PERCONA

Databases run better with Percona

PERCONA

TDE as an Extension

A Different Path for PostgreSQL Encryption

A Different Path for PostgreSQL Encryption

- About the journey
- How a simple concept turns into a complex topic
- What did we do?
- What mistakes did we make?
- What could have been done better?

Transparent data encryption

- Encrypt the data before writing to the disk
- Without query changes
- Available in many SQL databases already
- Policy requirement

TDE: debate

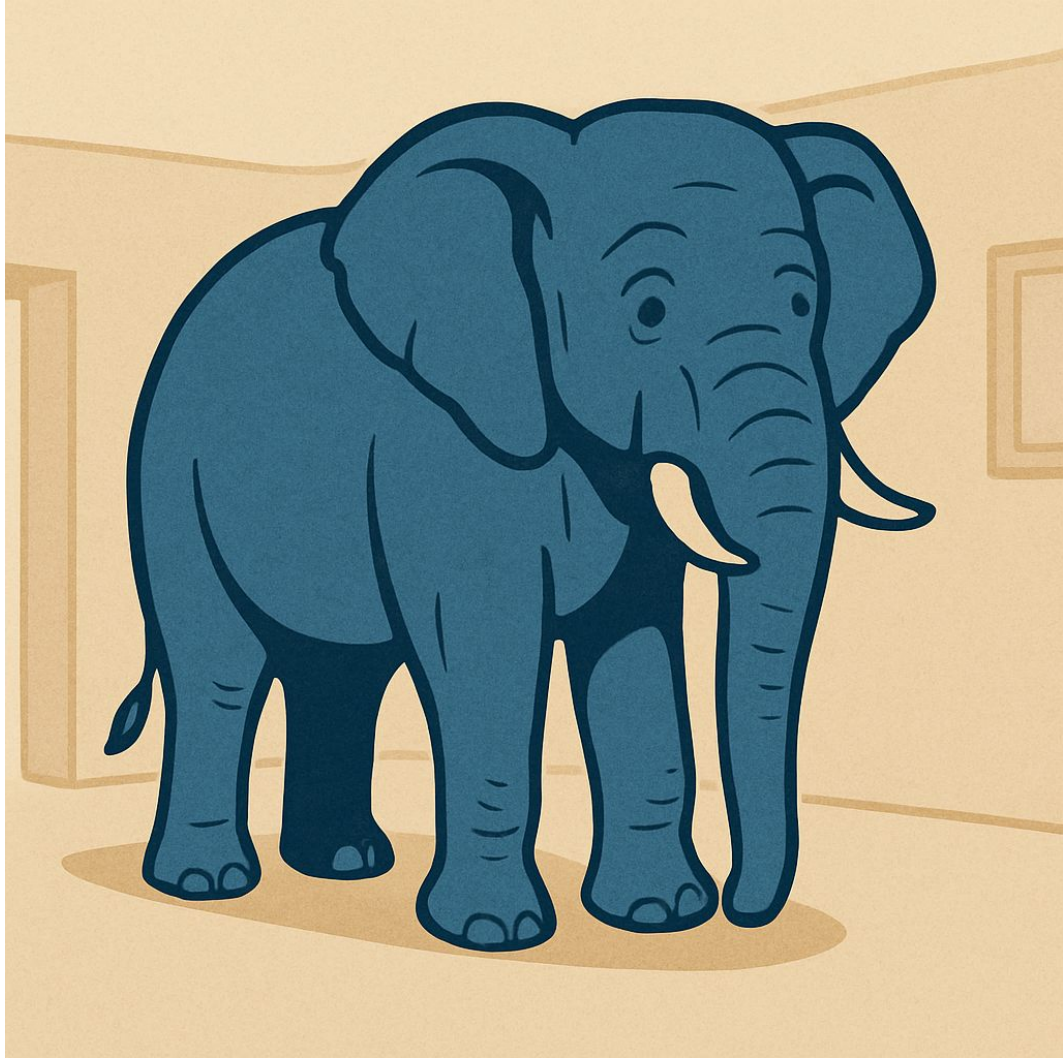
- Is it worth it?
  - Encrypted disks / filesystems
  - The memory remains unencrypted
  - Performance loss, even when not using it

TDE: debate

- Choices
  - What algorithms to support?
  - What to encrypt?
  - When to encrypt?
  - Global or customizable?

The elephant in the room

- TDE is a huge and difficult topic
- PostgreSQL is Community driven
- No single decision maker
- First step: agree on the feature



TDE In PostgreSQL

- First patch in 2016: cluster wide, single key
- 2018: table level granularity
- 2019: key management
- Huge patch
- Long discussions on pg-hackers
- 2025: still no consensus

TDE In PostgreSQL: commercial solutions

- Closed source enterprise forks
- With encryption features

TDE In PostgreSQL: step by step

- Many small patches instead of one huge patch
- Unified file system API
- Transparent column encryption
- Special page storage
- …
- Not much progress yet

Can we provide an open source version?

- Creating a fork
  - Maintaining the already existing patchset as a fork?
  - Writing a completely new patchset from scratch?
- Or trying to do it as an extension?
  - Without core changes
  - With minimal core changes

Can we provide an open source version?

- Creating a fork
  - Maintaining the original patchset as a fork?
  - Writing a completely new patchset from scratch?
- Or trying to do it as an **extension**?
  - **Without core changes**
  - With minimal core changes

Access methods

- Table access methods - storage for tables
- Index access methods - storage for indexes
- Postgres already provides extension points
- CREATE TABLE ... **USING <access_method>**;
- Heap - the default table access method

Let’s duplicate heap!

1. Extract the code from postgres
2. Put it into an extension
3. Rename it to tde_heap
4. Add the minimal modifications required for encryption

Let’s duplicate heap!

- Proof of concept done in a few days
- Executes simple queries without issues
- Success!?

Q: How can we support other storage engines?

Q: How can we handle indexes?

Internals

- Hook into tuple retrieval/saving
- When heap loads a tuple, add decryption code
- When heap persists a tuple, add encryption code
- Everything happens within the connection handler
- Buffer pool stores encrypted records
  - More than data-at-rest!

Challenge: memory management

- Where do we store the decrypted data?
- Decrypt in buffer pool
  - Would require some flags/modifications
- Decrypt in local process
- Most of the code works with one tuple at a time
  - Load - use - save/forget - go to next tuple

Challenge: memory management

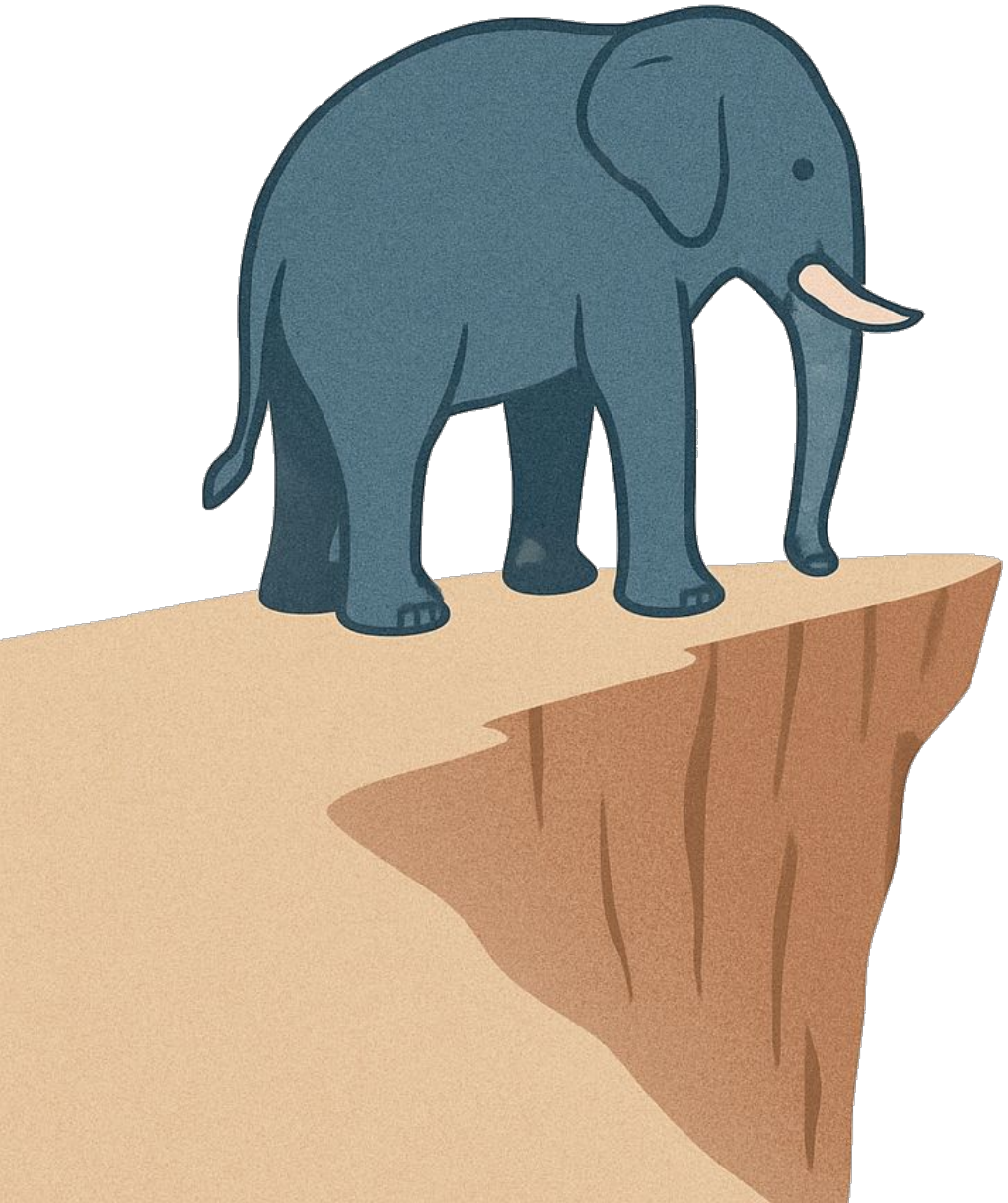
- **Most of the code** works with one tuple at a time
- An exception: scans
- Initial symptom: data corruption
- First fix: keep decrypted tuples until end of scope
- Huge memory spike
- Proper fix with modified slots later

Challenge: performance

- Promising initial pgbench/sysbench results
- But: no index encryption
- Basic perf tests use index scans
- Sequential scans: terrible result

How can we handle indexes?

- Duplicate all index methods?
- What about extensions?
- Performance concerns
- Security concerns
  - Only tuple data is encrypted, not header
  - Structure remains the same and is visible



Dead End

- Unfixable performance problem
- Questions about correctness
  a. Heap is huge
  b. Are there more hidden problems?
- Is it good enough without index encryption?

Let’s try it differently

- Creating a fork
  - Maintaining the original patchset as a fork?
  - Writing a completely new patchset from scratch?
- Or trying to do it as an **extension**?
  - Without core changes
  - **With minimal core changes**

Storage manager

- Layer between postgres and disk for database objects
- Designed for extensibility
- Extension point not exposed for extensions
- There’s a proposed patch
- Patch is already used by multiple companies

Storage manager

- Layer between postgres and disk for **database objects**
  - Tables
  - Indexes
  - Anything else visible in the database
- NOT: WAL
- NOT: temporary files

Let’s add extension points!

- Take existing SMGR patch
- Add similar extension point for WAL
- Future: add extension point for temporary files
- Minimal core patches
- Easy to maintain fork

Reuse what we can

- Keep table access method as a marker for encryption
- Indexes/sequences/… inherit encryption status
- Keep complete user facing API as is
- Proof of concept done in a few days
- Executes simple queries without issues
- Success!?

Challenge: wrong abstraction level

- SMGR deals with raw data and file identifiers
- Access methods are database concepts
- Bridge: event triggers

Event triggers

- User code executed before/after DDL
- If in CREATE TABLE … USING tde_heap;
- Then write encrypted objects
- If CREATE INDEX or CREATE SEQUENCE or …
- Then check encryption of parent table

Problem: table rewrite

- Some commands completely rewrite tables
- In practice: create files with new filenames
- Do we have to encrypt these new files?
- Some commands do not trigger event triggers
  - VACUUM FULL …
- One more core patch

Problem: multiple tables in one statement

- Table inheritance
- Typed tables
- Partitioned tables
- One partition encrypted, another not
- Altering them both in one ALTER
- No good solution, we report an error instead

Problem: WAL encryption

- Not an issue with first solution
  - Modified heap code adds tuples to WAL after encryption
  - Encrypted tables write encrypted WAL
  - Plain tables write plain WAL
- New solution writes clear unencrypted WAL
- WAL code needs new extension points

Problem: wrong abstraction level

- WAL code doesn’t know about database objects
- Can’t tell if we write WAL for encrypted or plain tables
- Workaround:
  - separate setting
  - encrypt entire WAL
  - or don’t encrypt it at all

Problem: WAL is sometimes rewritten

- Original assumption: WAL is write-once, read often
  - we can always write with the latest encryption key
- Reality: WAL is rewritten in some corner cases
- Debugging nightmare
- Error happens during write
- But only discovered later, **IF** something tries to read it back

Problem: tool ecosystem

- WAL files are completely encrypted
- Not just the records in them
- Tools can no longer read them
  - waldump
  - backup tools
  - WAL archiving

Problem: tool ecosystem

- Duplicate some tools
  - pg_waldump -> pg_tde_waldump
- Write decrypted data for backup streams
  - Not ideal
  - Backup tools have their own encryption support

Future: not yet done

- Upstreaming new extension points
- Temporary files
- System tables
- initdb
- aio

Future: tablespace as a marker?

- “Keep table access method as a marker for encryption”
- The main source of many problems
- Could we do better by using tablespaces?

Future: multitenancy?

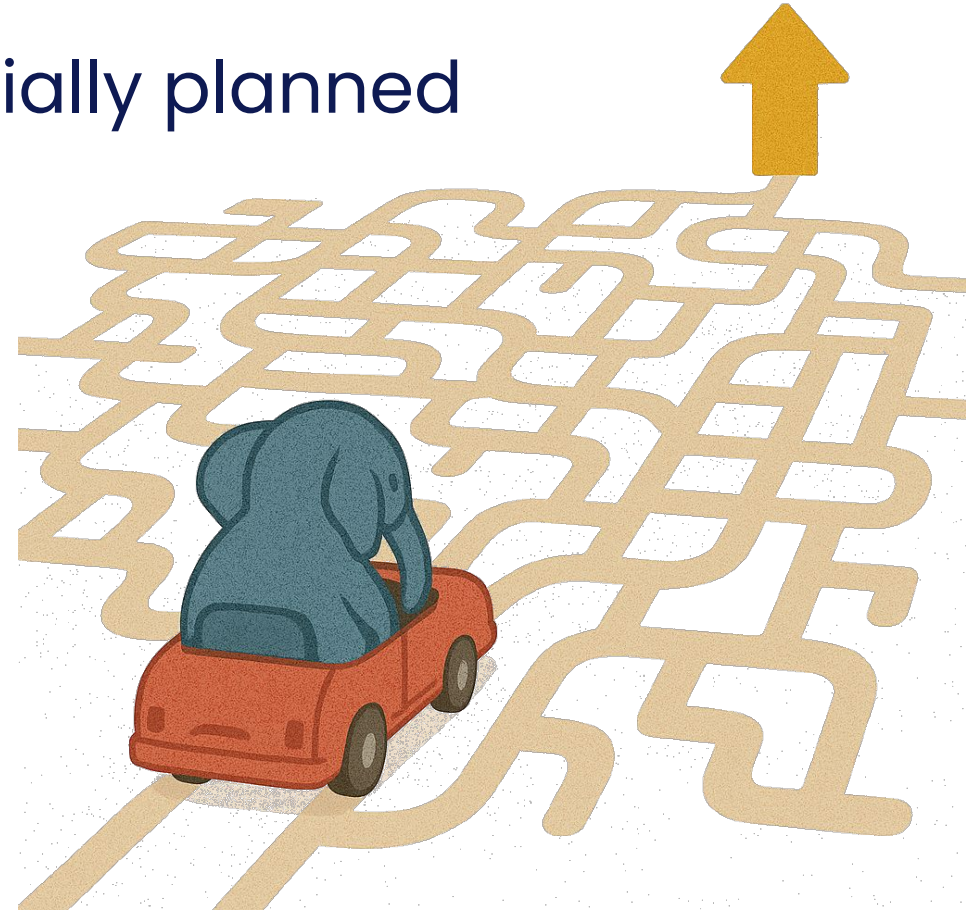
- “Keep complete user facing API as is”
- Multitenancy: each database configured individually
  - Different keys
  - Different key servers
- Great feature in the first prototype

Future: multitenancy?

- WAL is global, it has its own separate key
- Recovery needs access to **all** key servers
- If even one is unreachable, recovery fails
- Defining new key servers requires SUPER

Summary

- Complex projects often don’t go as initially planned
- Our first idea looked great initially…
- … but we had to abandon it later
- Second idea also had major problems
- Still has open questions
- But it definitely works!



Takeaways

- Don’t delay detailed testing
- Be very critical about (performance) tests
- Don’t be afraid to change things
- Keep questioning the design

Thank You!