Srivathsava Rangarajan

2018/09/06

Disclaimer: Here there may be no dragons.

What is this talk not about?

- Innovation, development and advances in PostgreSQL and new features
- Brilliant depth first analysis of a single facet of PostgreSQL

What is this talk about?

- Research, compaction and simplified presentation of critical fundamentals of PostgreSQL gleaned from years of painful experience and many high-quality sources, online and in-real-life

Why are you giving this talk?

- I assume that I can't be alone in having had a non-traditional introduction to RDBMSes given the low barrier to entry for SQL
- Often people, such as I, reactively consume these fundamentals and end up knowing bits and pieces and everything but miss the beauty of the grand design

Why are *you* giving this talk?

- I, a millennial, have been labeled an official PostgreSQL groupie
- I am the lead software engineer for underwriting services at a billion dollar financial company
- My team owns 15+ services that use more than 6 types of persistence technologies
- My team owns more than a couple of terabytes of transactional PostgreSQL data

Confidential

Where is it you work again?

Enova: Chicago based FinTech Lending and Analytics/aa/Service

Sizeable PostgreSQL shop:

- 300+ production clusters, 500+ production databases
- > 10 databases of TB+ size

Great:

- People, value, leadership, opportunities

As is every company, we are hiring!

- If interested, please contact me: srangarajan@enova.com

Confidential

Dude, where's my byte?
a/k/a
The Millennial's Guide to PostgreSQL

enova.

So, I have this PostgreSQL database…and can't find my data.

Hmm, have you looked in PG_DATA?

What is PG_DATA?

- An environment variable denoting data directory during PostgreSQL cluster initialization.

What is a cluster?

- A cluster is a collection of databases.
- YMMV, but we usually run one database per cluster for our transactional systems

How do I make a cluster?

- `initdb -D /usr/local/var/postgresopen2018`
- If the PG_DATA environment variable were set, we wouldn't have to specify the directory

Wow, that's a lot of stuff…?

- Yep. A lot of is control information and defaults.
- The ones that we're interested in for the purpose of this talk:
  - `base:` contains per database system catalogs and user data (default)
  - `global:` contains cluster-wide data
  - `pg_tblspc:` contains filesystem symlinks to defined tablespaces

Confidential

All of it is my data?

Well, depends on what we define as data. If we care about just what we "see" as data – that is tables, rows and columns defined by us, `base` is a good place to start

- `ls /usr/local/var/postgresopen2018`

Hold on, I haven't even touched my cluster yet and there is stuff in there?

- Well, let's start up the cluster and see what's going on
- `pg_ctl -D /usr/local/var/postgresopen2018/ -l /usr/local/var/postgresopen2018/server.log start`

Well, so what do numbered subdirectories mean?

- oids of databases in `pg_database`
- `select oid, datname from pg_database;`

Confidential

Very nice. Now let's make a database?

```
create database postgresopen2018;
```

Hold on! What are these 290+ inside my database already?!

- Yeah, those are system catalogs

```
create tablespace special location
'/usr/local/var/postgresopen2018/special';
create table byte_test(test_byte "char") tablespace special;
```

Woah there. What is this tablespace thing?

- Tablespaces allow us define exactly where the data in our table will be saved
- This allows us to choose the right hardware for each set of tables

Great! So where exactly is my table now?

- `select pg_relation_filepath('byte_test');`

If we weren't using a custom tablespace, it would be at:

- `du -sh /usr/local/var/postgresopen2018/base/{db_oid}/{rel_filenode}`
- db_oid: `select oid, datname from pg_database;`
- rel_filenode: `select pg_relation_filenode('byte_test');`

Confidential

OK, enough setup, can we just make some data?

```
insert into byte_test values('1');
```

- Note that the file size is 8.0K

Wow, I thought I wrote 1 byte, but this is of size 8.0k. Nice overkill PostgreSQL?

- Actually, turns out the entire file is empty
- 8.0k is just the default "page" size that PostgreSQL has "allocated" and filled with nulls
- **Err, what is a "page"?**
  - TBD.

Confidential

Empty?! Dude, where's my byte?

Don't worry! It's safe!

Sure it is. If it's not here, then where is it?

- It's in the WAL - Write Ahead Log
- It will also be in the file really soon

Wait, what? It's in two places?

- That's right. One is a record of the command to insert the data and the other is the result of that action

Why?!

- Because it's safer this way

What if I shutdown my database before it shows up? Will it be lost?

- No

What if I just kill my database before it shows up? Will it be lost?

- That would be quite rude, but no

Confidential

OK, so what is this Write Ahead Log? Why do I need this?

Let us take the previous example – we want to simply insert a ‘1’, a single byte of data

Really how hard can this be?

- Open the file
- Seek to the right location – in our case, there is no data, we might already be at the right location
- Setup our data-structures
- Write the single byte of data
- Release the file handle
- ???
- Profit!!!

So it’s pretty simple right? Why not just do this?

- Sure, if everything works perfectly and nothing can ever go wrong
- What if we lose power as we’re writing our data-structure?
- What if we were writing more than a single byte of data and we lose disk connectivity half-way through it?
- What if everything went great, but our optimized SAN array has a write cache for rapid write which doesn’t get flushed out before it loses power?

Turns out we need a better mechanism than simply – “well, write it”. Thus, the WAL

Confidential

Write Ahead Log, I guess I need this?

Here is a safer sequence of operations to write the same byte of data

- Have a separate file, a log, that is append only with a strict format
- Append the disk operations on the data file as the result of running my insert to this log
- Ensure that this log is flushed to disk
- Apply change to the data file
- Mark operation as complete in the log file

So what is in this WAL is essentially a translation of a DML command to filesystem operations?

- Essentially, yeah. The specifics are of course monstrously more complicated

And this happens every time I run a command, standalone or inside a transaction?

- Yep
- Note that the “translation” itself may get buffered into a WAL buffer in-memory until one commits the transaction to avoid unnecessary I/O overhead

Confidential

So exactly how does this make things safer?

The WAL is a critical part of PostgreSQL's design to guarantee the **Atomicity** and **Durability** of ACID

- **Atomic:** The "translations" of commands to data file operations are appended to the WAL, but unless there is a "commit" entry (from transaction commit), they are ignored
- **Durable:** Any risk of data corruption is subverted by the presence of this structured master record, the WAL, thus guaranteeing the durability of our data

Are there any other advantages?

- Easy to detect, correct and/or discard corruption without furious seeking due to strict format
- Allows for batching and throttling disk data I/O, reducing overhead and improving latency

Where is this WAL file?

- `ls /usr/local/var/postgresopen2018/pg_wal/`
- Since this is the only synchronous file system operation that happens when we modify data, a simple hardware optimization here would be to mount this directory onto expensive hardware optimized for appends

How do I read this?

- `pg_controldata /usr/local/var/postgresopen2018/`
- Binaries are available in the community to decode these

Confidential

OK, this is great, but when do I really get my data?

That depends on three factors:

- `CHECKPOINTs, page_swap, background writer`
- Turns out our 5-point plan was a lie, the last 2 steps aren't performed immediately

Wait, what last 2 steps? You mean the ones where data is *actually* written to the file?!

- Yep. Turns out doing that synchronously on evert commit is a massive overhead

So then where exactly is my database reading my data from after updates?!

- `shared_buffers.` When a DML operation occurs, it is written to the WAL and the heap tuple's cached-copy in PostgreSQL's data page cache, `shared_buffers`, is updated and marked dirty
- Increasing this reduces I/O and improves latency. However, remember that the OS kernel is caching disk pages too and PostgreSQL relies on that heavily, so there's a tradeoff

So why are they called `shared_buffers`? I don't see any sharing…

- These buffers are shared between the server processes spun up by each client

Wait, what?! There is more than one PostgreSQL server process?

- Oh yeah. Everytime a client connects, the postmaster forks off a new process
- `select pg_backend_pid();`

Confidential

enova.

A process per client? Isn't that really expensive?

Oh yeah. This is why it is highly recommended that one install a client pooler ASAP. Not only does this reduce the memory footprint of your rig, it also improves latency and performance by having less chatter for synchronization

So shared_buffers are like a centralized cache over the data pages for all these connections?

- Yep. Can you imagine the complexity of managing a distributed data page cache across N-many backends?
- However, these processes do communicate with each other through shared memory (buffers, semaphores) to ensure consistency and integrity. More later

So if this shared_buffers are a cache, it is open to eviction at some point right? Then what will happen to my byte?

- Yep, and that's why `page_swaps` happen. A page_swap is basically a dirty, cached data page being sync'd back to the disk, thus causing the byte to be written to the data file

What if there is no eviction? What if my DB is idle? Will it stay in memory?

- Nope, it will find it's way to disk eventually using the `background writer & CHECKPOINTs`

Confidential

What is the background writer and CHECKPOINTs?

The `background writer` is a daemon that pre-emptively syncs dirty pages from the `shared buffers` to disk

Pre-emptively? I thought the point of buffering page writes was to prevent this steady overhead by batching?

- It is. But there are tradeoffs. Too much buffering will cause chokepoints, and so we amortize that

But what is the rate of amortization?

- It is determined by `checkpoint_completion_target` which is a measure of what percentage of the shared buffers PostgreSQL wants the daemon to have flushed by CHECKPOINT time

Wait, what are these CHECKPOINTs?

- Checkpoints are essentially markers in the WAL that note that at the time of occurrence, there is no dirty data in the PostgreSQL `shared buffers`
- To guarantee this, if there *is* any dirty data, the CHECKPOINT will cause a flush/sync of all that data to disk, making it a very I/O intensive operation
- Without CHECKPOINTs, to guarantee durability, we need to keep *all* our WAL around forever!

So when do these CHECKPOINTs happen?

- On a timer, and also when a certain amount of WAL churn has happened decided by a formula
- Tuning these checkpoint parameters is crucial to smooth I/O for you database
- They can also be triggered by hand and/or admin commands

Confidential

Hey, ps -ef | grep postgres makes a lot of sense now!

Indeed

- `ps -ef | grep postgres`

And so, finally, here is our data

- `cat /usr/local/var/postgresopen2018/{tblspace_oid}special/…/{db_oid}/{relation_oid}`

Confidential
enova.

Err, that's not my byte, it looks like a bunch of gibberish?

Well, it's a bunch of bytes in a structure representing our data. This is what it really looks like:

Confidential

enova

What are all these things?

tuple?

- Representation of row in the data file

ctid*?

- Static pointer to tuple as tuples tend to move around
- `select ctid, * from byte_test;`

heap?

- A collection of unordered tuples. A heap file is basically the data file. Note max size 1GB. If size of relation exceeds this, then PostgreSQL will make files with suffixes: …/`{oid}_1`, …/`{oid}_2`,…

page header?

- Metadata about page including checksums, WAL information

row header?

- transaction_ids, metadata about row, field sizes if variable length

field?

- Data in the column of a row

Confidential

OK great, so next I can simply update this data… right?

Of course we can! In fact, let's change it a hundred or so times!

- We have a simple pre-made list of UPDATE statements in a file
- `psql -f foo.sql postgresopen2018`

And now I expect that the final value of my byte is whatever the last value was, right?

- Yep!

And that's what is in my data file? Overwritten in place?

- Ooh. No, not really ☺
- As we can see, the data in the file has increased and the ctid is moving ahead!
- `select ctid, * from byte_test;`

…?!

- Yeah, turns out the database needs to account for more people than just us using it

What does that mean?

- Well, let's take an example

Confidential

enova.

Why isn't my data on disk being overwritten?

Let us open two database sessions and begin transactions on each

- `begin;`
- `select txid_current();`
- `select xmin, xmax, test_byte from byte_test;`

What is this xmin/xmax and txid stuff?

- `txid_current` gives us the id of the current tx - every tx gets one
- only transactions with `xmin < id < xmax` can see the row of data

Why do we need this?

- To guarantee an important property of ACID compliant databases:
  - **I**solation: R/W between transactions are isolated. The final state of my data as a result of all the commands in different transactions should be reachable by placing all these transactions in a queue and performing them one at a time without concurrency

So there is no concurrency in my database?

- You bet there is. Let's run this update on two different sessions and see how the xmin/xmax/test_byte values change:
- `begin; update byte_test set test_byte = 'a';`

Wait, what? Why do I see two different things?

- Because isolation is configurable. PostgreSQL has "level"s of isolation that we can choose from. Different isolation levels change what concurrent transactions see of each other's work

What is transaction isolation?

Is this why, one transaction believes that the byte is now ‘a’ and the other one believes that the byte is still ‘1’?

- Yep. Seeing is believing. Isolation controls what each tx “sees” of concurrent operations

So, which is true?

- They’re both true, in different timelines. That the byte was ‘1’ was true to begin with. That the byte is ‘a’ may become true if the transaction commits. Nevertheless to the session that set the byte to ‘a’, it is now true in the scope of that transaction

What if I make it absolutely true that the byte is now ‘a’ by committing the transaction?

- Then, based on the level of isolation, it will become true either immediately for the other transaction as well, or it may never become true
- `show transaction_isolation;`

What if I make it so that ‘a’ was never true by rolling back the transaction?

- Then we continue on our merry-way and pretend like nothing ever happened. Note that `xmax` remains set on the row

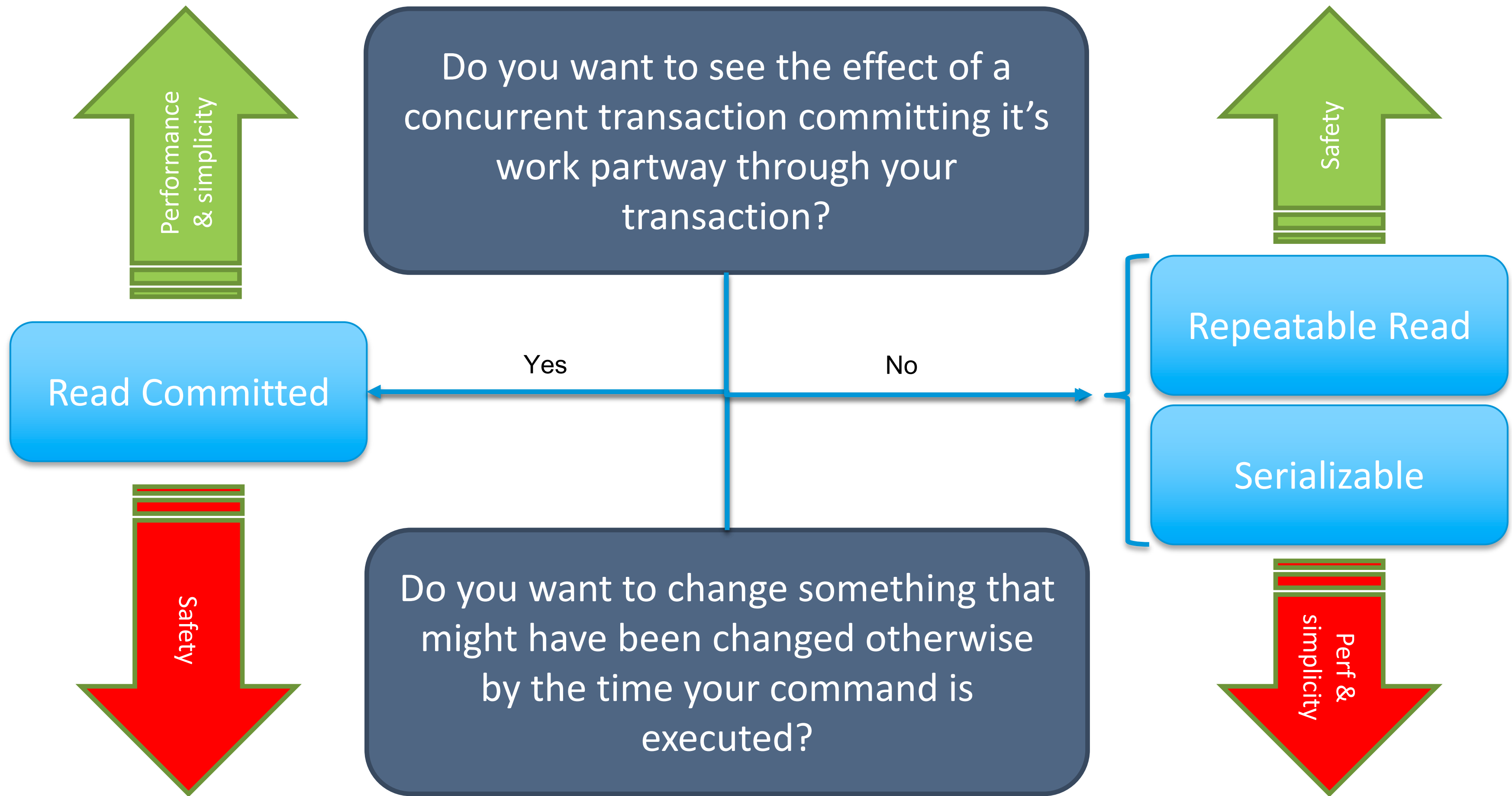
So I get to control what is “true” to my tx using isolation by trading off performance?

- Yep, by setting `transaction_isolation` one can actually choose what one’s transaction sees

 Confidential

What level of control over “visibility” between txs do I have?

Turns out, at least in PostgreSQL, it really just boils down to one question:



Remember that just because we choose to close our eyes to the rest of the world, doesn’t mean everything stops for us. So, listed here above is another question that can be asked

Confidential

enova

So, what does this have to do with overwriting my data?

Well, to offer this “truth” option, parallel state needs to be managed somehow.

- PostgreSQL manages this by never overwriting data in place. It instead uses an elegant, simple storage system called MVCC: Multi Version Concurrency Control to transform:
- Update: set `xmax` on old version and insert new version with `xmin`
- Delete: set `xmax` on row

Wait, what?! So my database will just keep on growing?

- Enter VACUUM. A process that cleans-up data files for a living, VACUUM is responsible for marking rows (tuple) “dead”. “Dead” here means that this tuple is now no longer visible/true to any active transaction and the space it occupies can be re-used
- There is an automatic background VACUUM process that does this for you

Re-used? Why not just you know…like delete it?

- Can’t do that without essentially rewriting the entire data file which is a very expensive operation
- Disk is cheap, might as well just mark it for re-use

What if I really want my space back?

- VACUUM FULL will rewrite the table and give you back the space
- Note: this operation will create a new filenode

Why MVCC? Why not queue up the transactions?

Yeah, that is definitely a legitimate option.

- Clearly having this isolation and MVCC makes this things confusing and causes unnecessary overhead in the form of "dead tuples" that need to get cleaned up later by yet another process
- But, queuing up the transactions causes a backlog and kills concurrency thus causing a latency overhead, an especially exacting sacrifice when one considers that this way we would:
  - Block transactions that are simply performing reads from happening in parallel with writes
  - Block transactions writing two unrelated sets of data from happening in parallel

Well, how about we line-up only the transactions that interfere?

- Ah, but how we would know that they *could* interfere until we see the entirety of all the transactions involved? And to want to see that is essentially equivalent to queuing them up

So, I have to choose between space and time?

- Precisely. And PostgreSQL chose to sacrifice a little space and complexity to gain time

And so MVCC is basically controlling the visibility of the effects of one transaction to another?

- MVCC is the storage mechanism that enables PostgreSQL to do so
- The thing that actually controls this visibility is the level of transaction isolation

Confidential

Does this mean PostgreSQL is fully parallel in execution?

Can it run any number of transactions containing any commands in them at the same time?

- No. Isolation and MVCC leave PostgreSQL free to execute only non-contending commands inside different transactions at the same time
- If there are contending commands, such as two updates on the same row, PostgreSQL will block one of the commands until the transaction containing the other complete
- And remember that execution of commands within a transaction is serial, and so essentially that command and the rest of the transaction after it is blocked behind the completion of the contending transaction
- This blocking happens using locks

So, isolation and MVCC enable some concurrency, but not unlimited concurrency?

- Yep. The design enables as much "safe" and "correct" concurrency as best possible

Safe and correct? What is that?

- PostgreSQL doesn't like arbitrating data races, because that is our responsibility

So for non-contending commands, it's totally "safe" right?

- Yeah, it is. But remember that even though commands may not appear to be directly contending in the data they affect, their ultimate effects may be so

What? How come?

```
insert into byte_test values('a');
insert into byte_test select count(1)::char from byte_test where test_byte = 'a';
```


Confidential

So how does this isolation thing fit in with WAL?

So to keep things "isolated", does PostgreSQL commit the WAL entries resulting from transactions as "blocks", atomically, frog-leaping each other?

- That would be a good guess. However, let us consider two things:
  - The WAL is the guarantee from the database that our operation was recorded. If PostgreSQL were to queue up the WAL equivalents of all the commands in our transaction to atomically commit them behind another batch from another transaction, that means we're synchronously waiting, killing concurrency
  - In order to perform the translation of the command into a WAL record, PostgreSQL needs to execute the command. But, the result of the execution of a command may be dependent on concurrently executing commands. And so, if this were the case, PostgreSQL cannot even execute commands to generate WAL entries until all commands from the previous transaction were executed, translated and atomically committed

So…no?

- No. Isolation does not imply serialization. It is simply a function of visibility, and PostgreSQL can control what is visible by using transactions, the isolation level and MVCC
- WAL entries for commands from different transactions can happily overlap because of the "safe" and "correct" guarantee of PostgreSQL's concurrency
- Only when there is a conflict between commands, is their execution and consequently their WAL entries serialized, thus continuing the motto of safety and correctness

Confidential

So transactions are really just visibility markers?

That's definitely a big part of it. Transactions don't mean that commands are being serialized, they just mean that whatever is happening inside them is not yet visible to the rest of the world

Remember that transactions also give us the power to commit or rollback all of our changes at once

So transactions are responsible for giving us Atomicity and Isolation?

- Yep. One could definitely argue that transactions are key to the design of PostgreSQL and RDBMSes in general delivering these two properties

Confidential

Phew. OK, moving on to delete, I guess those operate like updates?

You got it. PostgreSQL due to MVCC won't actually delete data, just mark its visibility "dead" after a certain tx_id using `xmax`

And so if I really wanted my space back…?

- Yep, you have to VACUUM FULL the table
- Or, you could TRUNCATE the table. Note that this operation will also create a new filenode

Wait. So, all these "dead tuples" from DELETEs and UPDATES – can they affect my operations?

- Oh yeah. These dead tuples are commonly referred to as bloat
- Bloat is dead space caught between visible tuples. This affects scan operations on the data

Confidential
enova.

How does bloat affect me?

Amongst other things, bloat can have adverse impacts on sequential scans.

- To prove this we need to cheat a little by using more than just a single byte of data:
  - We're going to create a row with a byte value of '1', then a million rows with values of '2' and one more with a value of '3'
  - We set `shared_buffers` to a low value of 128kB
  - We delete all the rows with a value of '2'
  - Then we ask PostgreSQL to get all values equal to '1' or '3' from the table
- PostgreSQL has to keep swapping in pages with only dead tuples even after a VACUUM because of the bloat
- If we VACUUM FULL, then the bloat is purged and we see the #page swaps go down significantly, speeding up our query dramatically

```
- insert into byte_test values('1');
- insert into byte_test select '2' from generate_series(1,1000000);
- insert into byte_test select '3' from generate_series(1,1);
- explain (buffers, analyze) select * from byte_test where test_byte
  in('1', '3');
- delete from byte_test where test_byte = '2';
- vacuum byte_test;
- vacuum full byte_test;
```


Confidential

So I just keep calm and VACUUM FULL?

Keeps my disk consumption in check and my bloat low right?

- Sure, but it kills concurrency by taking a table lock since it’s a rewrite
- Also, remember that “dead space” isn’t necessarily wasted space. The storage engine is going to re-use that for newly inserted data

So, what’s the trick then?

- Hire a DBA
- There is no magical one-size-fits-all solution. It’s a full time job running a high-performance system, depending on a lot of characteristics ranging from usage patterns to hardware

Confidential

And so where is my byte now, after VACUUM FULL?

Gone. Obliterated.

- And that's the journey of a single byte – birth to death

So there's no way I can get it back?

- Oh, of course there is
- Remember the WAL? Remember that it is append only? We can use that fact for PITR – point in time recovery by playing out all the data file operations recorded in the WAL right up to the point where we purged everything and now our byte is back!

Confidential
enova.

Thank You

PostgreSQL is a database. What is a database again?

An abstraction for management of data.

- Creating data
- Updating it
- Querying it in a timely fashion
- Securing it
- Purging it

How do databases work?

- Structure (mandatory)
- Process (optional)
- Administration (recommended)

Do databases need to be persistent?

- No

What is the cost of persistence?

- Latency

Would people use databases if they were not persistent?

- Yes

Which is easier – making something persistent not, or otherwise?

- Turning off persistence is definitely easier

Confidential

So, PostgreSQL is durable?

You bet! And it can be otherwise if you were to wish it so.

- Caution: wish carefully.
- In fact, turns out the ability to support persistence of data is one of the 4 core conditions to meet for ACID compliance:
  - **A**tomic: I have the ability to perform a batch of operations or if it fails, none at all
  - **C**onsistent: Operations on the database take it from one state of integrity to the next
  - **I**solated: I can isolate my operations from the effect of other concurrent operations
  - **D**urable: persistent, can recover from crashes

What does this mean?

- It means that if you were to gracefully shut down your database and start it back up, it remembers the state of your data as it was
- It also means that if your database were to crash for whatever reason, you aren't left with nothing – you can restore your database to a last-best-known-state
- It means that some process is tracking and noting down changes you're making and saving that somewhere "safe".

Oh, where?

- Well, on arguably one of the best abstractions of operating systems – the file system!

Confidential