10 Ways to Kill Performance.

Christophe Pettus
PostgreSQL Experts, Inc.

PgDay SCALE 9x
25 February 2011
Greetings.

- Hi, I’m Christophe.
- PostgreSQL user/developer since 1997
- 7.2 instance still running!
- Consultant with PostgreSQL Experts, Inc.
- [http://pgexperts.com/](http://pgexperts.com/)
This Talk Brought to You By…

- PostgreSQL Experts, Inc.
- DayQuil®
Dualist Heresies in the Byzantine Empire, 1130-1195AD.
Why Break Performance?

- Passive-Aggressive Response to Employer.
- General Rage Against the World.
- Fixing a Customer Complaint in Bizarro World. (“DB run too fast!”)
- Or, You Actually Want to Improve Performance.
Method 1: Entity-Attribute-Value

Schemas are for suckers.
EAV?

- A single table for highly heterogenous values.
- Generally has a foreign key ("entity"), a column for the entity’s semantics ("attribute"), and a column for the value.
<table>
<thead>
<tr>
<th>ENTITY</th>
<th>ATTRIBUTE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>NAME</td>
<td>Herman Munster</td>
</tr>
<tr>
<td>12</td>
<td>ADDRESS-1</td>
<td>1313 MOCKINGBIRD LANE</td>
</tr>
</tbody>
</table>
How does that kill performance?

• Monstrous to join over.

• Makes it hard (if not impossible) to enforce consistency at the database level.

• Everything’s a string!

• Increases the number of tuples (and thus database overhead).
Then why do it?

- Frequently found in ports from old-skool databases.
- Handy for user-defined values in a packaged application.
- PostgreSQL does have DDL. You might check it out.
Method 2: Little Teeny Tuples

Why “best practice” isn’t always.
Denormalization is Bad, Right?

- Right. Never do it.
- Never?
- No, never!
- NEVER?
- Well, hardly ever.
Let’s Take an Example

• Primary table: 1.2 billion rows.
• Each row has a variable number of "attributes."
• Each attribute is a boolean (present/absent).
• 3-12 attributes per primary row.
CREATE TABLE secondary (  
  primary_row BIGINT REFERENCES primary(pk),  
  attribute INTEGER REFERENCES attributes(pk)  
) PRIMARY KEY (primary_row, attribute);
Why does that kill performance?

• Suddenly, we have a new table with 18+ billion rows.
• Have fun joining over that sucker.
• Each row has a significant overhead.
• And then... indexes!
So, what should we do?

- Depends on the access model.
- What’s the selectivity of different attributes?
- intarray
- bit vector with indexes
Method 3: work_mem

A consultant’s retirement plan in a single setting!
work_mem

- How much memory to assign to a hash / sort / group operation.
- Per planner node, not per query or per session.
- Out of the box, it’s one megabyte.
A typical client interaction

• “This query is really really slow.”
• “What does it do?”
• “It takes 125 million rows and groups them down to 8.”
• “Let me see your postgresql.conf”
• “Hmm. I think I can help here.”
Next Stop, Cabo!
How does this kill performance?

- Spills big hash / sort / group operations to disk.
- Disks are slow. (You may have heard this.)
- 1MB is usually too low.
What to do?

• Bump it up!
• It’s a hard value to get right.
• Too small, performance issues.
• Too high, out of memory problems.
• Monitor, monitor, monitor.
• EXPLAIN ANALYZE is your friend.
Method 4: Mix ‘n’ Match Info

Don’t join! You’ll kill yourself!
**Base vs Derived Information.**

- **Base information** are facts about the row that rarely change once created.
  - Name, date of birth, gender.
- **Derived information** is dynamic and changes frequently.
  - Last ordered, incarceration status.
Slam it into one table!

- Everyone will need to write to the same row, all the time.
- Think of the fun you’ll have debugging locking conflicts!
- It’s even more exciting if multiple applications have different sets of derived information.
How does this kill performance?

- Deadlock, deadlock, deadlock.
- Single-file through the record.
- Different applications need to know each other’s access patterns.
So, what do to?

- Separate derived information into a separate table.
- 1:1 relationship, so joining is efficient.
- Different applications are isolated, so fewer conflicts.
Method 5: Poll the Database

“Got anything for me? How about now? Huh, huh, c’mon, you must have something for me now…”
Databases are great!

- Simple API.
- Consistency.
- Crash recovery.
- Concurrency controls.

Let’s use them FOR EVERYTHING IN THE ENTIRE WORLD EVAR!
Like, Say, Task Queues!

- Producer inserts a task into a task queue table.
- Consumers poll the database looking for new work.
- Profit, right?
Wrong.

• High rates of polling crush the database.
• Low rates of polling make inefficient use of the consumers.
• It’s actually quite hard to get the guarantees right.
What do to?

• Use a dedicated task queuing product for task queuing.

• If you must use the database, use LISTEN / NOTIFY.

• Never, ever, ever poll the database on a high duty cycle.
Method 6: Long Transactions

“This transaction has been open since July 2, 2001.

We call it ‘Edward.’”
PostgreSQL Rocks Transactions.

- PostgreSQL has very light-weight transactions, compared to other high-powered databases.
- Remember the rollback buffer? Yeah, that was a lot of fun.
- But with great power comes great responsibility.
Don’t Do This.

- User selects a record in a GUI application.
- Opens it for editing, opening a transaction.
- Goes to lunch.
- Decides to move to Croatia.
- Transaction is still open five months later.
What’s the big deal?

- <IDLE IN TRANSACTION>
- Holds system resources.
- Blocks VACUUM.
- Heaven help you if the transaction is holding locks.
“I’d never do that!”

- You probably wouldn’t.
- But is your ORM, API library, or pooler on the same page?
- Django is notorious for this.
- Monitor, monitor, monitor.
Method 7: The Single Row

“One row to rule them all, one row to find them...”
We all have them.

- “Settings.”
- “Preferences.”
- “Control information.”
- “You know, that row. In the table. With the stuff.”
It’s all fun and games…

- Until someone holds a lock.
- And, suddenly, the database is single-threaded.
- Or deadlocks start appearing left and right.
“I’d Never Do That!”

- Yeah, right.
- Do you really know what transaction model you are using?
  - Really?
- Particularly bad with ORMs that attempt to “help” you with transactions.
So, what to do?

- Don’t hold a transaction open on singletons.
- Get in, say what you need to say, get out.
- Understand what transaction model your frameworks are giving you.
Method 8: Attack of the BLOB

“Magic Database Disk Access Powers, Activate!”
Clients Love Databases.

- Sometimes to death.
- “We want to store these images in a database.”
- “How big are they?”
- “Oh, 64MB apiece.”
- “Uh, why store them in the database?”
DATABASES ARE FAST!

- PostgreSQL doesn’t have a special red phone to the underlying disk.
- It’s not designed to handle very large objects, although it does a heroic job of it if you ask.
- There’s no magic.
So, what do to?

• Every OS has a database optimized for the manipulation of large binary objects.

• It’s called a “file system.”

• Know it, use it.

• To be fair, databases do offer some advantages… but superior disk I/O isn’t among them.
Method 9: Partitioning Disasters

Partitioning is the chemotherapy of databases.
Partitioning

- Using table inheritance to split a single table up into multiple children...
- ... on the basis of a partitioning key.
- It can do amazing things for performance...
IN THE RIGHT SITUATION.

- Data can be divided into roughly-equal sized “buckets” based on the partitioning key.
- Queries tend to land in a (very) small number of those buckets.
PARTITIONING KILLS!

- ... in the wrong circumstances.
- Queries span large number of partitions.
- Partitions of extremely unequal size.
- Confusion about the data model.
So, what do we do?

- Partitioning is great…
- … in the right situation.
- In the wrong one, it can make things much, much, MUCH worse.
- The final partition merge can be the death of a query.
Method 10:
Lots of Indexes

“If adding one index is good…”
Let’s index EVERYTHING!

• What can go wrong?

• After all, if it never uses an index, what’s the overhead?

• (pause)

• Oh. That’s the overhead, hm?
Good Indexes.

- High selectivity on common queries.
- Required to enforce constraints.
Bad Indexes.

- Pretty much everything else.
- Bad selectivity.
- Rarely used.
- Expensive to maintain compared to the query acceleration.
- FTS particularly vulnerable to this.
Stupid Indexing Tricks

- Multi-level indexes.
- Ordering is very important.
- Expensive functional indexes.
- Small variations that defeat index usage.
- Redundant indexes.
- PKs, text_pattern_ops
Bonus Method: Date/Time Functions!

Even a broken timezone is right twice a year.
Pop Quiz!

• What’s interesting about this calculation?

• SELECT '2011-03-13 02:00'::TIMESTAMPTZ + '1 hour'::INTERVAL;
2 + 1 = 4!

?column?

2011-03-13 04:00:00-07

(1 row)
This is absolutely correct.

- PostgreSQL is correctly handling the time offset change.
- There is an unfortunate side-effect, though.
- Calculations on TIMESTAMPTZs are VOLATILE.
This can be... surprising.

- Defeats queries on indexes.
- Defeats partition constraints.
- Hey, you could be doing a query at the exact moment a timezone shift happens!
- No, really, it could happen.
So, what do to?

- Precalculate TIMESTAMPTZs before doing queries on them.
- Understand what this means in terms of your query ranges.
- ... and be glad that PG isn’t Oracle.
Questions?

Sorry, I don’t actually know anything about dualist heresies in the Byzantine Empire. I’m sure they rocked.
Thanks.

cpettus@pgexperts.com
xof@thebuild.com