app from Heroku.
- For other OSes, check [postgresql.org](https://www.postgresql.org).

# Creating a database cluster.

- A single PostgreSQL server can manage multiple databases.
- The whole group on a single server is called a “cluster”.
- This is very confusing, yes. We’ll use the term “server” here.

# initdb

- The command to create a new database is called `initdb`.
- It creates the files that will hold the database.
- It doesn't automatically start the server.
- Many packaging systems automatically create and start the server for you.

# Note on Debian

- Debian-style packaging has a sophisticated cluster management system.
- Use it! It will make your life much easier.
- `pg_createcluster` instead of `initdb`



# Just Do This.

- Always create databases as UTF-8.
  - Once created, cannot be changed.
  - Converting from “SQL\_ASCII” to a real encoding is a total nightmare.
- Use your favorite locale, but not “C locale.”
- UTF-8 and system locale are the defaults.

- Built-in command to start and stop PostgreSQL.
- Frequently called by init.d, upstart or other scripts.
- -m fast is the way to go.
- Use the package-provided scripts if they exist; they do the right thing.

- Command-line interface to PostgreSQL.
- Run queries, examine the schema, look at PostgreSQL's various views.
- Get friendly with it! It's very useful for doing quick checks.

# PostgreSQL directories

- All of the data lives under a top-level directory.
- Let's call it `$PGDATA`.
  - Find it on your system, and do a `ls`.
  - The data lives in “base”.
  - The transaction logs live in `pg_xlog`.

# Configuration files.

- On most installations, the configuration files live in `$PGDATA`.
- On Debian-derived systems, they live in `/etc/postgresql/9.3/main/...`
- Find them. You should see:
  - `postgresql.conf`
  - `pg_hba.conf`

# Configuration

# Configuration files.

- Only two really matter:
  - postgresql.conf — most server settings.
  - pg\_hba.conf — who gets to log in to what databases?

# postgresql.conf

---

- Holds all of the configuration parameters for the server.
- Find it and open it up on your system.





**We're All Going To Die.**



**It Can Be Like This.**

# Important parameters.

- Logging.
- Memory.
- Checkpoints.
- Planner.
- You're done.
- No, really, you're done!

# Logging.

- Be generous with logging; it's very low-impact on the system.
- It's your best source of information for finding performance problems.

# Where to log?

- `syslog` — If you have a `syslog` infrastructure you like already.
- standard format to files — If you are using tools that need standard format.
- Otherwise, CSV format to files.

# What to log?

```
log_destination = 'csvlog'  
log_directory = 'pg_log'  
logging_collector = on  
log_filename = 'postgres-%Y-%m-%d_%H%M%S'  
log_rotation_age = 1d  
log_rotation_size = 1GB  
log_min_duration_statement = 250ms  
log_checkpoints = on  
log_connections = on  
log_disconnections = on  
log_lock_waits = on  
log_temp_files = 0
```

# Memory configuration

- `shared_buffers`
- `work_mem`
- `maintenance_work_mem`

# shared\_buffers

- Below 2GB (?), set to 20% of total system memory.
- Below 32GB, set to 25% of total system memory.
- Above 32GB (lucky you!), set to 8GB.
- Done.



# work\_mem

- Start low: 32-64MB.
- Look for 'temporary file' lines in logs.
- Set to 2-3x the largest temp file you see.
- Can cause a **huge** speed-up if set properly!
- But be careful: It can use that amount of memory per planner node.

# maintenance\_work\_mem

- 10% of system memory, up to 1GB.
- Maybe even higher if you are having VACUUM problems.
- (We'll talk about VACUUM later.)

# effective\_cache\_size

- Set to the amount of file system cache available.
- If you don't know, set it to 50% of total system memory.
- And you're done with memory settings.

# Checkpoints.

- A complete flush of dirty buffers to disk.
- Potentially a lot of I/O.
- Done when the first of two thresholds are hit:
  - A particular number of WAL segments have been written.
  - A timeout occurs.

# Checkpoint settings.

`wal_buffers = 16MB`

`checkpoint_completion_target = 0.9`

`checkpoint_timeout = 10m-30m # Depends on restart time`

`checkpoint_segments = 32 # To start.`

# Checkpoint settings, 2.

- Look for checkpoint entries in the logs.
- Happening more often than `checkpoint_timeout`?
- Adjust `checkpoint_segments` so that checkpoints happen due to timeouts rather than filling segments.
- And you're done with checkpoint settings.

# Checkpoint settings notes.

- The WAL can take up to  $3 \times 16\text{MB} \times \text{checkpoint\_segments}$  on disk.
- Restarting PostgreSQL *from a crash* can take up to `checkpoint_timeout` (but usually less).

# Planner settings.

- `effective_io_concurrency` — Set to the number of I/O channels; otherwise, ignore it.
- `random_page_cost` — 3.0 for a typical RAID10 array, 2.0 for a SAN, 1.1 for Amazon EBS.
- And you're done with planner settings.



# Do not touch.

- `fsync = on`
  - Never change this.
- `synchronous_commit = on`
  - Change this, but only if you understand the data loss potential.

# Changing settings.

- Most settings just require a server reload to take effect.
- Some require a full server restart (such as `shared_buffers`).
- Many can be set on a per-session basis!

# pg\_hba.conf

# Users and roles.

- A “role” is a database object that can own other objects (tables, etc.), and that has privileges (able to write to a table).
- A “user” is just a role that can log into the system; otherwise, they’re synonyms.
- PostgreSQL’s security system is based around users.

# Basic user management.

- Don't use the "postgres" superuser for anything application-related.
- Sadly, you probably will have to grant schema-modifications privileges to your application user, if you use migrations.
- If you don't have to, don't.

# User security.

---

- By default, database traffic is not encrypted.
- Turn on ssl if you are running in a cloud provider.
- Have you responded to the Heartbleed OpenSSL bug, btw?

# The WAL.

# Why are we talking about this now?

- The Write-Ahead Log is key to many PostgreSQL operations.
- Replication, crash recovery, etc., etc.
- Don't worry (too much!) about the internals.



# The Basics.

---

- When each transaction is committed, it is logged to the write-ahead log.
- The changes in that transaction are flushed to disk.
- If the system crashes, the WAL is “replayed” to bring the database to a consistent state.

# A continuous record of changes.

- The WAL is a continuous record of changes since the last checkpoint.
- Thus, if you have the disk image of the database, and every WAL record since that was created...
- ... you can recreate the database to the end of the WAL.

# pg\_xlog

- The WAL is stored in 16MB segments in the pg\_xlog directory.
- Don't mess with it! Never delete anything out of it!
- Records are automatically recycled when they are no longer required.

# WAL archiving.

- `archive_command`
- Runs a command each time a WAL segment is complete.
- This command can do whatever you want.
- What you want is to move the WAL segment to someplace safe...
  - ... on a different system.

# On a crash...

- When PostgreSQL restarts, it replays the WAL log to bring itself back to a consistent state.
- The WAL segments are essential to proper crash recovery.
- The longer since the last checkpoint, the more WAL it has to process.

# synchronous\_commit

- When “on”, COMMIT does not return until the WAL flush is *done*.
- When “off”, COMMIT returns when the WAL flush is *queued*.
- Thus, you might lose transactions on a crash.
- No danger of database corruption.

# Backup and Recovery

# pg\_dump

- Built-in dump/restore tool.
- Takes a logical snapshot of the database.
- Does not lock the database or prevent writes to disk.
- Low (but not zero) load on the database.



# pg\_restore

- Restores database from a pg\_dump.
- Is not a fast operation.
- Great for simple backups, not suitable for fast recovery from major failures.

# pg\_dump / pg\_restore advice

- Back up globals with `pg_dumpall --globals-only`.
- Back up each database with `pg_dump` using `--format=custom`.
- This allows for a parallel restore using `pg_restore`.

# pg\_restore

- Restore using `--jobs=<# of cores + 1>`.
- Most of the time in a restore is spent rebuilding indexes; this will parallelize that operation.
- Restores are not fast.

# PITR backup / recovery

- Remember the WAL?
- If you take a snapshot of the data directory...
- ... it won't be consistent, but if we add the WAL records...
- ... we can bring it back to consistency.

# Getting started with PITR.

- Decide where the WAL segments and the backups will live.
- Configure `archive_command` properly to do the copying.

# Creating a PITR backup.

- `SELECT pg_start_backup(...);`
- Copy the disk image and any WAL files that are created.
- `SELECT pg_stop_backup();`
- Make sure you have all the WAL segments.
- The disk image + WAL segments are your backup.

# WAL-E

- <http://github.com/wal-e/wal-e>
- Provides a full set of appropriate scripting.
- Automates create PITR backups into AWS S3.
- Highly recommended!

# PITR Restore

- Copy the disk image back to where you need it.
- Set up `recovery.conf` to point to where the WAL files are.
- Start up PostgreSQL, and let it recover.



# How long will this take?

- The more WAL files, the longer it will take.
- Generally takes 10-20% of the time it took to create the WAL files in the first place.
- More frequent snapshots = faster recovery time.

# “PITR”?

- Point-in-time recovery.
- You don't have to replay the entire WAL stream.
- It can be stopped at a particular timestamp, or transaction ID.
- Very handy for application-level problems!

# Replication.

- Hey, what if we sent the WAL directly to another server?
- We could have that server keep up to date with the primary server!
- And that's how PostgreSQL replication works.

# WAL Archiving.

- Each 16MB segment is sent to the secondary when complete.
- The secondary reads it, and applies it to its copy.
- Make sure the WAL file copied automatically.
  - Use rsync, WAL-E, etc., not scp.

# Hmm... but what if we...

- ... transmitted the WAL changes directly to the secondary without having to ship the file?
- Great idea!
- Such a great idea, PostgreSQL implements it!
- That's what Streaming Replication is.

# Streaming Replication Basics.

- The secondary connects via a standard PostgreSQL connection to the primary.
- As changes happen on the primary, they are sent down to the secondary.
- The secondary applies them to its local copy of the database.

# recovery.conf

- All replication is orchestrated through the `recovery.conf` file.
- Always lives in your `$PGDATA` directory.
- Controls how to connect to the primary, how far to recover (for PITR), etc., etc.
- Also used if you are bringing the server up as a PITR recovery instead of replication.

# Disaster recovery.

- Always have a disaster recovery strategy.
- What if you data center / AWS region goes down?
- Have a plan for recovery from a remote site.
- WAL archiving is a great way to handle this.



# pg\_basebackup

- Utility for doing a snapshot of a running server.
- Easiest way to take a snapshot to start a new secondary.
- Can also be used as an archival backup.

**Let's see!**

# Replication!

# Replication, the good.

- Easy to set up.
- Schema changes are automatically replicated.
- Secondary can be used to handle read-only queries for load balancing.
- Very few gotchas; it either works or it doesn't, and it is vocal about not working.

# Replication, the bad.

- Entire database or none of it.
- No writes of any kind to the secondary.
  - This includes temporary tables.
- Some things aren't replicated.
  - Temporary tables, unlogged tables.

# Advice?

- Start with WAL-E.
  - The README tells you everything you need to know.
- Handles a very large number of complex replication problems easily.
- As you scale out of it, you'll have the relevant experience.

# Trigger-based replication

- Installs triggers on tables on master.
- A daemon process picks up the changes and applies them to the secondaries.
- Third-party add-ons to PostgreSQL.

# Trigger-based rep: Good.

- Highly configurable.
- Can push part or all of the tables; don't have to replicate everything.
- Multi-master setups possible (Bucardo).



# Trigger-based rep: The bad.

- Fiddly and complex to set up.
- Schema changes must be pushed out manually.
- Imposes overhead on the master.

# Transactions, MVCC and VACUUM

# “Transaction”

- A unit of which which must be:
  - Applied atomically to the database.
  - Invisible to other database clients until it is committed.

# The Classic Example.

```
BEGIN;  
INSERT INTO transactions(account_id, value, offset_id)  
    VALUES (11, 120.00, 14);  
INSERT INTO transactions(account_id, value, offset_id)  
    VALUES (14, -120.00, 11);  
COMMIT;
```

# Transaction Properties.

- Once the COMMIT completes, the data has been written to permanent storage.
- If a database crash occurs, any transactions will be COMMITed or not; no half-done transactions.
- No transaction can (directly) see another transaction in progress.

# In PostgreSQL...

- Everything runs inside of a transaction.
- If no explicit transaction, each statement is wrapped in one for you.
- This has certain consequences for database-modifying functions.
- Everything that modifies the database is transactional, even schema changes.

# A brief warning...

- Many resources are held until the end of a transaction.
- Temporary tables, working memory, locks, etc.
- Keep transactions brief and to the point.
- Be aware of IDLE IN TRANSACTION sessions.

# Transaction would be easy...

- ... if databases were single user.
- They're not.
  - Thank goodness.
- So, how do we handle concurrency control when two sessions are trying to use the same data?



# The Problem.

- Process 1 begins a transaction.
- Process 2 begins a transaction.
- Process 1 updates a tuple.
- Process 2 reads that tuple.
- What happens?

# Bad Things.

- Process 2 can't get the new version of the tuple (ACID [generally] prohibits dirty reads).
- But where does it get the old version of the tuple from?
  - Memory? Disk? Special roll-back area?
  - What if we touch 250,000,000 rows?

# Some Approaches.

---

- Lock the whole database.
- Lock the whole table.
- Lock that particular tuple.
- Reconstruct the old state from a rollback area.
- None of these are particularly satisfactory.

# Multi-Version Concurrency Control.

- Create multiple “versions” of the database.
- Each transaction sees its own “version.”
  - We call these “snapshots” in PostgreSQL.
- Each snapshot is a first-class member of the database.
  - There is no privileged “real” snapshot.

# The Implications.

- Readers do not block readers.
- Readers do not block writers.
- Writers do not block readers.
- Writers only block writers to the same tuple.

# Snapshots.

- Each transaction maintains its own snapshot of the database.
- This snapshot is created when a statement or transaction starts (depending on the transaction isolation mode).
- The client only sees the changes in its own snapshot.

# Nothing's Perfect.

- PostgreSQL will not allow two snapshots to “fork” the database.
- If this happens, it resolves the conflict with locking or with an error, depending on the isolation mode.
- Example: Two separate clients attempt to update the same tuple.

# Isolation Modes.

- PostgreSQL supports:
  - READ COMMITTED — The default.
  - REPEATABLE READ
  - SERIALIZABLE
- It does not support:
  - READ UNCOMMITTED (“dirty read”)



# What is a “snapshot”?

- Logically, it is the set of all transactions that have committed at a particular point in time.
- You can even manipulate snapshots (save them, load them).
- Snapshots are integral to how MVCC works in PostgreSQL.

# When does a snapshot begin?

- In `READ COMMITTED`, each statement starts its own snapshot.
- Thus, it sees anything that has committed since the last statement.
- If it attempts to update a tuple another transaction has touched, it blocks until that transaction commits.

# Higher isolation modes.

- REPEATABLE READ and SERIALIZABLE take the snapshot when the transaction begins.
- Snapshot lasts until the end.
- An attempt to modify a tuple another transaction has changed blocks...
- ... and returns an error if that transaction commits.

# Wait, what?

- PostgreSQL attempts to maintain an illusion of a perfect snapshot.
- But if it can't, it throws an error.
- The application then can retry the transaction against the new, updated snapshot.

# SERIALIZABLE

- Not every “conflict” can be detected at the single tuple-level.
- INSERTing calculated values.
- SERIALIZABLE detects these using predicate locking.
- Requires some extra overhead, but remarkably efficient.

# MVCC consequences.

- Deleted tuples are not (usually) immediately freed.
- Tuples on disk might not be available to be readily checked.
- This results in dead tuples in the database.
- Which means: **VACUUM!**

# VACUUM

- VACUUM's primary job is to scavenge tuples that are no longer visible to any transaction.
- They are returned to the free space for reuse.
- autovacuum generally handles this problem for you without intervention.

# ANALYZE

- The planner requires statistics on each table to make good guesses for how to execute queries.
- ANALYZE collects these statistics.
- Done as part of VACUUM.
- Always do it after major database changes — especially a restore from a backup.



# “Vacuum’s not working.”

- It probably is.
- The database generally stabilize at 20% to 50% bloat. That’s acceptable.
- If you see autovacuum workers running, that’s generally not a problem.

# “No, really, VACUUMs not working!”

- Long-running transactions, or “idle-in-transaction” sessions?
- Manual table locking?
- Very high write-rate tables?
- Many, many tables (10,000+)?

# Unclogging the VACUUM.

- Reduce the autovacuum sleep time.
- Increase the number of autovacuum workers.
- Do low period manual VACUUMs.
- Fix IIT sessions, long transactions, manual locking.

# Schema Design.

# What's “Normal”?

- Normalization is important.
- But don't obsess.
- It flows naturally from proper separation of data.

# Pick “Entities.”

- An entity is the top-level logical object in your data model.
- Customer, Order, InventoryItem.
- Flow down from there to subsidiary items.
- Make sure that no entity-level information gets pushed into the subsidiary items.

# Pick a naming scheme and stick with it.

- Are tables plural or singular?
  - DB people tend to like plural, ORMs tend to like singular.
- Are field names CamelCase, lower\_case, or what?

# Don't Repeat Yourself.

---

- “Denormalization” generally means including data that could be derived from other sources.
  - Copied.
  - Calculated.
- Calculated denormalization can sometimes be useful; copied almost never.



# Joins are Good.

- PostgreSQL executes joins very efficiently.
- Don't be afraid of them.
- Especially don't worry about large tables joining small tables.
- PostgreSQL will almost always do the right thing.

# Use the Typing System.

- PostgreSQL has a very rich set of types.
- Use them!
- If something's a numeric, don't store it as a string.
- Use domains to create custom types.

# No Polymorphic Fields.

- Avoid fields whose interpretation is dependent on another field.
- Avoid fields which use strings to store multiple types.
- Keep each field well-defined as to what data goes into it.

# Constraints.

- Use them. They're cheap and fast.
- Constraints on single columns.
- Constraints on multiple columns.
- Exclusion constraints for constraints across multiple rows.

# Pick a naming scheme and stick with it.

- Are tables plural or singular?
  - DB people tend to like plural, ORMs tend to like singular.
- Are field names CamelCase, lower\_case, or what?

# Avoid Entity-Attribute-Value Schemas.

- Each field should mean one thing, and one thing only.
- EAV schemas are nightmares to join and report on.
- They can also result in enormous database bloat.

# Key Selection.

- SERIAL is convenient and straight-forward, but...
  - What if you have to merge two tables?
- Use natural keys in preference to synthetic keys if you possibly can.
- Consider UUIDs instead of serials as synthetic keys.

# Don't Have “Thing” Tables.

- OO programmers sometimes like to have table hierarchies.
- These tend to result in big base tables that have common attributes factored out.
- It looks normalized...
  - ... but it's really a pain in the neck.



# Fast / Slow

- If a table has a frequently-updated section and a slowly-updated section, consider splitting the table.
- Do a 1:1 relationship between the two.
- Keeps foreign key locking under control.

# Arrays.

- First-class type in PostgreSQL.
- Can be searched, indexed, etc.
- Often a good substitute to a subsidiary table.
- Often a great substitute to a big many-to-many table.

# hstore

- Much, much better than an EAV schema.
- Great for optional, variable attributes.
- Can be indexed, searched, etc.
- But don't use it as a replacement for schema modification!

# JSON

- First-class, in-core type.
- Not quite as many search / indexing operator as hstore...
  - But it's getting there.
- Coming in 9.4: jsonb!

# Indexing on Big Types.

- PostgreSQL makes it work.
- But it can be very inefficient.
- Consider indexing on an expression of the data:
  - Like the first 32 / last 16 characters of a text string.

# NULL

- NULL is a total pain in the neck.
- Sometimes, you have to deal with NULL, but:
- Only use it to mean “missing value.”
- Never, ever have it as a meaningful value in a key field.
- WHERE NOT IN (SELECT ...)

# Very Large Objects

- Let's say 1MB or more.
- Store them in files, store metadata in the database.
- The database API is not designed for passing large objects around.

# Many-to-Many Tables

- These can get extremely large.
- Consider replacing with array fields.
  - Either one way, or both directions.
- Can use a trigger to maintain integrity.
- Much smaller and more efficient.
- Depends, of course, on usage model.



# Character Encoding.

- Use UTF-8.
- Just. Do. It.
- There is no compelling reason to use any other character encoding.
- One edge case: the bottleneck is sorting text strings. This is very, very rare.

# Time Representation.

- Always use `TIMESTAMPTZ`.
  - `TIMESTAMP` is a bad idea.
- `TIMESTAMPTZ` is “timestamp, converted to UTC.”
- `TIMESTAMP` is “timestamp, at some time zone but we don’t know which one, hope you do.”

# Indexing

# Test your database knowledge!

What does the SQL standard require for indexes?

# Trick Question!

# It doesn't.

- The database should work identically whether or not you have indexes.
- Of course, “identically” in this case does not mean “just as fast.”
- No real-life database can work properly without indexes.

# PostgreSQL Index Types.

- B-Tree.
- Hash.
- GiST.
- ~~SP-GiST.~~
- GIN.

# B-Tree Indexes.

- The standard PostgreSQL index is a B-tree.
- Provides  $O(\log N)$  access to leaf nodes.
- Provides total ordering.
- Operates on scalar values that implement standard comparison operators.



# B-Tree Index Types.

- Single column.
- Multiple column (composite).
- Expression (“functional”) indexes.

# Single Column B-Trees

- The simplest index type.
- Can be used to optimize searches on  $<$ ,  $<=$ ,  $=$ ,  $>=$ ,  $>$ .
- Can be used to retrieve rows in sorted order on that column.

# When to create?

- If a query uses that column, and...
  - ... uses one of the comparison operators.
  - ... and selects <10-15% of the rows.
  - ... and is run frequently.
- ... the index will likely be helpful.

# Indexes and JOINS

- Indexes can accelerate JOINS considerably.
- But the usual rules apply.
- Generally, they help the most when indexing the key on the larger table and...
- ... that results in high selectivity against the smaller table.

# Indexes and Aggregates.

- Some GROUP BY and related operations can benefit from an index.
- Often only in the presence of a HAVING clause, though.
- If it has to scan the whole index, it might as well scan the whole table.

# Mandatory indexes.

---

- Constraints must have indexes to enforce them.
- Just accept those.

# Ascending vs Descending?

- By default, B-trees index in ascending order.
- Descending indexes are much faster in retrieving tuples in descending order.
- So, if the primary function is descending sortation, use that.
- Otherwise, just use ascending order.

# Composite Indexes.

- A single index can have multiple columns.
- The columns must be used left-to-right.
- An index on (A, B, C) does not help a query on just C.
- But it does on (A, B).



# Expression Indexes.

- Indexes on an expression.
- PostgreSQL can recognize when you are querying on that expression and use the index.
- Can be expensive to create, but very fast to execute.
- Make sure PostgreSQL is really using it!

# Partial Indexes.

- An index does not have to contain all of the rows of the table.
- The `WHEN` clause's boolean predicate limits the size of the index.
- This can be a huge performance improvement for queries that match the predicate, all or in part.

# Indexes and MVCC

- The full key value is copied into the index.
- Every version of the tuple on the disk appears in the index.
- Thus, PostgreSQL needs to check whether a retrieved tuple is live.
- This means indexes can bloat as dead tuples pile up.

# GiST Indexes.

- GiST is not a single index type, but an index framework.
- It can be used to create B-tree-style indexes.
- It can also be used to create other index types, like bounding-box and geometric queries.

# GiST Index Usage.

- Non-total-ordered types generally require a GiST index.
- Each type's index implementation decides what operators to support.
  - Inclusion, membership, intersection...
- Some GiST indexes do provide ordering.
  - KNN indexes, for example.

- Generalized Inverted Index.
- Maps index items (words, dict keys) to rows whose field contains those.
- Core PostgreSQL use: Full text search indexes.
- Maps tokenized words to the rows containing those words.

# GIN implementation

- A B-tree of B-trees.
- Tokens organized into B-trees.
- Row pointers also organized into B-trees.
- On-disk footprint can be quite large.

# “Why isn’t it using my indexes?”

- The most common complaint.
- First, get the `EXPLAIN ANALYZE` output of the query.
- Sometimes, it is using the index, and it’s just slow anyway!



# Bad Selectivity.

- If PostgreSQL thinks that the index scan will return a large percentage of the table, it will do a seq scan instead.
- Generally, it's right to think this.
- If it's wrong, and the query is very selective, try re-running `ANALYZE`.

# ANALYZE didn't help.

- Try running the query with:
  - SET enable\_seqscan = 'off';
- See how long it takes to use the index then.
  - PostgreSQL might be right.
- Hey, it didn't use the index even then!

# Index Prohibitorum

- This means PostgreSQL thinks that index doesn't apply to this query.
- Query mis-written? Index invalid?  
Confusing expression index?
- Try doing a very simple query on just that field, and build up.

# PostgreSQL is right, but wrong.

- In fact, using the index is faster even though PostgreSQL thinks it is not.
- Try lowering `random_page_cost`.
- Consider changing the default statistics target for that field.

# PostgreSQL, Your Query Plan Sucks.

```
Bitmap Heap Scan on mytable (cost=12.04..1632.35 rows=425  
width=321)
```

```
  Recheck Cond: (p_id = 543094)
```

```
    -> Bitmap Index Scan on idx_mytable_p_id  
(cost=0.00..11.93 rows=425 width=0)
```

```
      Index Cond: (p_id = 543094)
```

# What does this mean?

- First, PostgreSQL scans the index and builds a bitmap of pages (not tuples!) that contain candidate results.
- Then, it scans the heap (the actual database), retrieving those pages.
- And then rechecks the condition against the tuples on that page.

# That makes no sense whatsoever.

---

- PostgreSQL does this when the number of tuples to be retrieved is large.
- It can avoid doing lots of random access to the disk.

# Pure Index Scan.

```
Index Scan using testi on test (cost=0.00..8.27 rows=1  
width=4)  
  Index Cond: (whatever = 5)  
(2 rows)
```



# Index Creation.

- Two ways of creating an index:
  - CREATE INDEX
  - CREATE INDEX CONCURRENTLY

# CREATE INDEX

- Does a single scan of the table, building the index.
- Uses `maintenance_work_mem` to do the creation.
- Keeps an exclusive lock on the table while the index build is going on.

# CREATE INDEX CONCURRENTLY

- Does two passes over the table:
  - Builds the index.
  - Validates the index.
- If the validation fails, the index is marked as invalid and won't be used.
- Drop it, run again.

# REINDEX

- Rebuilds an existing index from scratch.
- Takes an exclusive lock on the table.
- Generally no need to do this unless an index has gotten badly bloated.

# Index Bloat.

- Over time, B-tree indexes can become bloated.
- Sparse deletions from within the index range are the usual cause.
- <http://pgsql.tapoueh.org/site/html/news/20080131.bloat.html>
- Generally, don't worry about it.

# Index Usage.

- `pg_stat_user_indexes`
- Reports the number of times an index is used.
- If non-constraint indexes are not being used, drop them.
- Indexes are very expensive to maintain.

# Debugging

# “This query is slow.”

- EXPLAIN or EXPLAIN ANALYZE
- The output is... somewhat cryptic.
- Let's look at an example from the bottom up.
- <http://explain.depesz.com/>



```
select COUNT(DISTINCT "ecommerce_order"."id") FROM
"ecommerce_order" LEFT OUTER JOIN "ecommerce_solditem" ON
("ecommerce_order"."id" = "ecommerce_solditem"."order_id") WHERE
("ecommerce_order"."subscriber_id" = 396760 AND
("ecommerce_solditem"."status" = 1 AND
("ecommerce_solditem"."user_access_denied" IS NULL OR
"ecommerce_solditem"."user_access_denied" = false ) AND
"ecommerce_order"."status" IN (3,9,12,16,14)));
```

-----  
Aggregate (cost=2550.42..2550.43 rows=1 width=4)  
-> Nested Loop (cost=0.00..2550.41 rows=3 width=4)  
-> Index Scan using ecommerce\_order\_subscriber\_id  
on ecommerce\_order (cost=0.00..132.88 rows=16 width=4)  
Index Cond: (subscriber\_id = 396760)  
Filter: (status = ANY ('{3,9,12,16,14}'::integer[]))  
-> Index Scan using ecommerce\_solditem\_order\_id  
on ecommerce\_solditem (cost=0.00..150.86  
rows=19 width=4)  
Index Cond: (ecommerce\_solditem.order\_id =  
ecommerce\_order.id)  
Filter: (((ecommerce\_solditem.user\_access\_denied  
IS NULL) OR  
(NOT ecommerce\_solditem.user\_access\_denied))  
AND (ecommerce\_solditem.status = 1))

# Query Analysis.

- Read the execution plan from the bottom up.
- Look for nodes that are processing a lot of data...
- ... especially if the data set is being reduced considerably on the way up.

# ANALYZE

- The planner requires good statistics to create these plans.
- ANALYZE collects them.
- If the statistics are bad, the plans will be, too.

---

```
Aggregate (cost=48353.52..48353.53 rows=1 width=4)
-> Nested Loop (cost=0.00..48353.52 rows=1 width=4)
    -> Seq Scan on ecommerce_solditem
        (cost=0.00..38883.38 rows=868 width=4)
        Filter: (((user_access_denied IS NULL) OR
            (NOT user_access_denied)) AND (status = 1))
    -> Index Scan using ecommerce_order_pkey on
        ecommerce_order (cost=0.00..10.90 rows=1 width=4)
        Index Cond: (id = ecommerce_solditem.order_id)
        Filter: ((subscriber_id = 396760) AND
            (status = ANY ('{3,9,12,16,14}'::integer[])))
```

-----  
Aggregate (cost=2550.42..2550.43 rows=1 width=4)  
-> Nested Loop (cost=0.00..2550.41 rows=3 width=4)  
-> Index Scan using ecommerce\_order\_subscriber\_id  
on ecommerce\_order (cost=0.00..132.88 rows=16 width=4)  
Index Cond: (subscriber\_id = 396760)  
Filter: (status = ANY ('{3,9,12,16,14}'::integer[]))  
-> Index Scan using ecommerce\_solditem\_order\_id  
on ecommerce\_solditem (cost=0.00..150.86  
rows=19 width=4)  
Index Cond: (ecommerce\_solditem.order\_id =  
ecommerce\_order.id)  
Filter: (((ecommerce\_solditem.user\_access\_denied  
IS NULL) OR  
(NOT ecommerce\_solditem.user\_access\_denied))  
AND (ecommerce\_solditem.status = 1))

# Planner Statistics

- Collected as histograms on a per-column basis.
- 100 buckets by default.
- Not restored from backup!
- Not automatically updated on major database updates!

# Cost.

- Measured in arbitrary units (traditionally have been “disk fetches”).
- First number is the startup cost for the first tuple, second is the total cost.
- Comparable with other plans using the same planner configuration parameters.
- Costs are inclusive of subnodes.



# Actual Time.

- In milliseconds.
- Wall-clock time, not only query execution time.
- Also presents startup time, total time.
- Also inclusive of subnodes.

# Rows.

---

- Estimated and actual rows emitted by each planner node.
- Not the number processed; that could be larger, and is reflected in cost.
- A large mismatch is one of the first places to look for query problems.

# Loops.

- Number of times a subplan was executed by its parent.
- In this case, actual times are averages, not totals.

# Things that are bad.

- JOINS between two very large tables.
  - Very difficult to execute efficiently unless the sides can be reduced by a predicate.
- CROSS JOINS
  - These can be created by accident!
- Sequential scans on large tables.

# SELECT COUNT(\*)

- Always results in a full table scan in PostgreSQL.
- So don't do that.

# OFFSET / LIMIT

- Everyone's favorite way of implementing pagination.
- OK for low OFFSET values...
  - ... but comes apart fast for higher ones.
  - GoogleBot Is Relentless.
- Precalculate, use other keys.

# “The database is slow.”

- What’s going on?
- `pg_stat_activity`
- `tail -f` the logs.
- Too much I/O? `iostat 5`

# “The database isn’t responding.”

- Make sure it’s up!
- Can you connect with psql?
- pg\_stat\_activity
- pg\_locks



# Python Particulars

# Python 2? psycopg2

- Overall, the best library for accessing PostgreSQL directly from Python.
- Hard to justify using anything else.
- Very feature-rich, very Pythonic (such as DB API 2 is Pythonic).

# psycopg2 notes.

- The result set of a query is loaded into client memory when the query completes...
- ... regardless of the size of the result set!
- If you want to scroll through the results, use *named* cursors.

# py-postgresql

---

- Python 3.x driver.
- Pure Python, so can run under interpreters that require it.

# Django Notes.

- If you are running on 1.6+, always use the `@atomic` decorator.
- Cluster write operations into small transactions, leave read operations outside.
- Do all your writes at the very end of the view function.

# Django + Replication

- Multi-database works very nicely with hot standby.
- Point the writes at the primary, the reads at the secondary.

# Django 1.5 or earlier.

- Use the `@xact` decorator and style.
- <https://github.com/xof/xact>
- Sloppy transaction management can cause the dreaded Django idle-in-transaction problem.

# Go South.

- Use South in Django for migration management.
- Create manual migrations for schema changes that Django can't specify.
- Specialized constraints, indexes, etc.



# Special Situations.

# Minor version upgrade.

- Do this promptly!
- Only requires installing new binaries.
- If using packages, often as easy as just an `apt-get / yum upgrade`.
- Very small amount of downtime.

# Major version upgrade.

- Requires a bit more planning.
- `pg_upgrade` is now reliable.
- Trigger-based replication is another option for zero downtime.
- A full `pg_dump` / `pg_restore` is always safest, if practical.
- Always read the release notes!

# Don't get caught!

- Major versions are EOLd after 5 years.
- Always have a plan for how you are going to move between major versions.
- All parts of a replication set must be upgraded at once (for major versions).

# Bulk loading data.

- Use COPY, not INSERT.
- psycopg2 has a very nice COPY interface.
- COPY does full integrity checking and trigger processing.
- Do a VACUUM afterwards.

# Very high insert rates.

- Reduce shared buffers by 25%-75%.
- Reduce checkpoint timeouts to 3min or less.
- Make sure to do enough ANALYZEs to keep the statistics up to date, manual if required.

- Generally, works like any other system.
- Remember that instances can disappear and come back up without instance storage.
- Always have a good backup / replication implementation on AWS!
- PIOPS are useful (but pricey) if you are using EBS.

# Larger-Scale AWS Deployments

- Script everything: Instance creation, PostgreSQL setup, etc.
- Put everything inside a VPC.
- Scale up and down as required to meet load.
- AWS is a very expensive equipment rental service.



# PostgreSQL RDS

- Overall, not a bad product.
- BIG plus: Automatic failover.
- BIG minus: No reading from the secondary.
- Other minuses: Expensive, fixed (although large) set of extensions.
- Not a bad place to start with PostgreSQL.

# Sharding.

---

- Eventually, you will run out of write capacity on your master.
- Then what?
- Community PostgreSQL doesn't have an integrated multi-master solution.
- But there are options!

# Postgres-XC

- Open-source fork of PostgreSQL.
- Intended for dedicated hardware in a single rack.
- Node failure is still a challenge.
- Somewhat experimental, but shows great promise.

# Bucardo

- Has multi-master write capability.
- Handles burst-writes effectively.
- Not great for sustained writes, since the writes ultimately have to end up on all machines.

# Custom Sharding.

- Distribute data across multiple machines in a way that the application can find it.
- Can shard on an arbitrary value (user ID), or something less abstract (region).
- Application is responsible for routing to the right database node.
- <http://instagram-engineering.tumblr.com/post/10853187575/sharding-ids-at-instagram>

# Pooling, etc.

# Why pooling?

- Opening a connection to PostgreSQL is expensive.
- It can easily be longer than the actual query time.
- Above 200-300 connections, use a pooler.

# pgbouncer

- Developed by Skype.
- Easy to install.
- Very fast, can handle 1000s of connections.
- Does not to failover, load-balancing.
  - Use HAProxy or similar.



# pgPool II

- Does query analysis.
- Can route queries between master and secondary in replication pairs.
- Can do load balancing, failover, and secondary promotion.
- Higher overhead, more complex to configure.

# Tools

# Monitor, monitor, monitor.

- Use Nagios / Ganglia to monitor:
  - Disk space — at minimum.
  - CPU usage
  - Memory usage
  - Replication lag.
- `check_postgres.pl` ([bucardo.org](http://bucardo.org))

# Graphical clients

- pgAdmin III
  - Comprehensive, open-source.
- Navicat
  - Commercial product, not PostgreSQL-specific.

# Log Analysis

- `pgbadger`
  - The only choice now for monitoring text logs.
- `pg_stat_statements`
  - Maintains a buffer of data on statements executed, within PostgreSQL.

# Questions?

[thebuild.com](http://thebuild.com) / [@xof](https://twitter.com/xof) / [pgexperts.com](http://pgexperts.com)

# Thank you!

<http://tinyurl.com/pycon2014survey>

[thebuild.com](http://thebuild.com) / [@xof](https://twitter.com/xof) / [pgexperts.com](http://pgexperts.com)