Module 1

Storing acid data

Transaction

A sequence of **Read** and **Write** operations that has the four well known **ACID** properties Remember what they stand for?

ACID

Atomicity: **All** changes commit or **none Consistency**: They only transition the system from a **valid** state to another **Isolation**: They execute **isolated** from each other

> More on this later

Durability: Committed transactions **persist** despite system failures

---

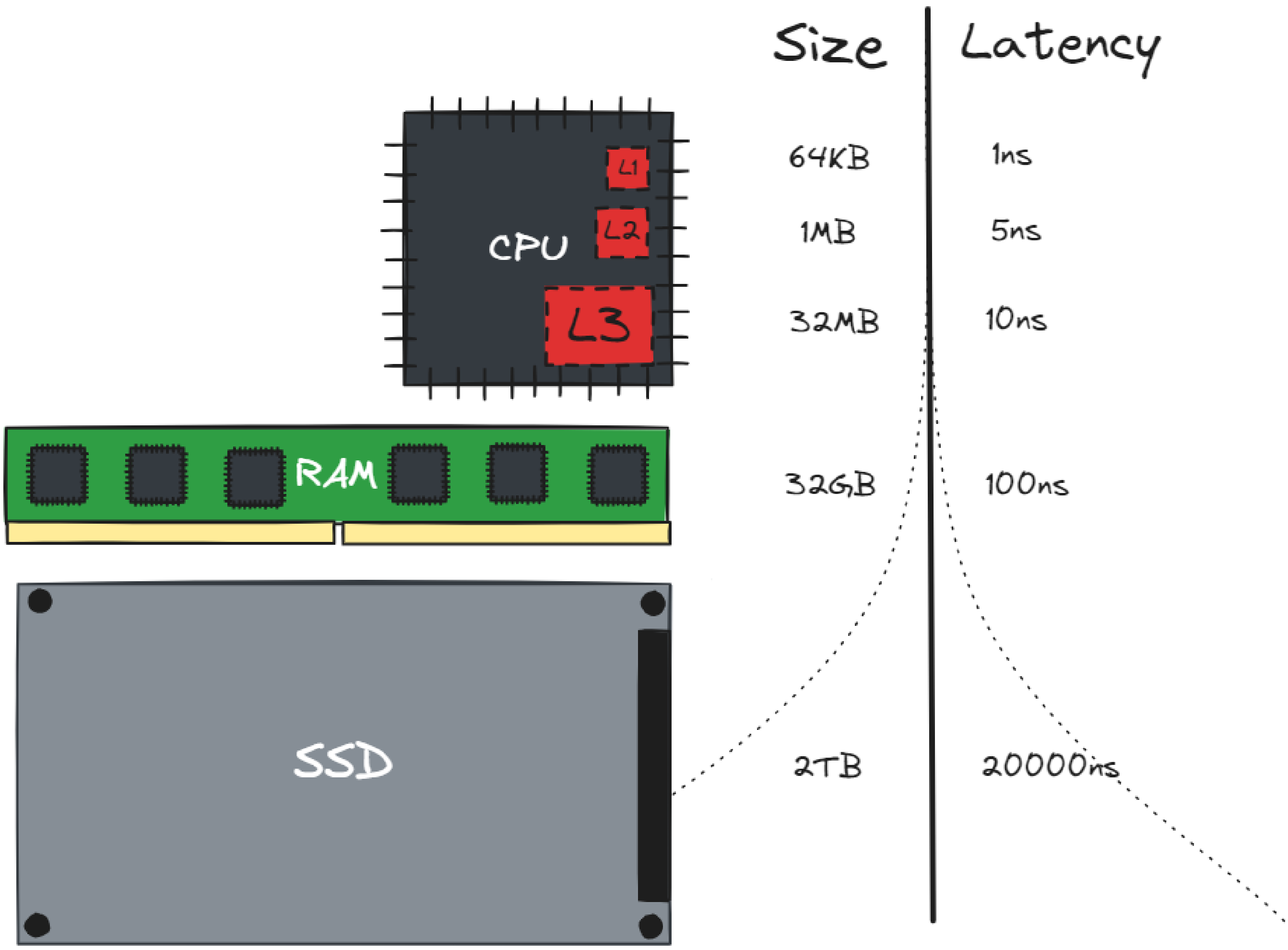
ACID

Youc an see this as **expectations** your users will have from your system

- e.g: in the previous exercise, a user would assume that calling `set` successfully guarantees that data has been **written** to disk

> Spoiler alert, this was **not the case**, which explains why there was a task asking to to run `sync` for every request, which was **insanely slow**

Because disks are indeed slow



Buffered vs Direct I/O

In the previous exercise, you were using **buffered I/O**. You were actually writing pages in **memory**

The kernel would eventually write them to disk after some time (generally 30s maximum). It does this precisely because **disks are slower than memory**

⚠ Yes, this means that in the event of a **power failure**, data was **lost**

But that's not the worst

You were modifying data ***in place***

Your system did not provide any way to **group operations together**

Nor to **cancel** all of them if something goes wrong

> Which is the definition of atomicity

Coming back to our problem

How do we come up with an implementation that

- Provides **durability** guarantees
- While also allowing to `ROLLBACK` a change (or a set thereof)
- **Bonus point** for some smart caching for reads

▯ Any guess?

Coming back to our problem

How do we come up with an implementation that

- Provides **durability** guarantees
- While also allowing to `ROLLBACK` a change (or a set thereof)
- **Bonus point** for some smart caching for reads

▯ Any guess? a **Log** and a **Buffer Pool**

The WAL or Write-Ahead Log

All **operations** are written in a log

Every line in the log has **enough information** to replay the operation to provide **P**oint-**I**n-**T**ime **R**ecovery

> ⚠ **All log entries** for a transaction must be physically written to disk using `open(O_DIRECT)` before the transaction can `COMMIT`, hence the **write-ahead**

Buffer Pool

Most databases choose to implement their **own buffer**, sitting **between** the database process and the filesystem.

> Because the database **knows** what it should retrieve, when it should do it, and how long it should keep it

---

Buffer Pool

The Buffer decides **when** data is read from disk to memory, and **when** pages are persisted to disk

PostgresSQL uses **Buffered I/O** there

😱 Wait, Postgres uses Buffered I/O?

It started as a **research** project in the 1980s and only has still a few contributors

Their focus was on the **database engine** itself and a Direct I/O stack is **hard to implement**

> e.g. need to manage how to **group** write operations together on the storage device, which the OS kundly does for you

---

😱 Wait, Postgres uses Buffered I/O?

And remember this is not an issue for **durability**, as the log itself has all the information and uses `IO_DIRECT`

Wait, my log is growing indefinitely

Run `CHECKPOINT` regularly

Checkpoint

Happens automatically **every few minutes**, done by the **Background Writer**

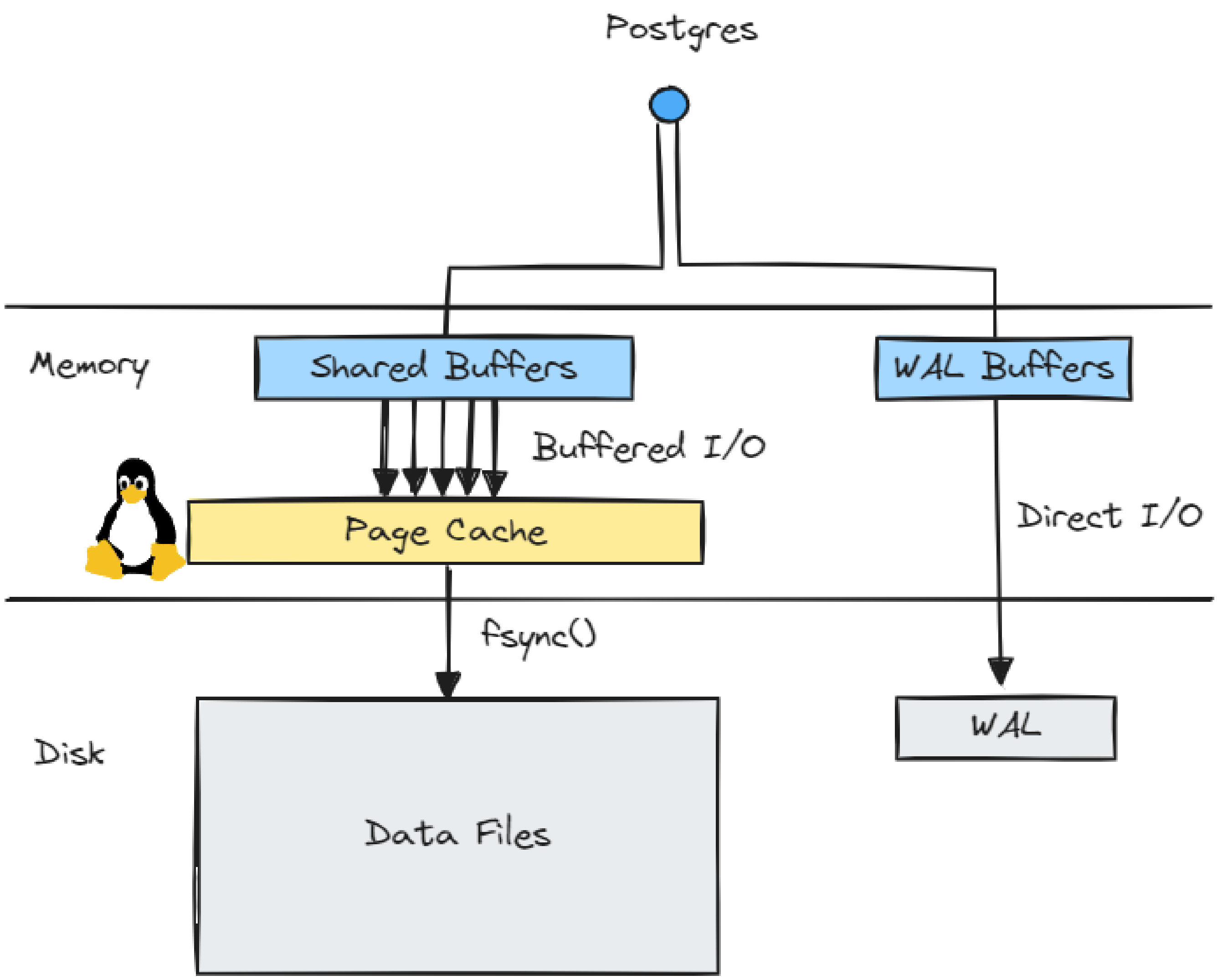
1. Get the current WAL **position**
2. Write outdated data from Buffer Pool to the **Linux Page Cache**
3. Call `fsync` on **all those files** which ensures they are physically written to disk
4. **Mark** the new point of recovery, and clean the log entries before

---

Checkpoint

□ Did you recently hear about **issues** there (think of `fsync`)?

Recap of Postgres storage



What are we storing exactly

Database files are split in units of a few KBs called **pages**

A page is the **smallest** unit of storage the database can interact with

> ➡ Postgres uses 8KB pages by default (same for Microsoft SQL Server)

The **Buffer Pool** is thus a **cache of pages**, organized as an in-memory array

---

Tuple Alignment

Tuples are stored on multiples of 64 bits.

Some systems **reorganize columns** in the tuple

PostgreSQL will **pad** every type to make sure that everything is **64-bit** aligned

> For big enough values, store them in specific pages (**TOAST** pages in Postgres) **TOAST** = **T**he **O**versized-**A**ttribute **S**torage **T**echnique

---

`NULL` handling

□ How are `NULL`s stored?

`NULL` handling

□ How are `NULL`s stored?

1. **Reserve** a value
2. Use a bitmap in the row **header**

Dealing with concurrency

Our homemade database was only used by us

In real life we would have had **multiple users** running multiple concurrent queries

> What problems does it cause?

Isolation Anomalies

Isolation is another story We are trying to prevent the following to happen:

- **Dirty** reads: reading uncommitted changes
- **Non-repeatable** reads: reading a value that has been **updated** by another committed transaction
- **Phantom** reads: reading a value that has been **inserted** by another committed transaction

---

Isolation Levels

Isolation Level	Dirty Reads	Non-Repeatable	Phantom
Read Uncommitted	□	□	□
Read Committed	□	□	□
Repeatable Read	□	□	□
Serializable	□	□	□

How to guarantee this?

There are mainly two approaches used in the industry □ Can you name one?

How to guarantee this?

There are mainly two approaches. □ Can you name one?

MVCC and **Locks**

Locks

A **Lock Manager** grants different kinds of locks on objects, such as **Exclusive** or **Shared** locks

Existing ➡ Requested ⤵	None	S (Shared)	X (Exclusive)
S (Shared)	□	□	□
X (Exclusive)	□	□	□

> **Drawback**: Locks limit the **concurrency**

A note on Optimistic CC

Locks □ do not have to be taken **immediately**

We can instead choose to assume that everything *generally* goes fine

1. Record the **version** of objects we are reading
2. At write time, if the destination has **not** changed `COMMIT`, otherwise, `ROLLBACK`

MVCC

Every operation operates on a **snapshot** of the database

> Reading never blocks writing and writing never blocks reading.

Tuples are not modified in **place**

> **Drawback**: outdated pages need to be **cleaned up** and txid wraparound □

Isolation Levels in PostgreSQL

- `REPEATABLE_READ` is provided by taking a snapshot at the beginning of the transaction. Concurrent updates will lead to a so-called serialization error and a `ROLLBACK`
- `SERIALIZABLE` is the same with a stronger guarantee: if a table is **read** by two concurrent transactions, one is aborted
- `READ_COMMITTED` is the default
- `READ_UNCOMMITTED` does not really exist (□ Why?)

Recap

- We looked at how PostgreSQL **stores data on disk**
- We saw that it uses a **log** named the **WAL** to ensure atomicity and durability, as well as a **cache** named the **Buffer Pool**
- Isolation is provided by **transactions** that have different isolation levels
- The default isolation level, `READ COMMITTED`is provided by the **MVCC** nature of Postgres