PostgreSQL: a look to the engine

The Ravenous Bugblatter Beast of Traal

Federico Campoli

Brighton PostgreSQL Users Group

13 June 2016

Table of contents

Table of contents

Don't panic!

Copyright by Kreg Steppe - https://www.flickr.com/photos/spyndle/

Don't panic!

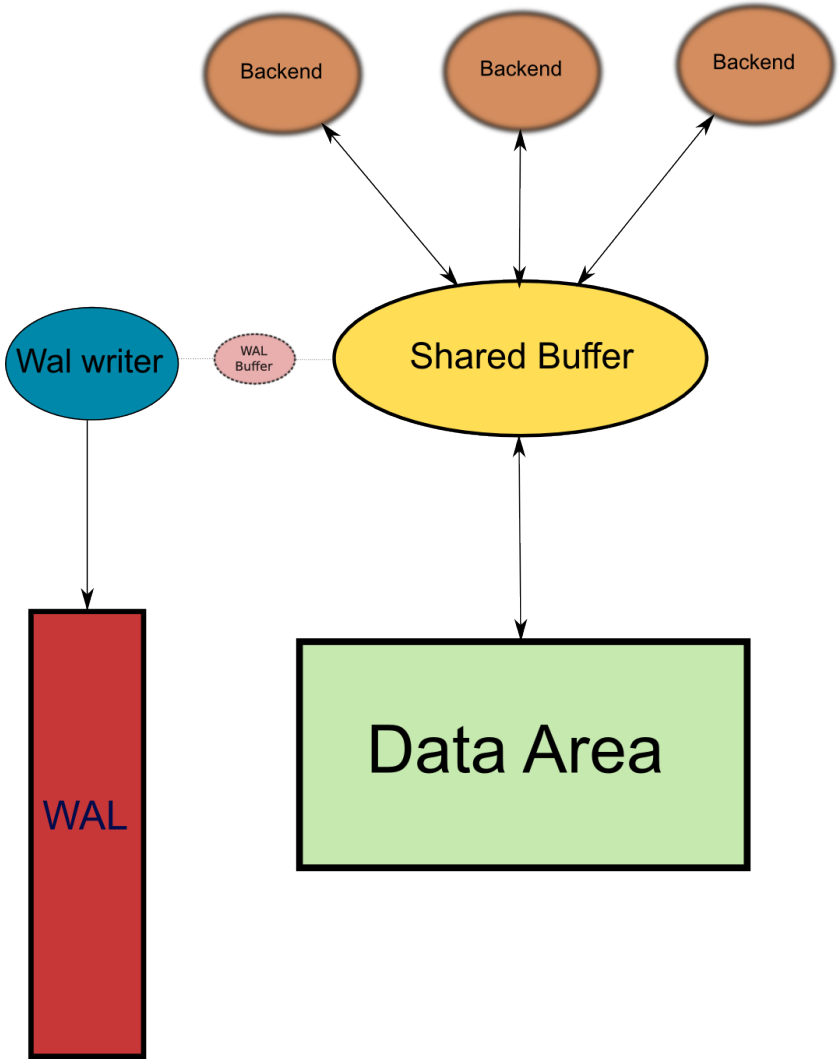
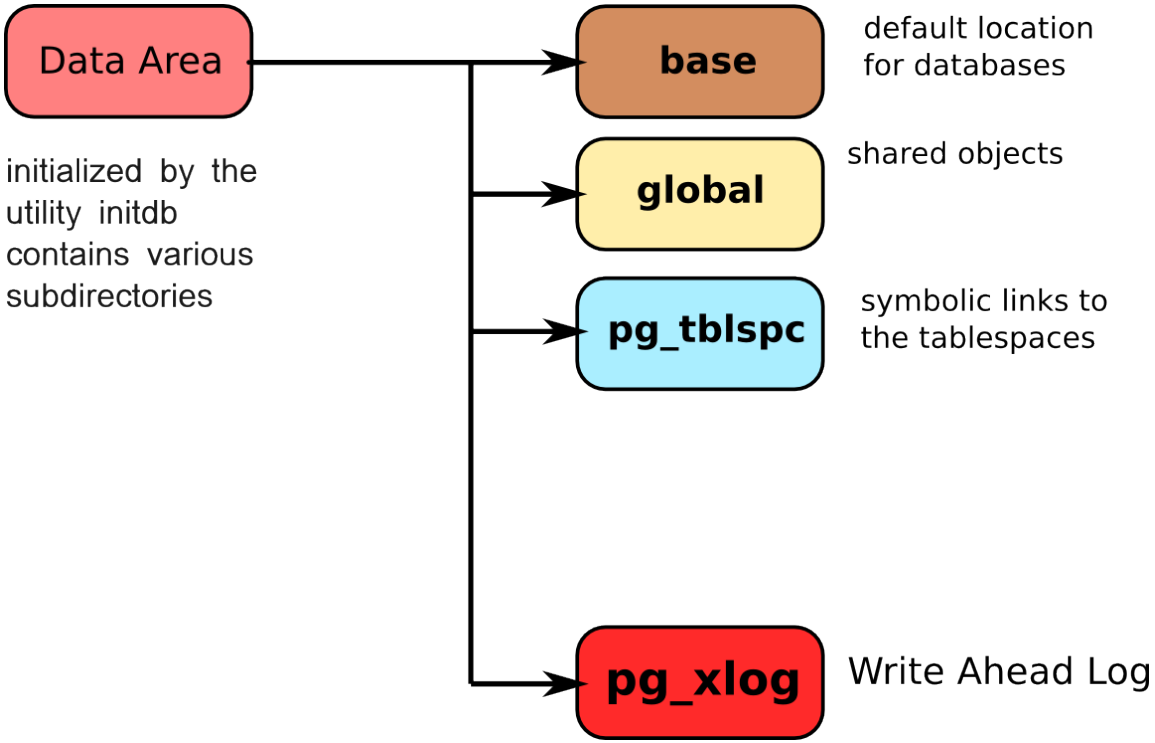

Backend
Backend
Backend
Wal writer
WAL Buffer
Shared Buffer
WAL
Data Area

Table of contents

Infinite Improbability Drive

The data area

The PostgreSQL's data area is the directory where the cluster stores the data on durable storage.



The directory base

The default location when a new database is created without the TABLESPACE clause.

- Inside, for each database there is a folder with numeric names
- An optional folder pgsql_tmp is used for the external sorts
- The base location is mapped as pg_default in the pg_tablespace system table

The directory base

Each database directory contains files with numeric names where postgres stores the relation's data.

- The maximum size a data file can grow is 1 GB, a new file is generated with a numerical suffix
- Each data file is organised in fixed size pages of 8192 bytes
- The data files are called file nodes. Their relationship with the logical relations is stored in the pg_class system table

The directory pg_global

The directory pg_global contains the data files used by the relations shared across the cluster.

There is also a small 8kb file named pg_control. This is a very critical file where PostgreSQL stores the cluster's vital data like the last checkpoint location.

A corrupted pg_control prevents the cluster's start.

The directory pg_tblspc

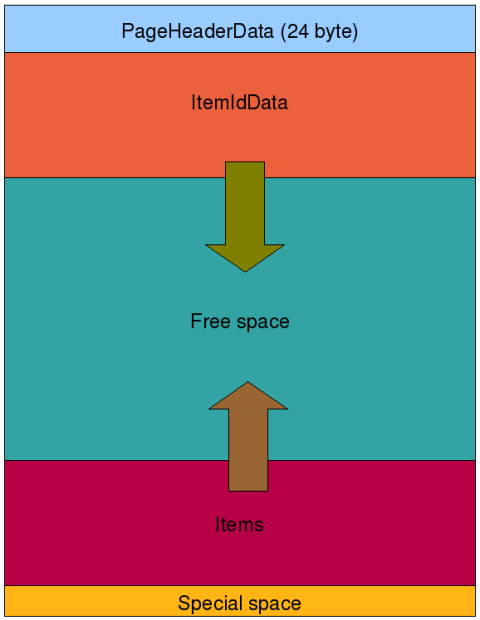
- Contains the symbolic links to the tablespaces.
- Very useful to spread tables and indices on different physical devices
- Combined with the logical volume management can improve dramatically the performance...
- or drive the project to a complete failure
- The objects tablespace location can be safely changed but this require an exclusive lock on the affected object
- the view pg_tablespace maps the objects name and identifiers

The directory pg_xlog

- The write ahead logs are stored in this directory
- Is probably the most important and critical directory in the cluster
- Each WAL segment is 16 Mb
- Each segment contains the records describing the tuple changed in the volatile memory
- In case of crash or an unclean shutdown the WAL are replayed to restore the cluster's consistent state
- The number of segments is automatically managed by the database
- Putting the location on a dedicated and high reliable device is **vital** for performance and reliability

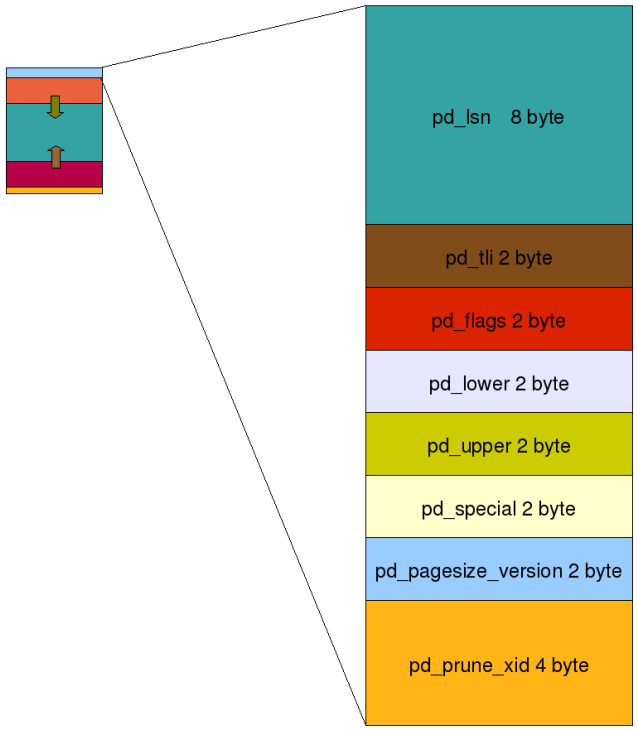
Data pages

Each block is structured **almost** the same, for tables and indices.



Page header

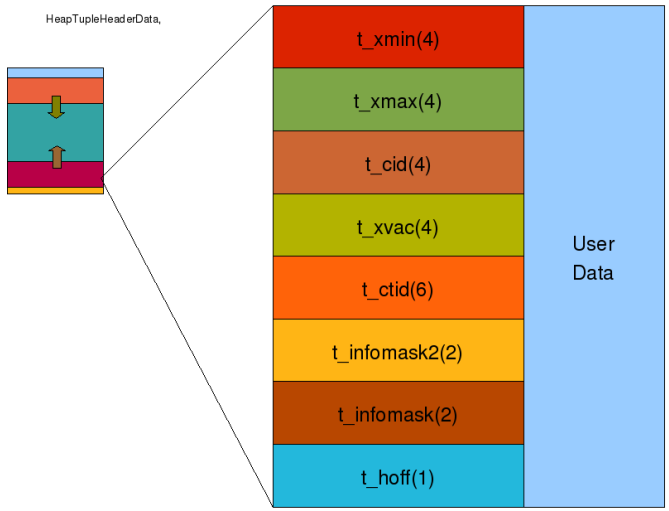
Each page starts with a 24 bytes header followed by the tuple pointers. Those are usually 4 bytes each and point the physical tuples are stored in the page's end



The tuples

Now finally we can look to the physical tuples. For each tuple there is a 27 bytes header. The numbers are the bytes used by the single values.

The user data can be either the data stream or a pointer to the out of line data stream.



TOAST with Marmite please

Copyright by David Martyn Hunt
TOAST, *the best thing since sliced bread*

- TOAST is the acronym for The Oversized Attribute Storage Technique
- The attribute is also known as **field**
- The TOAST can store up to 1 GB in the out of line storage (and free of charge)

TOAST with Marmite please

- Fixed length data types like integer, date, timestamp do not are not TOASTable.
- The data is stored after the tuple header.
- Varlena data types as character varying without the upper bound, text or bytea are stored in line or out of line.
- The storage technique used depends from the data stream size, and the storage method assigned to the attribute.
- Depending from the storage strategy is possible to store the data in external relations and/or compressed using the fast zlib algorithm.

TOAST with Marmite please

TOAST permits four storage strategies (shamelessly copied from the on line manual).

- PLAIN prevents either compression or out-of-line storage; This is the only possible strategy for columns of non-TOAST-able data types.
- EXTENDED allows both compression and out-of-line storage. This is the default for most TOAST-able data types. Compression will be attempted first, then out-of-line storage if the row is still too big.
- EXTERNAL allows out-of-line storage but not compression. Use of EXTERNAL will make substring operations on wide text and bytea columns faster at the penalty of increased storage space.
- MAIN allows compression but not out-of-line storage. Actually, out-of-line storage will still be performed for such columns, but only as a last resort.

TOAST with Marmite please

When the out of line storage is used the data is encoded in bytea and eventually split in multiple chunks.

An unique index over the chunk_id and chunk_seq avoid either duplicate data and speed up the lookups

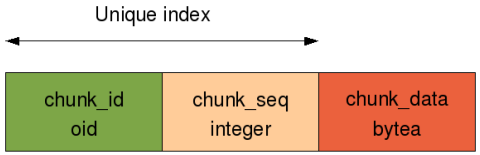


Figure : Toast table

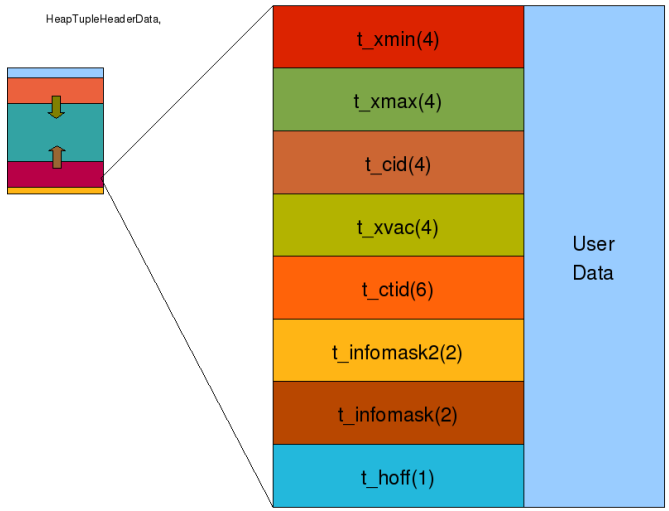
Table of contents

Time is an illusion. Lunchtime doubly so

Copyright by Federico Campoli

The magic of the MVCC

- t_xmin contains the xid generated at tuple insert
- t_xmax contains the xid generated at tuple delete
- t_cid contains the internal command id to track the sequence inside the same transaction



The magic of the MVCC

The PostgreSQL's consistency is achieved using MVCC which stands for Multi Version Concurrency Control.

The base logic seems simple.

- A 4 byte unsigned integer called **xid** is incremented by 1 and assigned to the current transaction.
- Every committed xid which value is lesser than the current xid is considered in the past and then visible to the current transaction.
- Every xid which value is greater than the current xid is in the future and then invisible to the current transaction.
- The commit status is managed in the $PGDATA using the directory pg_clog where small 8k files tracks the transaction statuses.

The magic of the MVCC

In this model there is no UPDATE field. Every time a row is updated it simply generates a new version with the field t_xmin set to the current XID value.

The old row is marked **dead** just by writing the same XID in the t_xmax.

Table of contents

The Pan Galactic Gargle Blaster

Copyright by Federico Campoli

A little history

Back in the days, when the world was young, PostgreSQL memory was managed by a simple MRU algorithm.

The new 8.x development cycle introduced a powerful algorithm called Adaptive Replacement Cache (ARC) where two self adapting memory pools managed the most recently used and most frequently used buffers.

Because a software patent on the algorithm shortly after the release 8.0 the buffer manager was replaced by the two queue algorithm.

The release 8.1 adopted the clock sweep memory manager still in use in the latest version because of its flexibility.

The clock sweep

The buffer manager's main goal is to keep cached in memory the most recently used blocks and adapt dynamically for the most frequently used blocks.

To do this a small memory portion is used as free list for the buffers available for memory eviction.

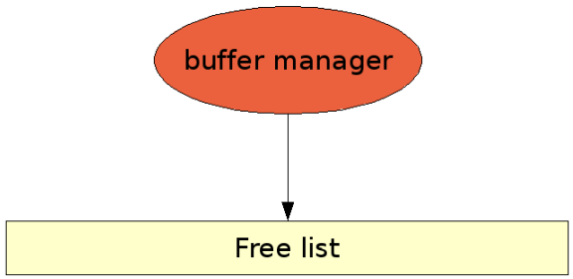


Figure : Free list

The clock sweep

The buffers have a reference counter which increase by one when the buffer is pinned, up to a small value.

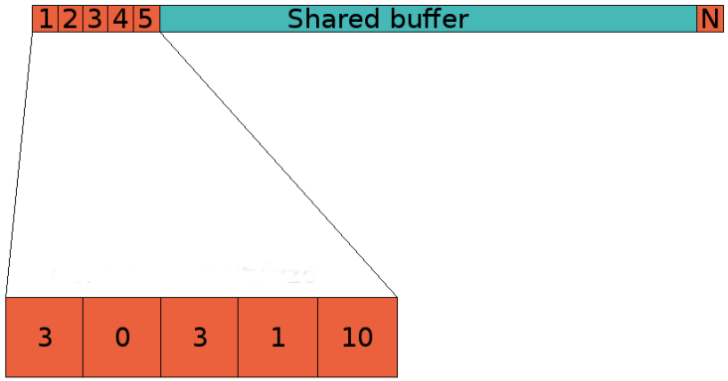


Figure : Block usage counter

The clock sweep

Shamelessly copied from the file src/backend/storage/buffer/README

There is a "free list" of buffers that are prime candidates for replacement. In particular, buffers that are completely free (contain no valid page) are always in this list.

To choose a victim buffer to recycle when there are no free buffers available, we use a simple clock-sweep algorithm, which avoids the need to take system-wide locks during common operations.

The clock sweep

It works like this:

Each buffer header contains a usage counter, which is incremented (up to a small limit value) whenever the buffer is pinned. (This requires only the buffer header spinlock, which would have to be taken anyway to increment the buffer reference count, so it's nearly free.)

The "clock hand" is a buffer index, NextVictimBuffer, that moves circularly through all the available buffers. NextVictimBuffer is protected by the BufFreelistLock.

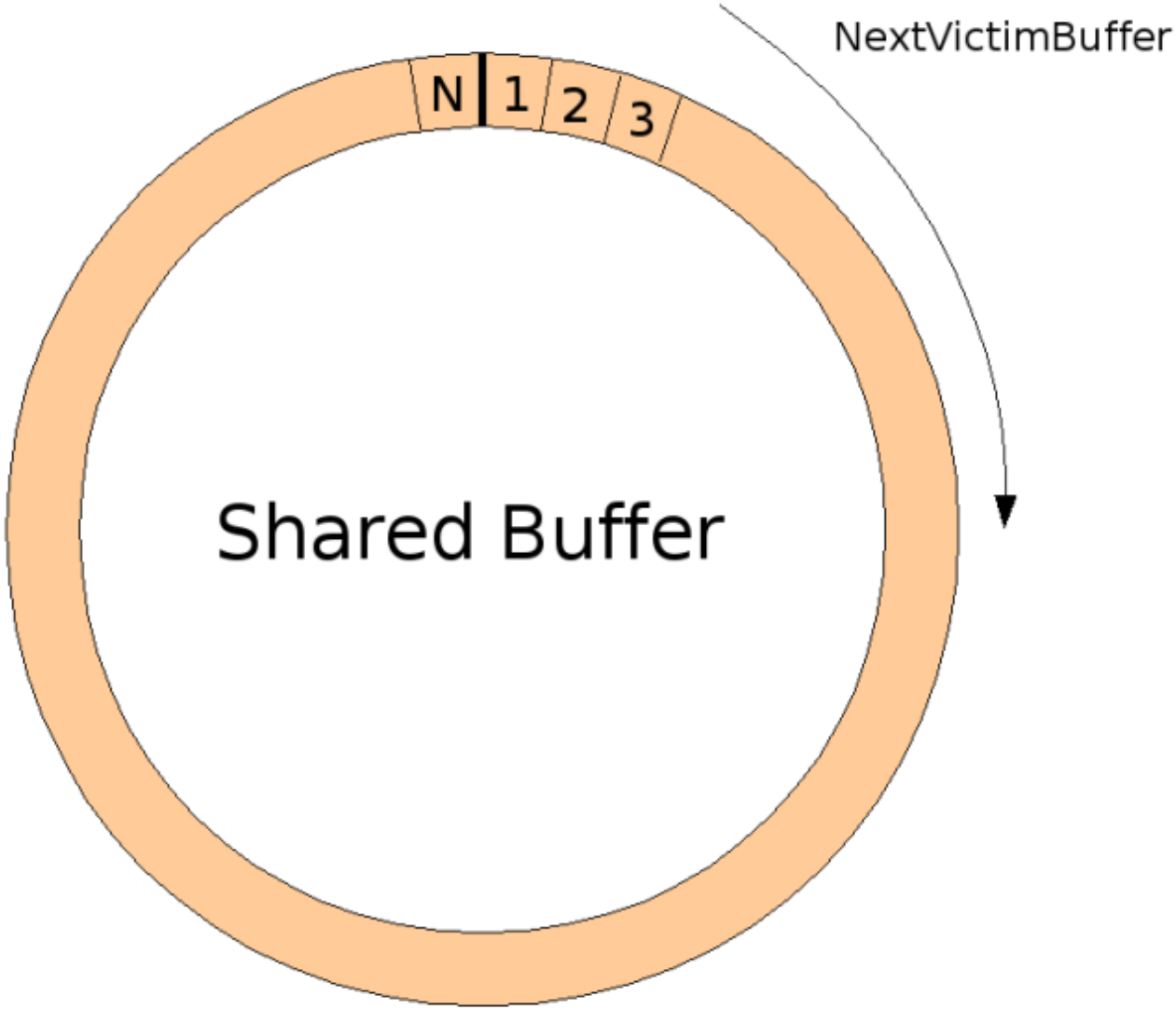
It's bigger in the inside

The algorithm for a process that needs to obtain a victim buffer is:

1. Obtain BufFreelistLock.
2. If buffer free list is nonempty, remove its head buffer. If the buffer is pinned or has a nonzero usage count, it cannot be used; ignore it and return to the start of step 2. Otherwise, pin the buffer, release BufFreelistLock, and return the buffer.
3. Otherwise, select the buffer pointed to by NextVictimBuffer, and circularly advance NextVictimBuffer for next time.
4. If the selected buffer is pinned or has a nonzero usage count, it cannot be used. Decrement its usage count (if nonzero) and return to step 3 to examine the next buffer.
5. Pin the selected buffer, release BufFreelistLock, and return the buffer.

(Note that if the selected buffer is dirty, we will have to write it out before we can recycle it; if someone else pins the buffer meanwhile we will have to give up and try another buffer. This however is not a concern of the basic select-a-victim-buffer algorithm.)

It's bigger in the inside



It's bigger in the inside

Since the version 8.3 the buffer manager have the ring buffer strategy.
Operations which require a large amount of buffers in memory, like VACUUM or large tables sequential scans, have a dedicated 256kb ring buffer, small enough to fit in the processor's L2 cache.

Table of contents

Mostly harmless

Copyright by Federico Campoli

The XID wraparound failure

- XID is a 4 byte unsigned integer.
- Every 4 billions transactions the value wraps
- PostgreSQL uses the *modulo* -2^{31} comparison method
- For each value 2 billions XID are in the future and 2 billions in the past
- When a xid's age becomes too close to 2 billions VACUUM freezes the xmin value to an hardcoded xid forever in the past

The XID wraparound failure

If for any reason an xid reaches 10 millions transactions from the wraparound failure the database starts emitting scary messages

```
WARNING:  database "mydb" must be vacuumed within 5770099 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in "mydb".
```

If a xid's age reaches 1 million transactions from the wraparound failure the database simply shut down and can be started only in single user mode to perform the VACUUM.

Anyway, the autovacuum deamon, even if turned off starts the required VACUUM long before this catastrophic scenario happens.

Questions?

Questions?

Boring legal stuff

All the images copyright is owned by the respective authors. The sources the author's attribution is provided with a link alongside with image.

The section's titles are quotes from Douglas Adams hitchhiker's guide to the galaxy. No copyright infringement is intended.

Contacts and license

- Twitter: 4thdoctor_scarf
- Blog:http://www.pgdba.co.uk
- Brighton PostgreSQL Meetup: http://www.meetup.com/Brighton-PostgreSQL-Meetup/

PostgreSQL: a look to the engine

The Ravenous Bugblatter Beast of Traal

Federico Campoli

Brighton PostgreSQL Users Group

13 June 2016