

# SNDBA: Module Intro

Systems and Database Administration: Jack O'Neill

# What is a Database?

We can answer this by asking what it *does*...

A database is a glorified file manager

It allows users to insert, delete, update, and search information held on a computer

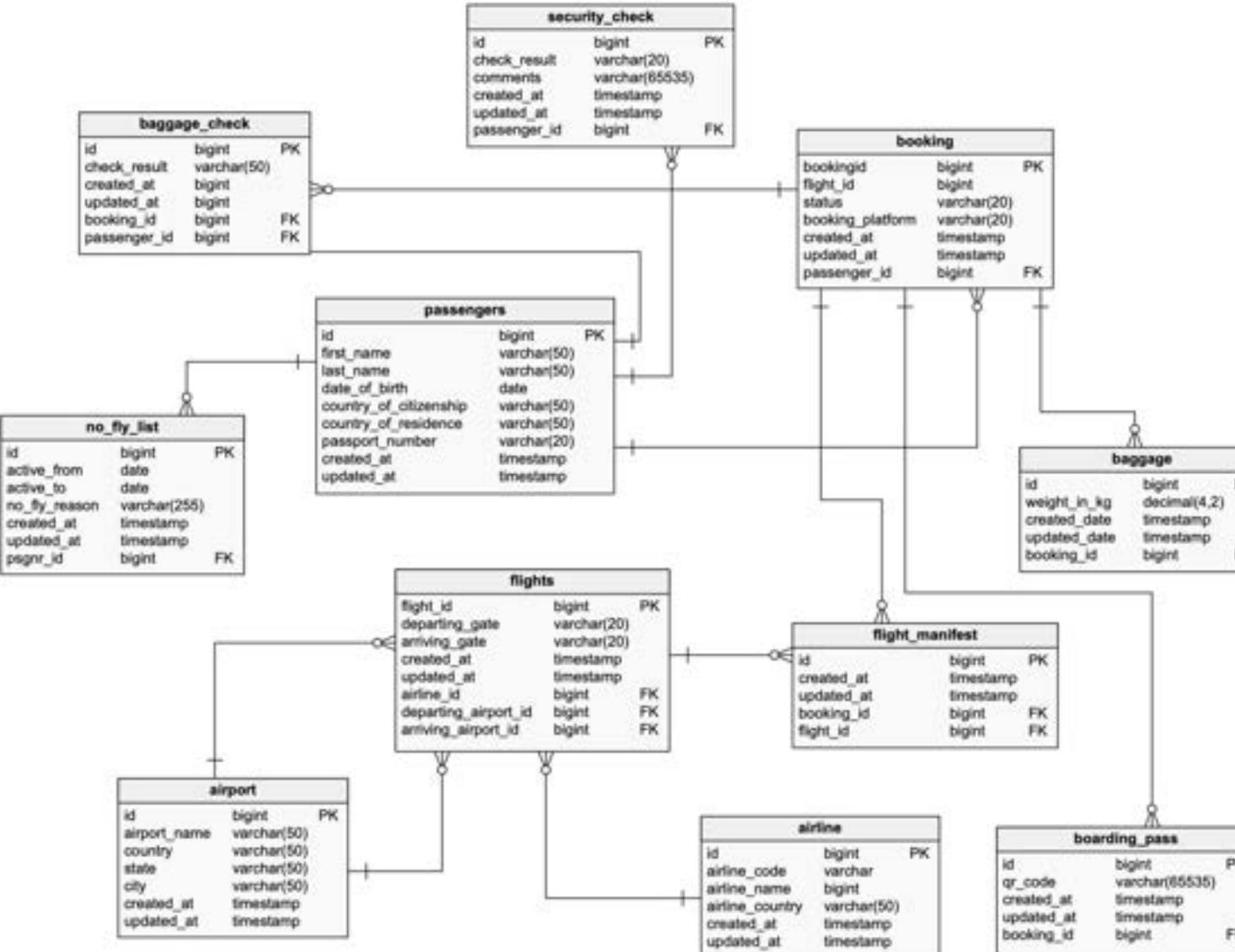
# History of Databases

Prior to 1960 computers used magnetic tape storage instead of hard disks

In the 1960s databases emerged around the same time as magnetic disk storage (HDDs)

Earliest databases, *navigational* databases worked through links, like browsing the internet without Google

The modern table-oriented *relational* database developed by E.F. Codd in 1970.  
Eventually resulted in IBM'S *System R* in 1974



# Goals of Relational Database

Physical Data Independence

Query Optimization

Authorization

Distribution Independence

Concurrency

# Physical Data Independence

Data is presented to the user logically (tables)

The user has no knowledge of how the data is stored (files on disk)

We can change the underlying storage mechanism without affecting the database

# Query Optimization

Users retrieve data from the Database using SQL

SQL uses set notation from mathematics to describe the information the user wants to see

There are many different ways to retrieve the same set of data. Some versions may run quicker than others

It should not be the user's job to find the optimal query

```
1 SELECT * FROM Departments d
2      INNER JOIN Employees e
3          ON e.deptID = d.deptID
4 WHERE e.name LIKE '%JACK%'
```

```
1 SELECT * FROM Employees e
2      INNER JOIN Departments d
3          ON e.deptID = d.deptID
4 WHERE e.name LIKE '%JACK%'
```

# Authorization (and Authentication)

Databases are intended to *share* data

Not everyone should have the same access to data

Every relational system provides tools for Authorization (restricting access based on user identity) and Authentication (ensuring a user is who they say they are)

# Distribution Independence

Becoming more and more important in recent years

Database systems can be made up of multiple computers on a network

It is the job of the system to act like only a single computer is acting as the database server

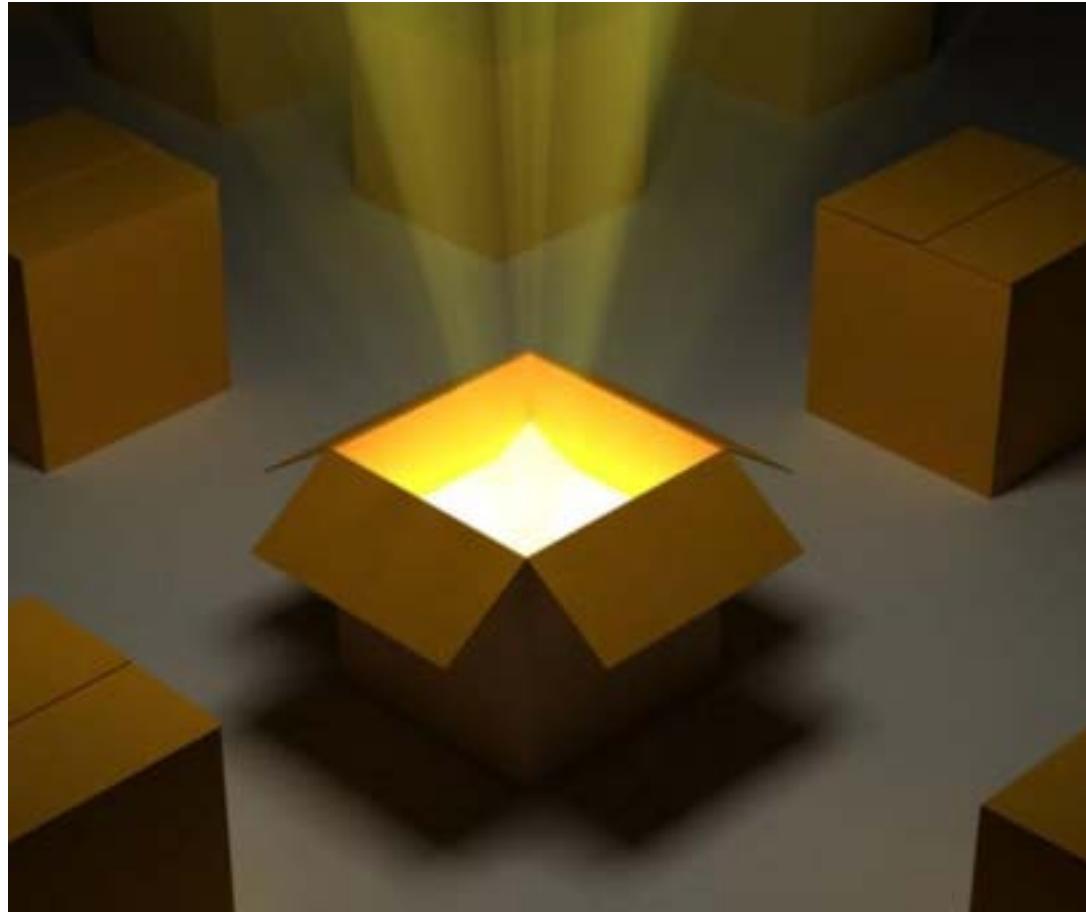
# Concurrency

Databases should allow multiple users to access data at the same time

However, they should *act as though* only a single user is on the system

Prevents users' queries from interfering with each other

# DB From a User's Point of View



- From an end user's point of view the database is a magic box
- SQL in, Data out

# DB from a DBA's Point of View

- All those concerns haven't magically gone away
- Whatever the user doesn't have to worry about is the DBA's job to make work



# Responsibilities of a DBA

Determine the system requirements both in terms of hardware and software

Install and Configure the operating system and database

Maintain operation of the database and the operating system on which it runs

Secure the database and carry out audits to prevent unauthorised access

Optimise the database through indexes, partitioning etc.

Bring the database online in case of failure, maintain a backup and recovery plan to prevent data loss

# Popular RDBMS's

Oracle SQL: Proprietary, feature-rich, expensive

Microsoft SQL: Proprietary, feature-rich, expensive

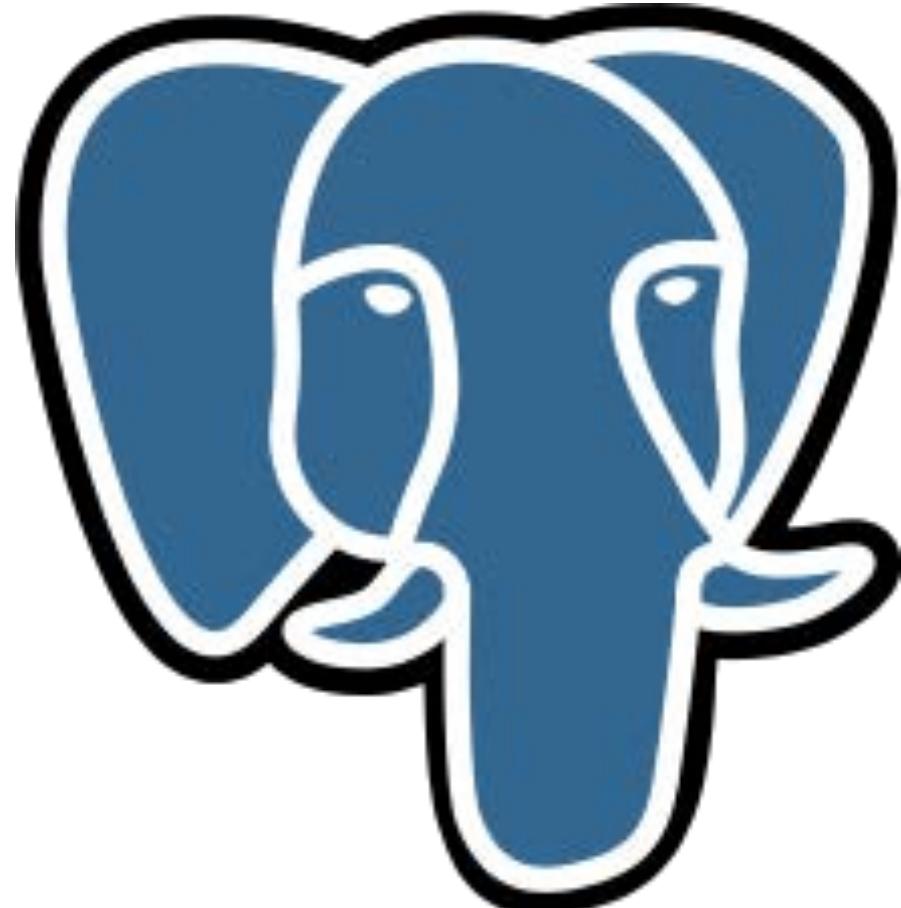
MySQL: Open-Source (kind-of), Free (kind-of)

Maria DB: Open-Source fork of MySQL

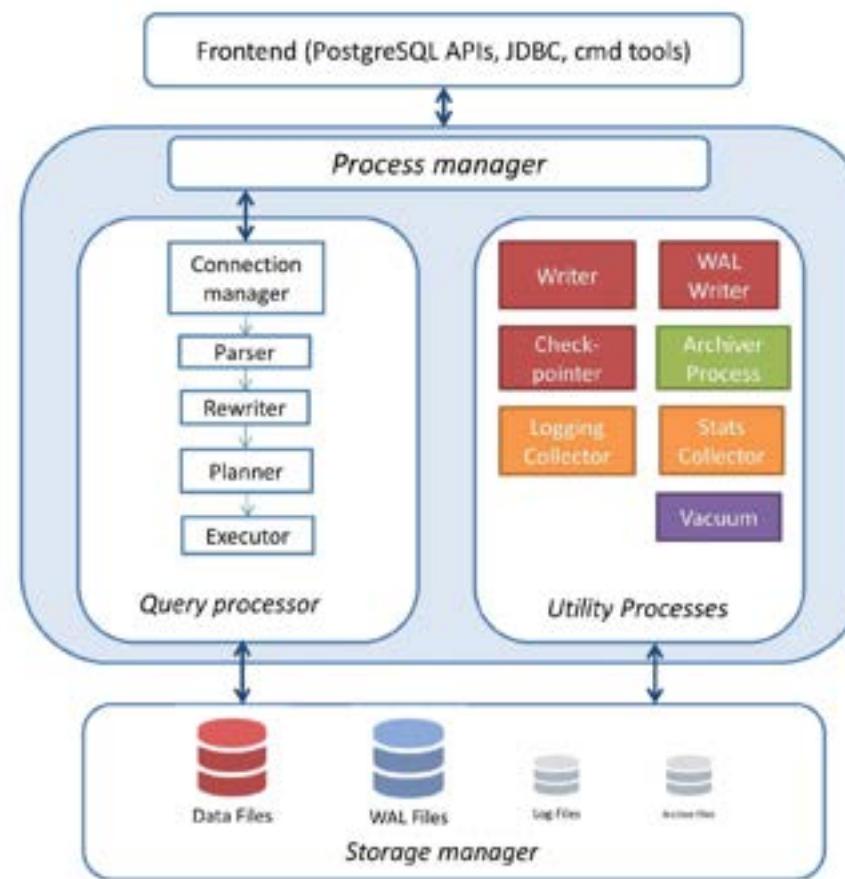
PostgreSQL: Open-source, feature-rich, free

# History of PostgreSQL

- The Ingres project was a database system developed at UC Berkeley in the 1980's
- Ingres developed into proprietary software and PostgreSQL was developed as an open source improvement over the original
- Focuses on SQL compliance, extensibility
- Most popular open source database in use today



# Database – More than just a file manager!



# Structure of the Module - Format

Lecture – 1 Hour: Theory-focussed

Lab – 2 Hours: Practical-focussed

Tutorial – 1 Hour (not every week): Support-focussed

# Structure of the Module - Topics

- Architecture
- Securing the Database
- Optimising Queries
- Backup and Recovery
- Transactions
- Distributed Databases
- Eventual Consistency

# Structure of the Module - Assessment



Exam (50%)



Assignment (25%)  
Week 7 (TBC)



Online Quiz – Short Format MCQ (15%)  
Week 11 (TBC)

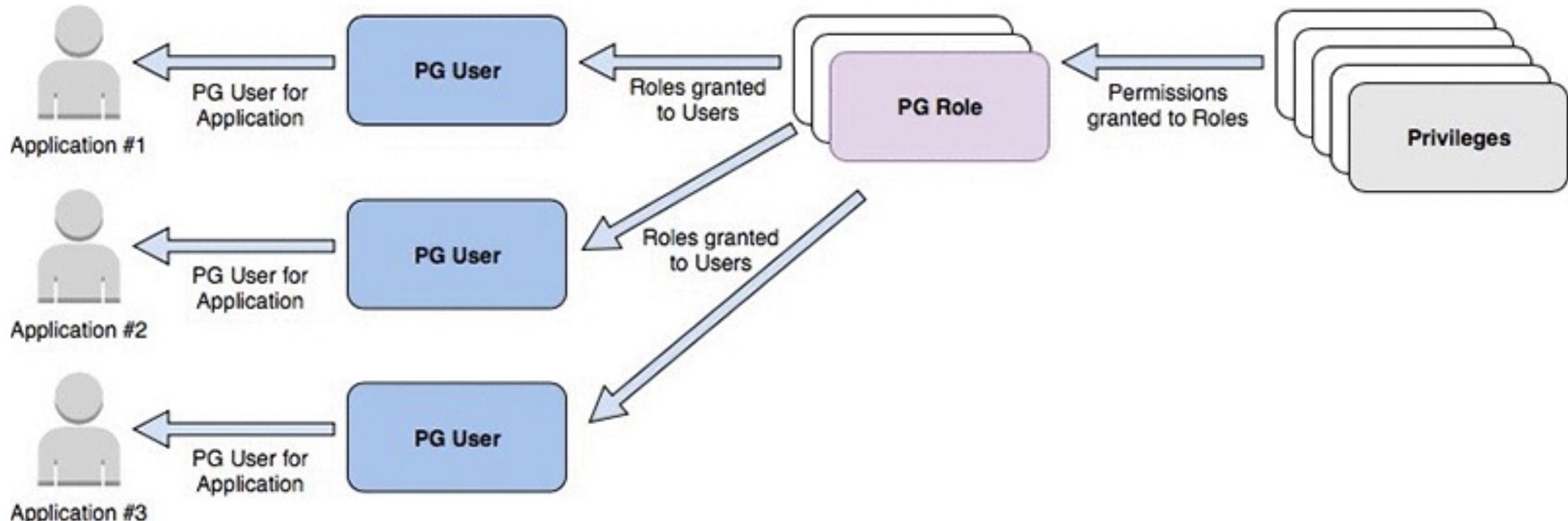


Labs (10%)  
Ongoing

# Security in the Database

SNDBA: Week 2

# Users, Roles, Privileges



# Users, Groups, Roles

## Traditional Security Architecture: Users and Groups

User = login, users can be granted privileges

Group = usually no login, can be granted privileges

Users <-> Groups = Many-to-many

Postgres *merges* users and groups into roles

# Roles in Postgres

A fresh installation of Postgres comes with a single login-enabled role with super-user privileges, `postgres`

As a super-user, should be used minimally

Connecting with the `postgres` role will allow you to manage and create additional roles

# Managing Roles

Each role name must be unique within a database cluster.

You can get a list of existing roles through the *pg\_roles system catalog*

```
SELECT rolname FROM pg_roles;
```

You can create and drop roles using SQL commands

```
CREATE ROLE <rolename>
```

```
DROP ROLE <rolename>
```

# Role Attributes

```
postgres@sndba-ubuntu:~$ /usr/local/pgsql/bin/psql
psql (14.1)
Type "help" for help.

postgres=# select * from pg_roles;
   rolname    | rolsuper | rolinherit | rolcreaterole | rolcreatedb | rolcanlogin | rolreplication | rolconnlimit | rolpassword | rolvaliduntil | rolbypassrls | rolconfig | oid
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
postgres      | t       | t       | t       | t       | t       | t       | f       | 10      | f       | f       | f       | 10
pg_database_owner | -1     | ***** | f       | t       | f       | f       | f       | 6171    | f       | f       | f       | 6171
pg_read_all_data | -1     | ***** | f       | t       | f       | f       | f       | 6181    | f       | f       | f       | 6181
pg_write_all_data | -1     | ***** | f       | t       | f       | f       | f       | 6182    | f       | f       | f       | 6182
pg_monitor      | -1     | ***** | f       | t       | f       | f       | f       | 3373    | f       | f       | f       | 3373
pg_read_all_settings | -1     | ***** | f       | t       | f       | f       | f       | 3374    | f       | f       | f       | 3374
pg_read_all_stats  | -1     | ***** | f       | t       | f       | f       | f       | 3375    | f       | f       | f       | 3375
pg_stat_scan_tables | -1     | ***** | f       | t       | f       | f       | f       | 3377    | f       | f       | f       | 3377
pg_read_server_files | -1     | ***** | f       | t       | f       | f       | f       | 4569    | f       | f       | f       | 4569
pg_write_server_files | -1     | ***** | f       | t       | f       | f       | f       | 4570    | f       | f       | f       | 4570
pg_execute_server_program | -1     | ***** | f       | t       | f       | f       | f       | 4571    | f       | f       | f       | 4571
pg_signal_backend  | -1     | ***** | f       | t       | f       | f       | f       | 4200    | f       | f       | f       | 4200
(12 rows)

postgres=#

```

# Role Attributes

Attribute	Description	Keyword
Login Privilege	Allows this role to be the initial role name for a db connection	LOGIN
Superuser Status	Superusers by-pass all security checks. Should be used sparingly	SUPERUSER
Database Creation	Needed to create databases	CREATEDB
Role Creation	Allows this role to create, alter and drop other roles (except Superuser)	CREATEROLE

Each keyword can be negated by putting NO before it,

*LOGIN/NOLOGIN, SUPERUSER/NOSUPERUSER*

**ALTER ROLE <rolename> <keyword1> <keyword2>**

# Creating a Role for a User

A user role needs to be able to connect to the database, login and supply a password to authenticate themselves

If a password is not supplied for a user all login attempts will fail

```
CREATE ROLE jack LOGIN WITH PASSWORD 'password';
```

# PostgreSQL Documentation

The PgSQL documentation contains everything you will ever need to know about installing, maintaining and working with a database

<https://www.postgresql.org/docs/14/index.html>

```
ALTER ROLE role_specification [ WITH ] option [ ... ]
```

where *option* can be:

- SUPERUSER | NOSUPERUSER
- | CREATEDB | NOCREATEDB
- | CREATEROLE | NOCREATEROLE
- | INHERIT | NOINHERIT
- | LOGIN | NOLOGIN
- | REPLICATION | NOREPLICATION
- | BYPASSRLS | NOBYPASSRLS
- | CONNECTION LIMIT *connlimit*
- | [ ENCRYPTED ] PASSWORD '*password*' | PASSWORD NULL
- | VALID UNTIL '*timestamp*'

```
ALTER ROLE name RENAME TO new_name
```

```
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_parameter FROM CURRENT
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] RESET configuration_parameter
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL
```

where *role\_specification* can be:

- role\_name*
- | CURRENT\_ROLE
- | CURRENT\_USER
- | SESSION\_USER

# Group Roles

**Group roles:** no real distinction but used informally

Group roles usually don't have a login, but are *granted* to other roles

```
CREATE ROLE jack LOGIN;  
  
CREATE ROLE administrators NOLOGIN CREATEDB CREATEROLE;  
  
GRANT administrators TO jack;
```

# Role Inheritance

Roles can be specified with the INHERIT or NOINHERIT keyword

This controls what happens when the role is granted membership of a *group role*.

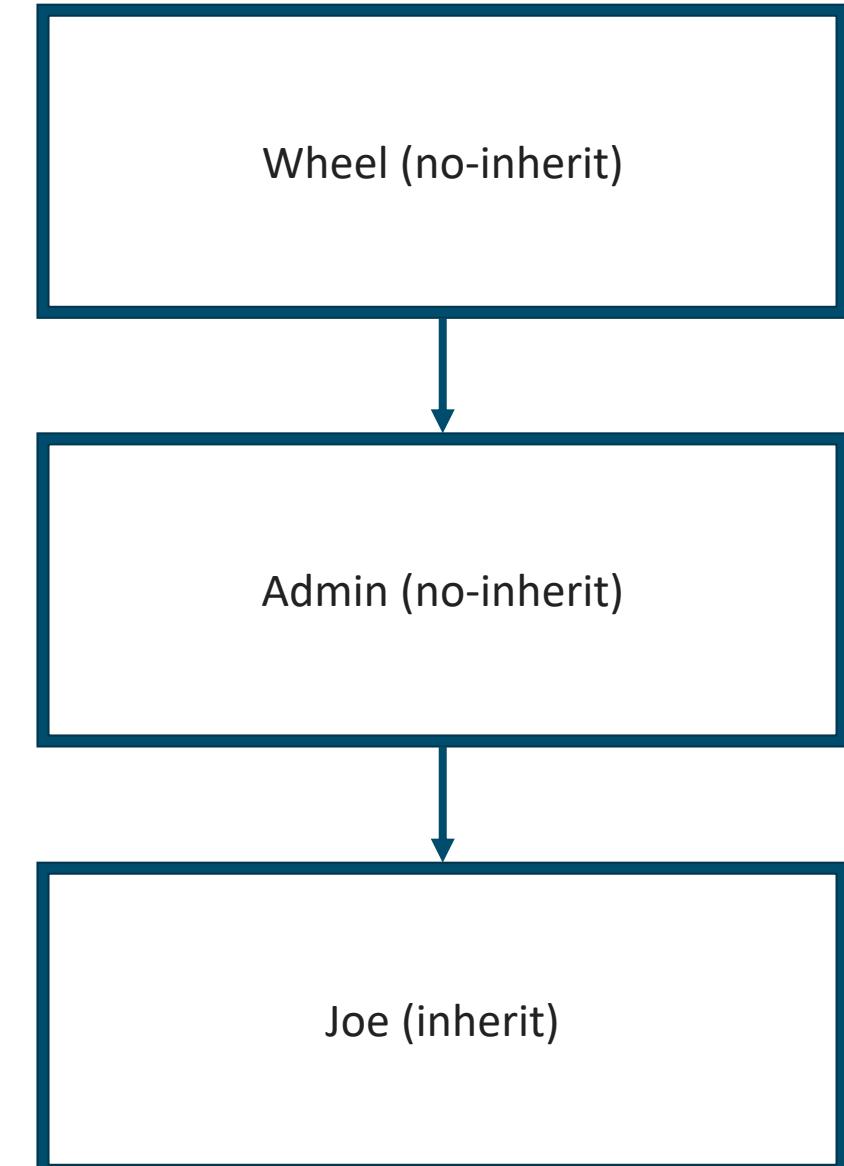
Without INHERIT, a role must explicitly "become" the granted role to use it (SET ROLE command)

With INHERIT, the role directly inherits all privileges of the granted role and doesn't need to explicitly invoke the role.

# Inheritance in Action

```
CREATE ROLE joe LOGIN INHERIT;  
CREATE ROLE admin NOINHERIT;  
CREATE ROLE wheel NOINHERIT;  
GRANT admin TO joe;  
GRANT wheel TO admin;
```

Which privileges has Joe automatically inherited?



# Dropping Roles

What's gone wrong?

```
postgres=# DROP ROLE jack;
2022-02-02 11:40:15.831 UTC [1798] ERROR:  role "jack" cannot be dropped because some objects depend
on it
2022-02-02 11:40:15.831 UTC [1798] DETAIL:  owner of table testtable
2022-02-02 11:40:15.831 UTC [1798] STATEMENT:  DROP ROLE jack;
ERROR:  role "jack" cannot be dropped because some objects depend on it
DETAIL:  owner of table testtable
postgres=#
```

# Handling Dependencies when Dropping Roles

A role may not be dropped while it has ownership of any database objects

Objects ownership may be manually reassigned one step at a time

```
ALTER TABLE testtable OWNER TO postgres;
```

# REASSIGN OWNED

If a role owns many objects, manually reassigning can be time-consuming

PgSQL lets us transfer ownership of all objects using the REASSIGN OWNED command

After reassigning owned we need to drop owned to take care of privileges granted on objects which do not belong to the role (order is very important!)

```
REASSIGN OWNED BY
jack TO postgres;
DROP OWNED BY jack;
DROP ROLE jack;
```

# Pre-Defined Roles

pg_read_all_data	Read all data (tables, views, sequences), as if having SELECT rights on those objects, and USAGE rights on all schemas, even without having it explicitly. This role does not have the role attribute BYPASSRLS set. If RLS is being used, an administrator may wish to set BYPASSRLS on roles which this role is GRANTed to.
pg_write_all_data	Write all data (tables, views, sequences), as if having INSERT, UPDATE, and DELETE rights on those objects, and USAGE rights on all schemas, even without having it explicitly. This role does not have the role attribute BYPASSRLS set. If RLS is being used, an administrator may wish to set BYPASSRLS on roles which this role is GRANTed to.

# Pre-Defined Roles

pg_read_all_settings	Read all configuration variables, even those normally visible only to superusers.
pg_read_all_stats	Read all pg_stat_* views and use various statistics related extensions, even those normally visible only to superusers.
pg_stat_scan_tables	Execute monitoring functions that may take ACCESS SHARE locks on tables, potentially for a long time.
pg_monitor	Read/execute various monitoring views and functions. This role is a member of pg_read_all_settings, pg_read_all_stats and pg_stat_scan_tables.

# Pre-Defined Roles

pg_database_owner	None. Membership consists, implicitly, of the current database owner.
pg_signal_backend	Signal another backend to cancel a query or terminate its session.
pg_read_server_files	Allow reading files from any location the database can access on the server with COPY and other file-access functions.
pg_write_server_files	Allow writing to files in any location the database can access on the server with COPY and other file-access functions.
pg_execute_server_program	Allow executing programs on the database server as the user the database runs as with COPY and other functions which allow executing a server-side program.

# Databases, Schemas and Objects

Security in the Database

# Database Objects

A database consists of **objects**: tables, views, schemas, functions *etc.*

Each database object has an **owner**: (default = creator)

An object owner can grant **privileges** on an object: SELECT, UPDATE, DELETE, EXECUTE

Privileges granted WITH GRANT OPTION allow the recipient to pass these privileges onto others

# Schemas and Tablespaces

Schemas provide a *logical separation* of database objects

Each table, view, function etc. belongs to a database schema

Tablespaces provide a *physical separation* of objects

Tablespaces correspond to files in the operating system which hold the data in tables and indexes

By default, new database objects use the *pg\_default* tablespace

# Schemas and Tables

The **fully qualified name** for an object includes both the database and schema in which the object resides `<database>.<schema>.<table>`

Each database has a **search\_path** which gives a list of schemas to be checked if the fully qualified name is not provided ("default" schemas)

The search\_path can be altered using the ALTER DATABASE command

```
ALTER DATABASE <dbname> SET search_path TO schema1, schema2,  
schema3;
```

# Minimum Privileges

In order to connect to a database, a role needs the CONNECT privilege

`GRANT CONNECT ON <database> TO <role>`

In order to use a schema, a role needs the USAGE privilege

`GRANT USAGE ON <schema> TO <role>`

# The CREATE Privilege

The CREATE privilege allows users to create new objects in a database

Granting CREATE on a database allows users to create new schemas

Granting CREATE on a schema allows users to create new db objects  
(provided they have USAGE)

Granting CREATE on a tablespace allows users to create tables, indexes  
and temporary files within the tablespace

# Object-Level Privileges

Each object within a database has its own privileges

Tables require the SELECT, UPDATE, INSERT, DELETE privileges for full access

SELECT privilege is required to make proper use of UPDATE and DELETE

The TRIGGER privilege allows users to write custom code in response to an insert, update or delete on a table.

The ALL keyword allows all privileges for an object to be deleted at once

# Cascading Revoke

We saw that a privilege can be granted with GRANT OPTION

This allows the grantee to pass that privilege on to other roles

Alice -> USAGE on cheshire to BOB with GRANT OPTION

Bob's privilege is now dependent on Alice's

We cannot revoke Alice's privilege while Bob's exists in the database

# Cascading Revoke

```
REVOKE <privilege> ON <object> FROM <role> CASCADE | RESTRICT
```

The **CASCADE** keyword automatically revokes all privileges granted with GRANT OPTION

The **RESTRICT** keyword produces an error message if any such privileges exist

# Additional Security Considerations

Security in the Database

# Password Hashing

Older versions of Postgres used the md5 hashing algorithm

Md5 is insecure, it is possible for hackers to identify passwords hashed using this algorithm

Currently Postgres uses SCRAM (Salted Challenge Response Authentication Mechanism) by default.

Postgres maintains MD5 for backward compatibility but this should not be used.

# Encryption Options

Password Encryption

Column-Specific Encryption

Data Partition Encryption

Encrypting across the Network

Two-Way Encryption

Client-Side Encryption

# Password Policy

An insecure password presents a vulnerability to the entire database

Password policies should be used to enforce strict passwords

The passwordcheck module can be used to write custom rules to determine if a password is secure enough

<https://www.postgresql.org/docs/current/passwordcheck.html>

An expiration date can be set for a password using the ALTER ROLE command

# Listen Addresses

Database servers typically have multiple IP addresses

Public IP addresses can be reached by anyone who knows the IP address of the server

We can configure PostgreSQL to only accept connections from certain IP addresses

By default, the database will not accept any connections over the network

# Principle of Least Privilege

Good database design obeys the **Principle of Least Privilege (PoLP)**

PoLP states that each user should have the minimum privileges necessary to do the job they need to do.

PoLP limits the impact of a security breach on the entire database

# Triggers, Auditing, Logging

To ensure that Information Security is being properly enforced

# Principles of Information Security

- Information security is the confidentiality, integrity and availability of information.
- **Confidentiality** prevents unauthorized disclosure of information
- **Integrity** protects information from unintentional / unauthorized alteration, modification or deletion.
- **Availability** information is readily accessible to authorized users.

# Controls

- Identification – scope, locality and uniqueness.
- Authentication (data entry / device / biometrics)
- Authorization (Principle of Least Privilege (PoPL)).
  
- Accountability and access control
  - Logs and Audit trails provide this.

# Auditing: What?

- An audit is an inspection.
- It shows us what has happened in the system.
- IT audit
  - Examines controls within an IT group

# Auditing: When and where?

- Once-off or ongoing?
  - Audit times and locations should be chosen in accordance with the objectives of the audit.



# Auditing: Who?

- Audit should be run by an independent auditor
- But does require help from:
  - System administrator
  - Applications administrators
    - E.g. DBA

# Audting: How?



# What does it provide?

- Should provide user (all levels) accountability
- Data leakage protection strategy
  - lost / stolen devices?
  - Separation of personal / corporate data?
- Ability to trace actions
  - Remote data retrieval
  - Data sharing
  - Data access (by whom?)
  - Trend analysis
  - Regulatory compliance

# What Is Auditing?

- Monitoring and recording of database actions
  - Individual actions
  - Combinations of actions and user info
  - Successful and failed activities

# Auditing Windows Systems

- Set up policies (GPOs)
- Look at logs:
  - System logs
  - Security logs
  - Application logs

# Logging vs Auditing

- Log files are key to the auditing process
- Log files can tell us what was done, by whom and when
- Logs are useful outside of auditing too (performance tuning, debugging)

# In Postgres

- Automatic auditing can be enabled (only stderror is logged by default).
- Easiest solution is to log all SQL statements (set in postgresql.conf)
- More fine-grained control available through the pgAudit extension
- Database triggers for fully customizable auditing

# Configuring auditing

- Can be configured in postgresql.conf
- Can also view and edit parameters using **show** and **set**.
- The logging\_collector parameter automatically rotates log files
- Doesn't record user by default! Use **log\_line\_prefix**.
- View pgAudit settings with
- **SELECT name,setting FROM pg\_settings WHERE name LIKE 'pgaudit%';**
- Can be viewed and edited using **show** and **set**.

# Difficulties with Auditing

- Why aren't audit files kept in a database table?
- What considerations do we need to take into account when choosing a log-file location?

# PgAudit Types

**Session auditing:** Audit all actions taken by any users

**Object auditing:** Audit only actions affecting certain database objects

# Event auditing

- Uses database triggers.
- Triggers can be run, before, after or instead of a SQL statement
- Can abort statement if desired
- Can be written in many languages, full power to do what you like

# Triggers

- A trigger can insert into data in separate tables
- It can also modify the values being inserted
- We can use INSTEAD OF triggers to make views "updateable"

When	Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
AFTER	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	—
	TRUNCATE	—	—

# Basic Auditing with Triggers

```
1  CREATE TABLE emp (
2      empname text,
3      salary integer,
4      last_date timestamp,
5      last_user text
6  );
7
8  CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
9      BEGIN
10          — Check that empname and salary are given
11          IF NEW.empname IS NULL THEN
12              RAISE EXCEPTION 'empname cannot be null';
13          END IF;
14          IF NEW.salary IS NULL THEN
15              RAISE EXCEPTION '% cannot have null salary', NEW.empname;
16          END IF;
17
18          — Who works for us when they must pay for it?
19          IF NEW.salary < 0 THEN
20              RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
21          END IF;
22
23          — Remember who changed the payroll when
24          NEW.last_date := current_timestamp;
25          NEW.last_user := current_user;
26          RETURN NEW;
27      END;
28  $emp_stamp$ LANGUAGE plpgsql;
29
30  CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
31      FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

# Trigger Variables

- Triggers have access to a bunch of specially defined variables providing context for the target operation
- Most popular variables are OLD and NEW, representing the old and new values before and after an update (insert/delete)
- TG\_TABLE\_NAME gives the name of the table targeted by the trigger.
- TG\_OP tells you what operation is being performed: insert, update, delete

# BEFORE Triggers

- BEFORE triggers are run before a statement is run on the target object
- BEFORE triggers can raise an exception, preventing the operation and alerting the user to the error
- BEFORE triggers can silently modify values by changing the properties of the NEW object
- BEFORE triggers can silently ignore an operation by returning NULL
- A single statement may trigger multiple BEFORE triggers

# INSTEAD OF Triggers

- INSTEAD OF triggers are row-level triggers which can be applied to a view
- When a user tries to update, insert or delete from the view, the INSTEAD OF trigger will be fired instead
- For an insert/update, the trigger performs the necessary operations on tables and returns the values of NEW
- For a delete, the trigger performs the necessary operations on tables and returns the values of OLD
- A return-value of NULL tells the user that no data was changed.
- Each object can only have one INSTEAD OF trigger

# AFTER Triggers

- AFTER triggers are fired after a statement has been completed
- AFTER triggers may not modify the values
- AFTER triggers can abort the operation by raising an exception
- Multiple AFTER triggers can be set off by a single statement.

# The Audit-Trigger Template

- The authors behind pgaudit have shared a template trigger which you can use for trigger-based auditing
- This SQL script looks after creating and populating audit tables, it's fully customizable
- <https://github.com/2ndQuadrant/audit-trigger/blob/master/audit.sql>

# Downside of logging/auditing

- All auditing increases the workload on the database system.
- Trigger-based auditing adds to the complexity and can introduce bugs which prevent normal operation of the system
- Too much auditing can make the output files unnecessarily large and difficult to parse

# Recap

- **Audit:** An inspection which shows us what has happened in a system
- **Logging:** Logging is an important tool to enable us to conduct effective audits
- **Tools:** Out-of-the-box, PgAudit, Trigger-based
- **Triggers:** Can be used to automate the database, also useful for auditing, the audit-trigger project is a great starting point for trigger-based auditing

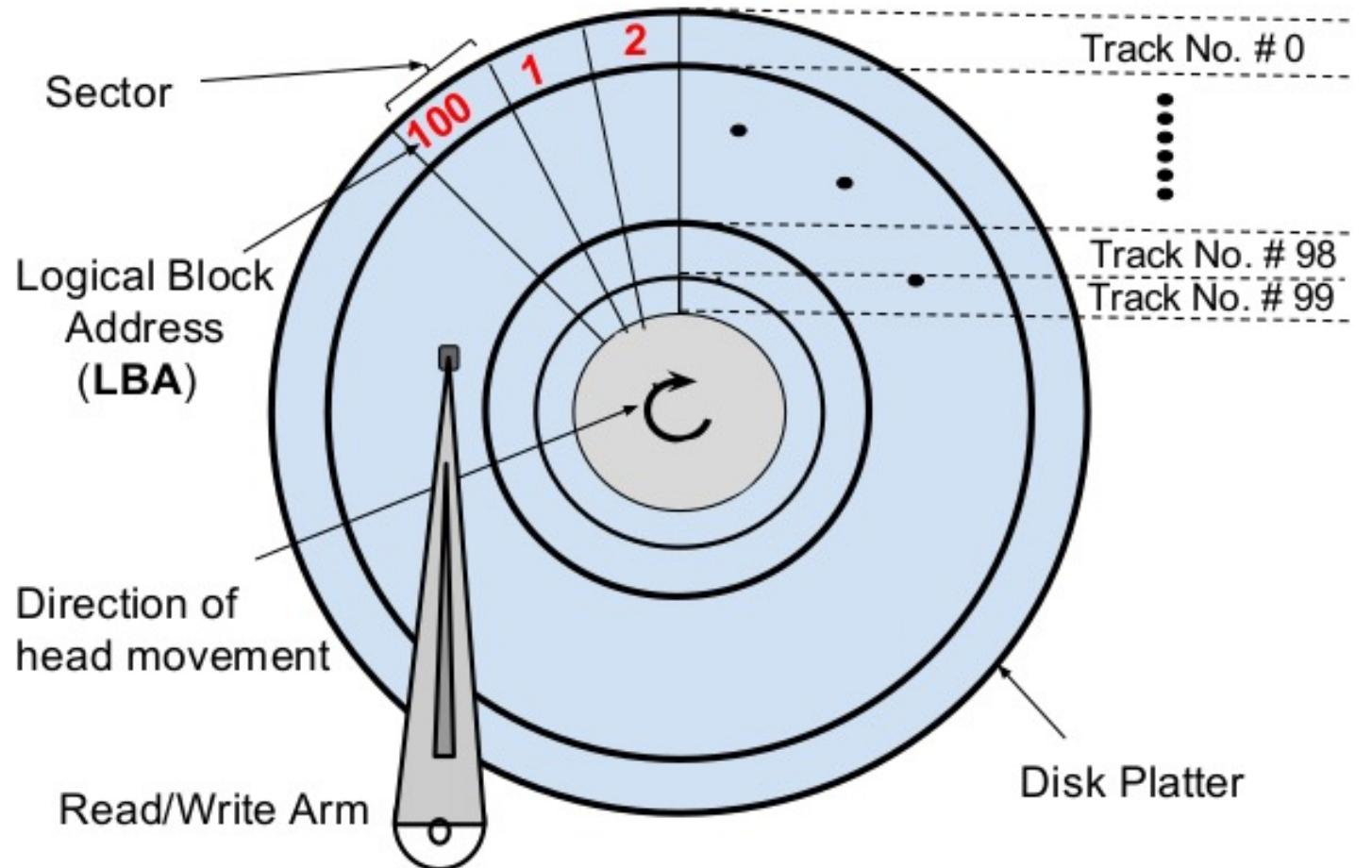


# Query Optimization

---

Physical Filesystems and  
Indexing

# Anatomy of a Hard Disk



# HDD Data Access is Slow

---



HARD-DRIVES CONTAIN  
MOVING PARTS



DATA MUST BE ACCESSED  
SEQUENTIALLY



GENERALLY OPTIMISED FOR  
CAPACITY MORE THAN  
SPEED

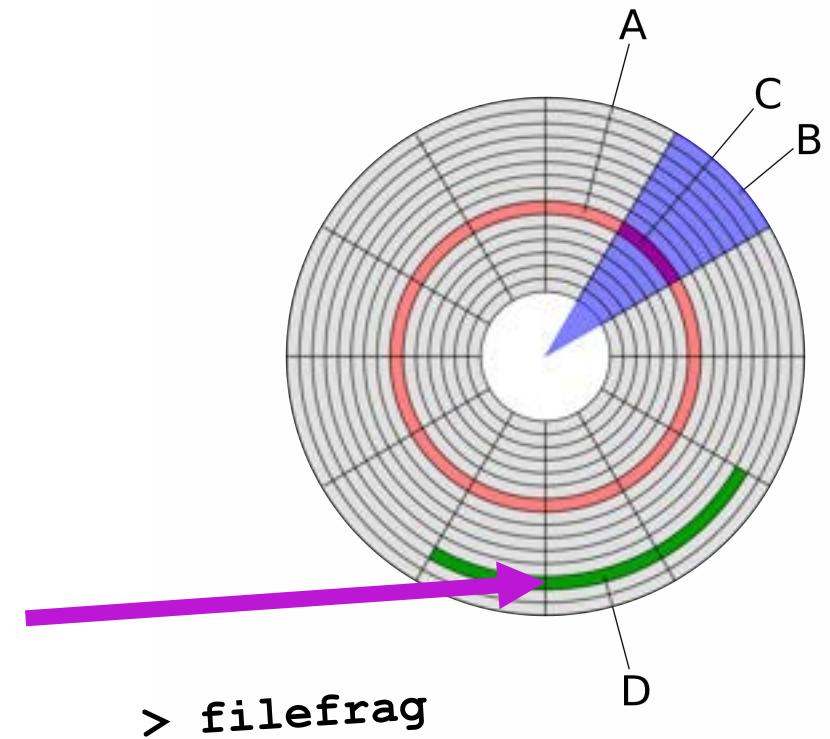
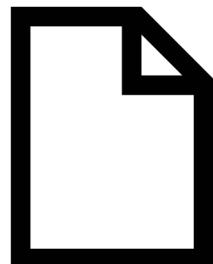
# But we Need HDDs (for now)

- Solid state drives are orders of magnitude faster than HDDs
- Extensions to PgSQL even allow you to use RAM for table storage
- But the *price-per-byte* of storage makes them impractical for many databases
- We're stuck with HDDs for now so performance will be an issue

# Data Storage: DB and OS

student_id	first_name	last_name	programme
C12345678	Alice	Doe	TU857
C87654321	Bob	Dole	TU858
C13799264	Christopher	Nolan	TU857
C02446911	Joe	Dolan	TU856

`SELECT pg_relation_filepath('tablename');`



`> filefrag`

# Database Query Pseudo-Code

Begin with first record in file

CREATE Linked List FOUND\_RECORDS

DO UNTIL no\_more\_records:

    READ first\_name, last\_name, programme

    IF programme == 'TU857' THEN

        ADD record to FOUND\_RECORDS

    END IF

DONE

```
SELECT first_name, last_name  
FROM students  
WHERE programme='TU857';
```

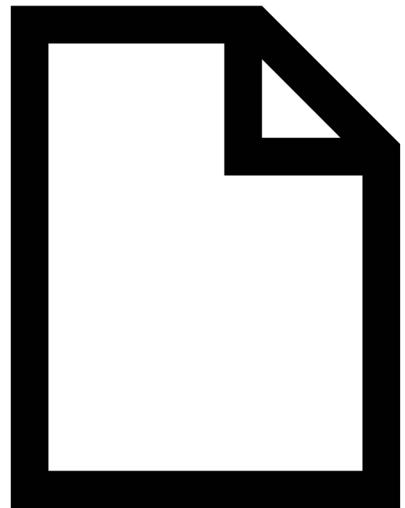
# How Can We Optimize This?

- If we have 200 words from disk
- If only 20 words reduce the number

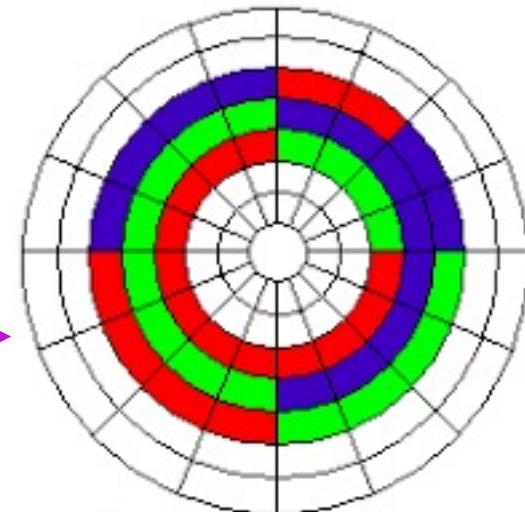
## INDEX

ABC, 164, 321n  
academic journals, 262, 280–82  
Adobe eBook Reader, 148–53  
advertising, 36, 45–46, 127, 145–46, 167–68, 321n  
Africa, medications for HIV patients in, 257–61  
Agee, Michael, 223–24, 225  
agricultural patents, 313n  
Aibo robotic dog, 153–55, 156, 157, 160  
AIDS medications, 257–60  
air traffic, land ownership vs., 1–3  
Akerlof, George, 232  
Alben, Alex, 100–104, 105, 198–99, 295, 317n  
alcohol prohibition, 200  
*Alice's Adventures in Wonderland* (Carroll), 152–53  
Anello, Douglas, 60  
animated cartoons, 21–24  
antiretroviral drugs, 257–61  
Apple Corporation, 203, 264, 302  
architecture, constraint effected through, 122, 123, 124, 318n  
archive.org, 112  
*see also* Internet Archive  
archives, digital, 108–15, 173, 222, 226–27  
Aristotle, 150  
Armstrong, Edwin Howard, 3–6, 184, 196  
Arrow, Kenneth, 232  
art, underground, 186  
artists:  
    publicity rights on images of, 317n  
    recording industry payments to, 52, 58–59, 74, 195, 196–97, 199, 301, 329n–30n

# Mapping to Physical Locations



Logical

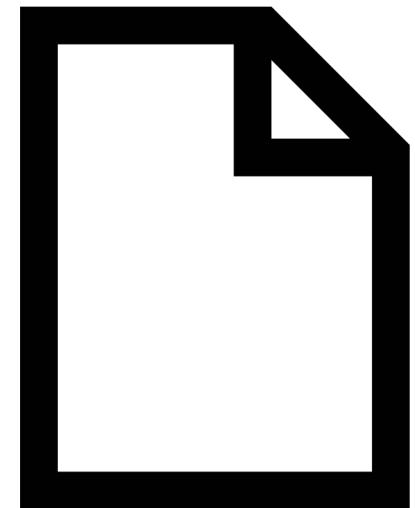


File A   File D  
File B  
File c

Physical

# Mapping from Table to Filesystem

Offset	student_id	first_name	last_name	programme
0	C12345678	Alice	Doe	TU857
26	C87654321	Bob	Dole	TU858
51	C13799264	Christopher	Nolan	TU857
86	C02446911	Joe	Dolan	TU856



# So What Does a DB Index Look Like?

Index Column Value	File Offset
TU856	86
TU857	0
TU857	51
TU858	26

A database index usually holds a single column value

As well as storing the value, it stores an offset allowing the database to directly access the record from file

This means smaller reads (small improvement)

Indexes are usually ordered meaning most of the reads can be skipped entirely

**It's not really a table!**

# The Read/Write Trade-off

- An index makes certain read operations significantly faster
- However, the database now has to maintain an additional set of data
- Indexes can make write operations significantly slower as index maintenance is carried out for each insert or update

# Types of Database Index

- Most databases support 2 types of index
- Hash indexing
- B-Tree indexing

# Hash Indexing

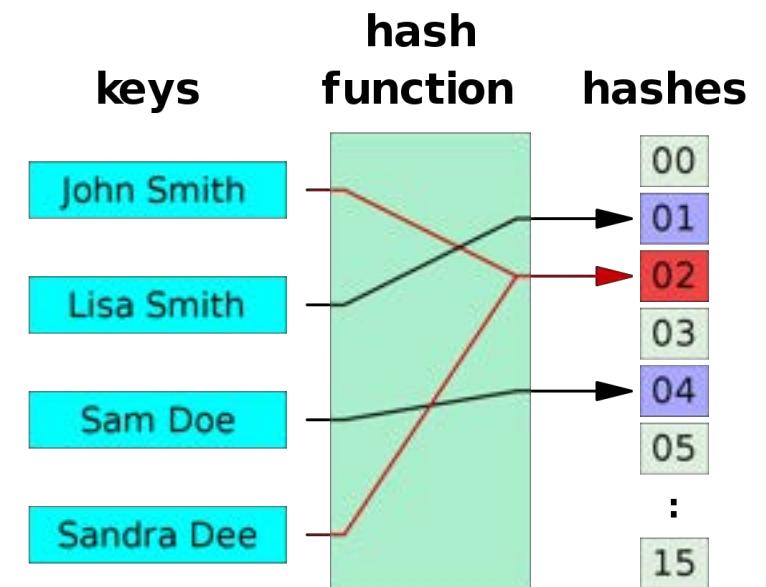
---

- Hash functions get their name from the phrase to "make a hash of something", i.e. to mess it up.
- Hash functions do lots of weird operations on their inputs to turn each one into a numeric output
- Hash functions are cheap and quick to run
- We can use this numeric output as a location at which to store our data



# How Hash Functions Work

- Hash functions look random, but the same input always produces the same output
- I store my index as an array with, say, 100 elements
- When I add John Smith to the index, I pass it to the hash function and store it in the slot putted by the hash function
- If I want to check for Joe Smith in the index I can put I can run it through the hash function and see immediately where it would be



# Hash Collisions

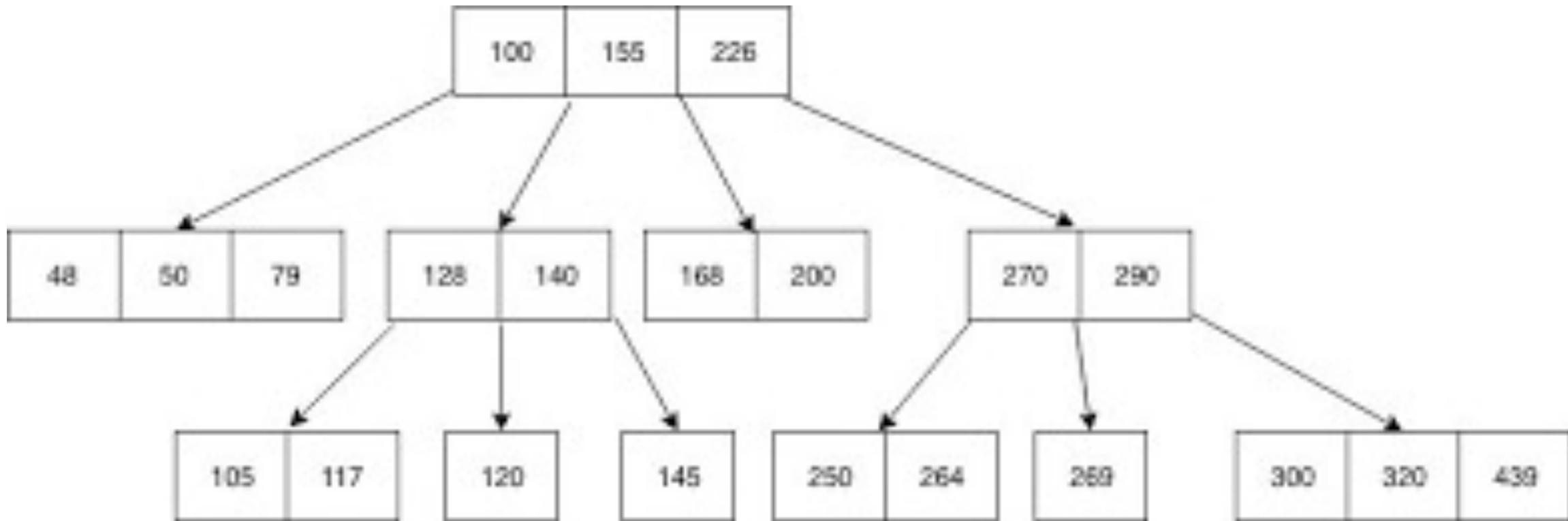
- Usually a hashing function can output any integer (not just 0 -99)
- If I want to reduce this to 100 buckets I can divide the number by 100 and use the remainder (modulo)
- If we have more data than buckets we end up with *collisions*
- We use **Linked Lists** to allow us to store multiple items in a bucket

# B-Tree Indexing

---

- B-Tree's get their name from the fact that they look like a tree.
- The B in B-Tree refers to the fact that this data-structure is self-balancing
- B-Trees are an excellent way to store and quickly retrieve sorted data





## How a B-Tree Works

- A B-Tree begins with a root node containing a certain number of sorted values
- Each node has a certain number of children
- The values in each node act as separators for the children

# B-Tree Rules

1. Every node has at most  $m$  children.
2. Every non-leaf node (except root) has at least  $\lceil m/2 \rceil$  child nodes.
3. Every non-leaf node has at least two children.
4. A non-leaf node with  $k$  children contains  $k - 1$  keys.
5. All leaves appear in the same level and carry no information.

# Self-Balancing

- Whenever new data is entered, there is a chance that it will *overfill* a node.
- When this happens, the node is split up and multiple child nodes are created.
- This ensures that B-Trees remain relatively optimized and never grow too deep

# Which Index to Use

## Hash Index

- Easier to maintain
- Only suitable for equality operations

## B-Tree

- Requires more maintenance
- Suitable for any operation making use of sorted information

# Creating an Index

- Indexes can be called anything you like
- Usually it makes sense to name them so they are identifiable
- `ix_<tablename>_<columnname>` is a good choice
- The `USING` keyword lets us specify hash or btree (and others), in pgsql btree is default

```
CREATE INDEX ix_students_student_id  
ON students USING hash;
```

# Fill-Factor

- The FILLFACTOR parameter allows the DBA to specify how much the database will attempt to compress the index
- FILLFACTOR is expressed as a percentage between 1 and 100
- A larger number means the index takes up less space in disk
- A larger number also means that it will be more difficult to update the index, as more rebuilding will be required
- The default value is 90

# Building Indexes CONCURRENTLY

---

- Building an index can take a long time on an existing table
- Usually, the table is locked for writing while the index is built
- It is possible to build an index in a two-step process using the CONCURRENTLY keyword
- This is useful on a production system where table downtime is not an option
- This brings a lot of overhead and should only be used if needed

# Index Considerations

- Indexes speed up queries
- An index should be created on the column used to filter a query
- Indexes should be used sparingly, they slow down writes

# Identifying Slow Queries

---

- We can identify slow-running queries in 3 ways
  1. The Slow Query Log
  2. The EXPLAIN tool
  3. The pg\_stat\_statements tool

# Slow Query Logging

- The postgresql.conf file allows us to set query execution time as a trigger for logging queries
- The **log\_min\_duration\_statement** setting determines the minimum time a query must run for in order to be logged
- This can then be queried in the log file and the slow query can be easily identified

# EXPLAIN Analyze

- All SQL implementations provide their own query optimiser which determines how best to execute a query
- The EXPLAIN command breaks down an SQL query and estimates how long each step will take
- This is an excellent way to identify columns which may need to be indexed
- EXPLAIN gives an estimate, EXPLAIN ANALYZE actually executes the query

QUERY PLAN
▶ Sort (cost=169.51..172.01 rows=1000 width=87)
Sort Key: f.title
-> Hash Join (cost=41.93..119.68 rows=1000 width=87)
Hash Cond: (f.film_id = fc.film_id)
-> Seq Scan on film f (cost=0.00..64.00 rows=1000 width=19)
-> Hash (cost=29.43..29.43 rows=1000 width=70)
-> Hash Join (cost=1.36..29.43 rows=1000 width=70)
Hash Cond: (fc.category_id = c.category_id)
-> Seq Scan on film_category fc (cost=0.00..16.00 rows=1000 width=4)
-> Hash (cost=1.16..1.16 rows=16 width=72)
-> Seq Scan on category c (cost=0.00..1.16 rows=16 width=72)

# The **pgstat\_statements** Extension

- The pgstat\_statements extension provides a view of all SQL statements executed in a database
- Millions of small queries may take longer than one long-running one.
- Aggregate data makes it easier to spot tables and columns which need optimisation
- Needs to be enabled by adding pg\_stat\_statements to the **shared\_preload\_libraries** config parameter

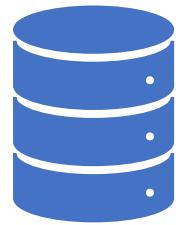
A perspective view of a server room aisle. On both sides, there are rows of server racks, each with numerous small blue lights indicating active components. The floor is a polished concrete or tile surface. The ceiling is white with integrated lighting and some structural elements. In the distance, at the end of the aisle, there is a set of double doors. The overall atmosphere is cool and technical.

# Partitions and Tablespaces

# Last Week



Last week we saw how indexes could be used to speed up queries



Indexes point the database to records of interest: no full scans



Very useful for filtering data, but still require lots of random access reads

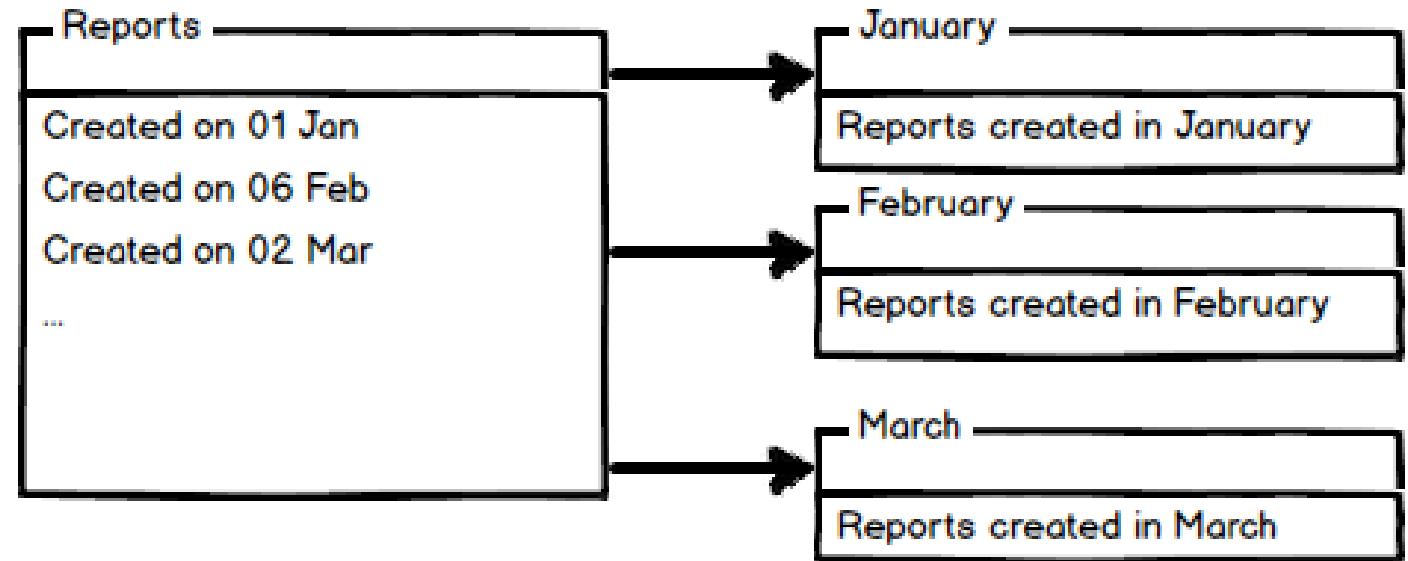
# Partitions

---

When Indexing isn't enough

# Partitions

- Partitions are used to break a table into multiple sub-tables
- They can improve performance dramatically
- Partitions make it easier to manage and archive data
- Seldom-Accessed data can be transferred to cheaper, slower, disks



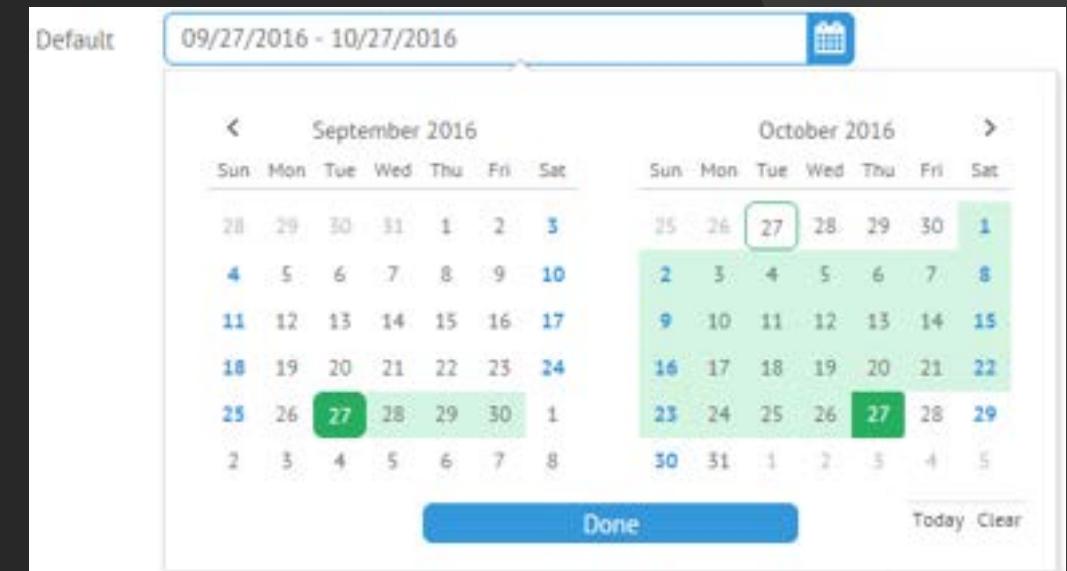
```
1 CREATE TABLE public.payment (
2     payment_id integer DEFAULT
3         nextval('public.payment_payment_id_seq' :: regclass) NOT NULL,
4     customer_id integer NOT NULL,
5     staff_id integer NOT NULL,
6     rental_id integer NOT NULL,
7     amount numeric(5,2) NOT NULL,
8     payment_date timestamp with time zone NOT NULL
9 )
10 PARTITION BY RANGE (payment_date);
11
```

## Anatomy of a Partitioned Table

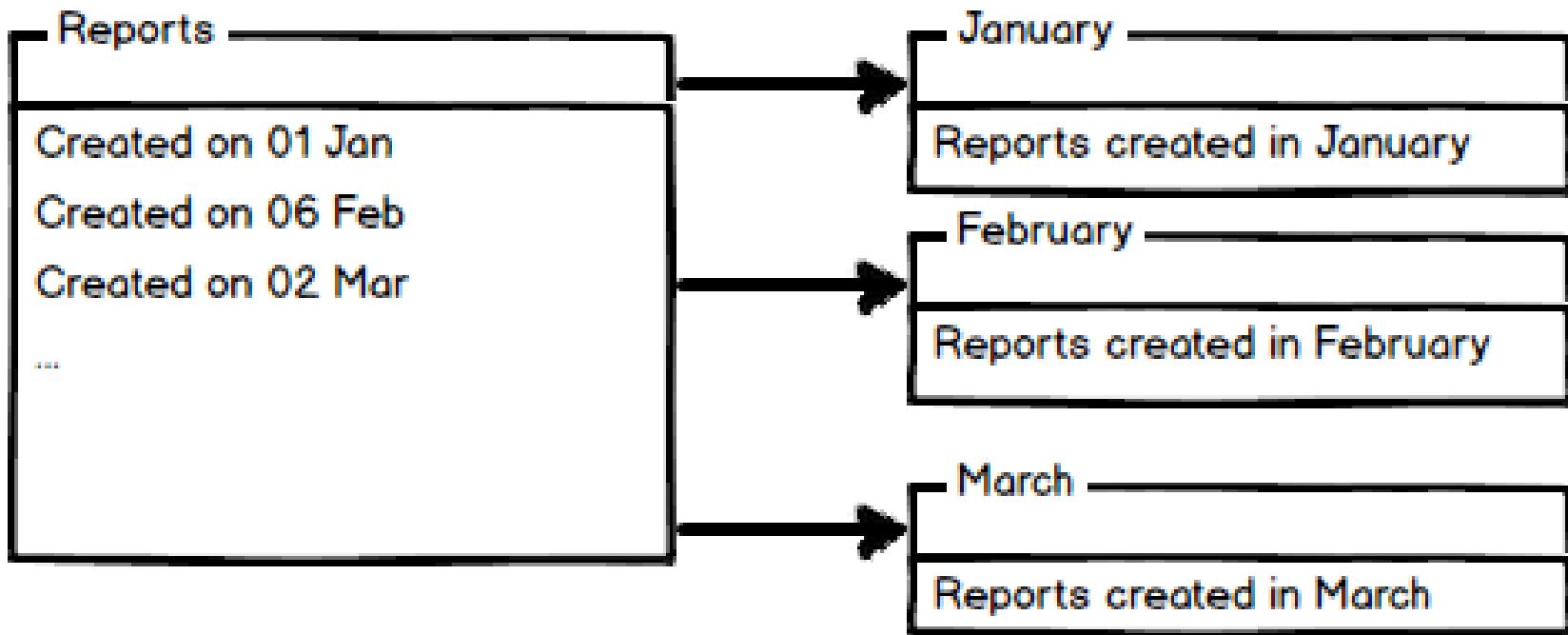
- Each partitioned table consists of 2 parts
  1. A partitioned table (the container)
  2. One or more partitions (the tables holding the actual data)
- Users can query the partitioned table as if it were a single entity
- Partitions can be added to or removed from a table quite simply

# Partition Types: List and Range

- Partitioning lets you break a table based on some value in each row
- Postgres supports 2 types of partitioning: list and range
- **List partitioning** uses a different partition for each value (or combination of values) in the partitioning columns
- **Range partitioning** uses a different partition for each range band (defined by the dba) in the partitioning columns



# Which is Faster?



A  
13 select \* from reports  
14 where extract(month from created\_date) = 2; |

B  
15 select \* from reports  
16 where extract(day from created\_date) = 1; |

# Creating a Partitioned Table

- Partitions automatically inherit all of the columns, CHECK constraints and NOT NULL constraints from the parent table
- Ranges must be specified by DBA, they cannot overlap (TO value exclusive)
- Ranges should be complete, an attempt to insert data for which there is no range results in an error

```
1 CREATE TABLE public.payment (
2     payment_id integer DEFAULT
3         nextval('public.payment_payment_id_seq' :: regclass) NOT NULL,
4     customer_id integer NOT NULL,
5     staff_id integer NOT NULL,
6     rental_id integer NOT NULL,
7     amount numeric(5,2) NOT NULL,
8     payment_date timestamp with time zone NOT NULL
9 )
10 PARTITION BY RANGE (payment_date);
11
12 CREATE TABLE public.payment_p2020_01 PARTITION OF public.payment
13     FOR VALUES FROM ('2020-01-01') TO ('2006-03-31');
14
15 CREATE TABLE public.payment_p2020_02 PARTITION OF public.payment
16     FOR VALUES FROM ('2020-04-01') TO ('2006-06-30');
17
18 CREATE TABLE public.payment_p2020_03 PARTITION OF public.payment
19     FOR VALUES FROM ('2020-06-01') TO ('2006-09-30');
20
21 CREATE TABLE public.payment_p2020_04 PARTITION OF public.payment
22     FOR VALUES FROM ('2020-09-01') TO ('2006-12-31');
23
```

# Selecting from a Partitioned Table

- The pagila.payments table itself contains no data
- How come this query works?

```
121 select * from pagila.payment;
122
```

payment_id	customer_id	staff_id	rental_id	amount	payment_date
16050	269	2	7	1.99	2020-01-24 21:40:19.996577+00
16051	269	1	98	0.99	2020-01-25 15:16:50.996577+00
16052	269	2	678	6.99	2020-01-28 21:44:14.996577+00
16053	269	2	703	0.99	2020-01-29 00:58:02.996577+00
16054	269	1	750	4.99	2020-01-29 08:10:06.996577+00
16055	269	2	1099	2.99	2020-01-31 12:23:14.996577+00

# What's Actually Happening?

- The original partition table acts kind of like a view. Any SELECT queries get routed to the individual partition tables.
- The user need not know or care that partitioning is in place

```
123
124 explain select * from pagila.payment;

QUERY PLAN
Append (cost=0.00..389.78 rows=17319 width=30)
-> Seq Scan on payment_p2020_01 payment_1 (cost=0.00..20.57 rows=1157 width=...
-> Seq Scan on payment_p2020_02 payment_2 (cost=0.00..40.12 rows=2312 width=...
-> Seq Scan on payment_p2020_03 payment_3 (cost=0.00..98.44 rows=5644 width=...
-> Seq Scan on payment_p2020_04 payment_4 (cost=0.00..117.54 rows=6754 width...
-> Seq Scan on payment_p2020_05 payment_5 (cost=0.00..3.82 rows=182 width=29)
-> Seq Scan on payment_p2020_06 payment_6 (cost=0.00..22.70 rows=1270 width=...
```

# Why Bother?

---

```
125 explain select * from pagila.payment where payment_date < '2020-02-01';
```

## QUERY PLAN

Seq Scan on payment\_p2020\_01 payment (cost=0.00..23.46 rows=1157 width=30)

Filter: (payment\_date < '2020-02-01 00:00:00+00'::timestamp with time zone)

# Implementing Partitioning Yourself

- Postgres supports **declarative partitioning**
- Declarative means you say (declare) what you want and let the system look after how it actually does it
- If the default functionality doesn't fit your needs you can always implement it yourself, *BUT* it's a lot of work

# Implementing Partitioning Yourself

---

How do we allow users to select from one object and get data from all child objects?

```
1 CREATE TABLE public.payment (
2     payment_id integer DEFAULT
3         nextval('public.payment_payment_id_seq' ::regclass) NOT NULL,
4     customer_id integer NOT NULL,
5     staff_id integer NOT NULL,
6     rental_id integer NOT NULL,
7     amount numeric(5,2) NOT NULL,
8     payment_date timestamp with time zone NOT NULL
9 )
10 PARTITION BY RANGE (payment_date);
11
12 CREATE TABLE public.payment_p2020_01 PARTITION OF public.payment
13 FOR VALUES FROM ('2020-01-01') TO ('2006-03-31');
14
15 CREATE TABLE public.payment_p2020_02 PARTITION OF public.payment
16 FOR VALUES FROM ('2020-04-01') TO ('2006-06-30');
17
18 CREATE TABLE public.payment_p2020_03 PARTITION OF public.payment
19 FOR VALUES FROM ('2020-06-01') TO ('2006-09-30');
20
21 CREATE TABLE public.payment_p2020_04 PARTITION OF public.payment
22 FOR VALUES FROM ('2020-09-01') TO ('2006-12-31');
```

# Views

```
12  
13  CREATE VIEW public.payment as  
14      SELECT * FROM public.payment_p2020_01  
15      UNION  
16          SELECT * FROM public.payment_p2020_02  
17      UNION  
18          SELECT * FROM public.payment_p2020_03  
19      UNION  
20          SELECT * FROM public.payment_p2020_04;  
21
```

# Implementing Partitioning Yourself

---

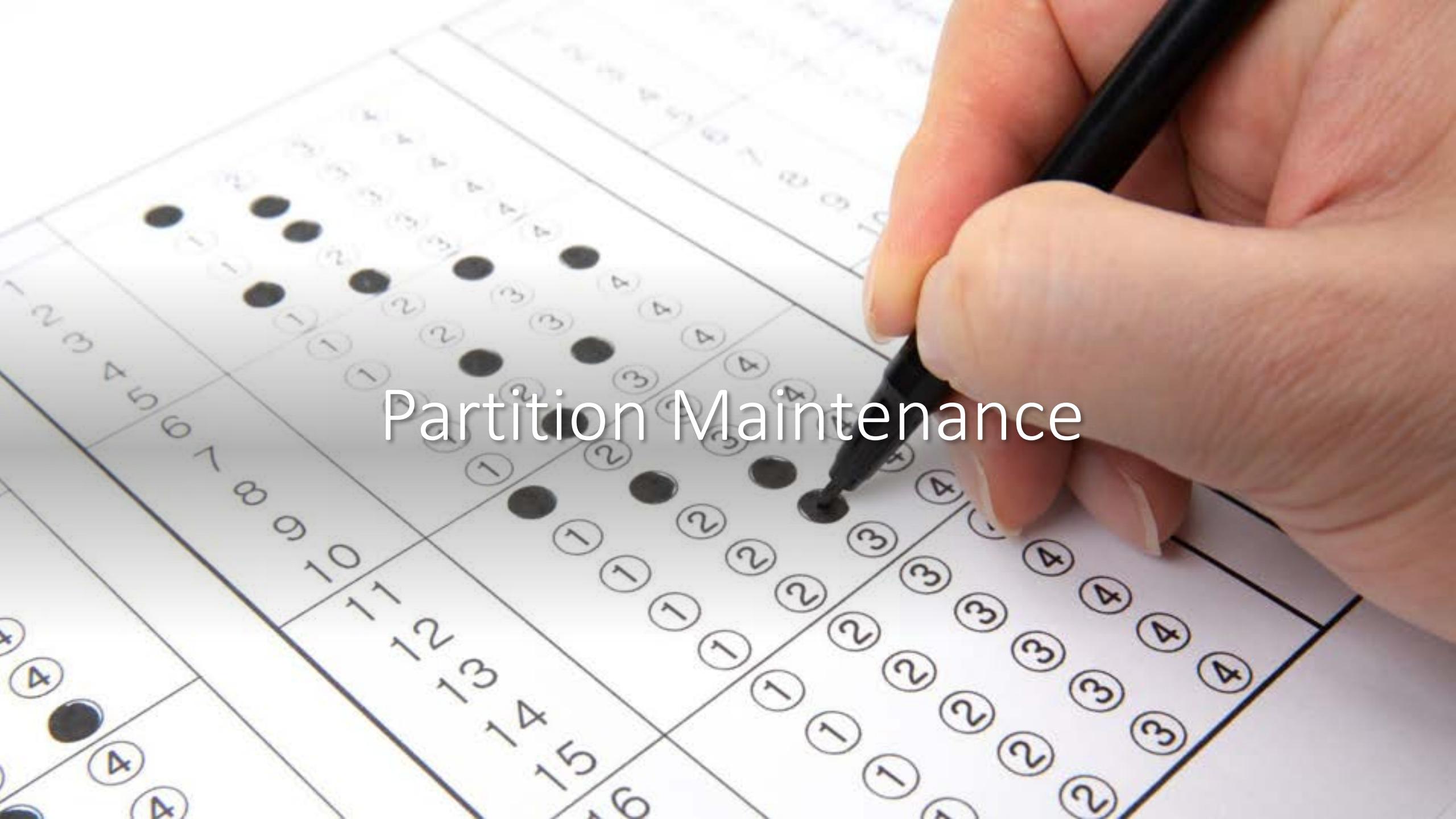
How do we allow users to insert into one object and have the data routed to the correct object?

```
1 CREATE TABLE public.payment (
2     payment_id integer DEFAULT
3         nextval('public.payment_payment_id_seq' ::regclass) NOT NULL,
4     customer_id integer NOT NULL,
5     staff_id integer NOT NULL,
6     rental_id integer NOT NULL,
7     amount numeric(5,2) NOT NULL,
8     payment_date timestamp with time zone NOT NULL
9 )
10 PARTITION BY RANGE (payment_date);
11
12 CREATE TABLE public.payment_p2020_01 PARTITION OF public.payment
13 FOR VALUES FROM ('2020-01-01') TO ('2006-03-31');
14
15 CREATE TABLE public.payment_p2020_02 PARTITION OF public.payment
16 FOR VALUES FROM ('2020-04-01') TO ('2006-06-30');
17
18 CREATE TABLE public.payment_p2020_03 PARTITION OF public.payment
19 FOR VALUES FROM ('2020-06-01') TO ('2006-09-30');
20
21 CREATE TABLE public.payment_p2020_04 PARTITION OF public.payment
22 FOR VALUES FROM ('2020-09-01') TO ('2006-12-31');
```

# Triggers

```
-- CREATE OR REPLACE FUNCTION payment_insert_trigger() AS
42 RETURNS TRIGGER AS $$ 
43 BEGIN
44     IF NEW.payment_date >= '2020-01-01' AND NEW.payment_date < '2020-03-31' THEN
45         INSERT INTO public.payment_p2020_01 VALUES (NEW.*)
46     ELSIF NEW.payment_date >= '2020-03-01' AND NEW.payment_date < '2020-06-30' THEN
47         INSERT INTO public.payment_p2020_02 VALUES (NEW.*)
48     ...
49     ELSE
50         RAISE EXCEPTION 'Date out of range. Fix the payment_insert_trigger() function!';
51     END IF;
52     RETURN NULL;
53 END;
54 $$ 
55 LANGUAGE plpgsql;
56
57 CREATE TRIGGER insert_payment_trigger
58     INSTEAD OF INSERT ON payment
59     FOR EACH ROW EXECUTE PROCEDURE payment_insert_trigger();
60
```

# Partition Maintenance



# Removing Old Data

- We can easily remove old data that is no longer necessary simply by dropping the partition

```
62  DROP TABLE public.payment_p2020_01 PARTITION OF public.payment;
```

- If we want to keep the data but remove it from the partition table we can DETACH it instead

```
65  ALTER TABLE payment DETACH PARTITION payment_p2020_01;
```

# Adding New Partitions

- It's easy to add a new partition declaratively
- We can also attach an existing table
- When we attach an existing table, every row needs to be checked to make sure it doesn't violate the partition
- This locks the entire partition table
- There is a workaround

```
66  
67 CREATE TABLE payment_p2019_04 PARTITION OF payment  
68   FOR VALUES FROM ('2019-09-01') TO ('2020-01-01');  
69
```

# Adding New Partitions

- By adding a check constraint we verify the table before we attach it
- When postgres sees that the check constraint matches the partition constraint it can skip verifying the data
- We usually want to drop the constraint afterwards because it's not needed

```
69  
70 CREATE TABLE payment_p2019_04  
71   (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);  
72  
73 ALTER TABLE payment_p2019_04 ADD CONSTRAINT chk_p2019_04  
74   CHECK ( logdate >= DATE '2019-09-01' AND logdate < DATE '2020-01-01' );  
75  
76 ALTER TABLE measurement ATTACH PARTITION payment_p2019_04  
77   FOR VALUES FROM ('2019-09-01') TO ('2020-01-01');  
78
```

# Indexes

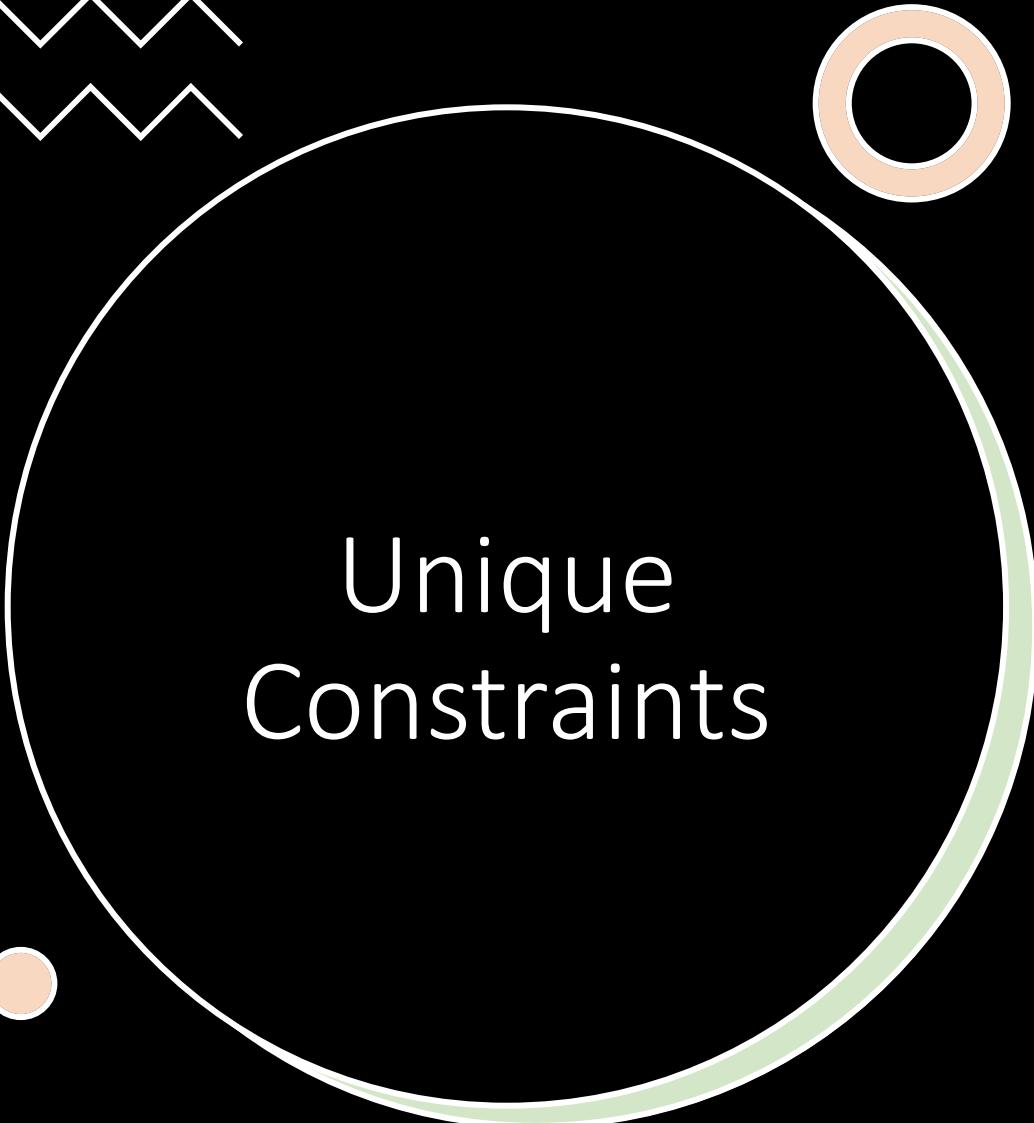
- Older versions of Postgres (< 12) don't support indexes on partition tables
- Indexes must be added to each partition individually
- Now that it's supported, it's best to add an index on the partition table, this will look after any future partitions

# Unique Constraints

---

- Unique constraints prevent duplicate values of a given column or combination of columns between rows
- Generally, a partitioned table holds all column definitions, constraints etc. and these are copied to the child whenever a new partition is created
- Why is it difficult to enforce unique constraints on a partitioned table?





# Unique Constraints

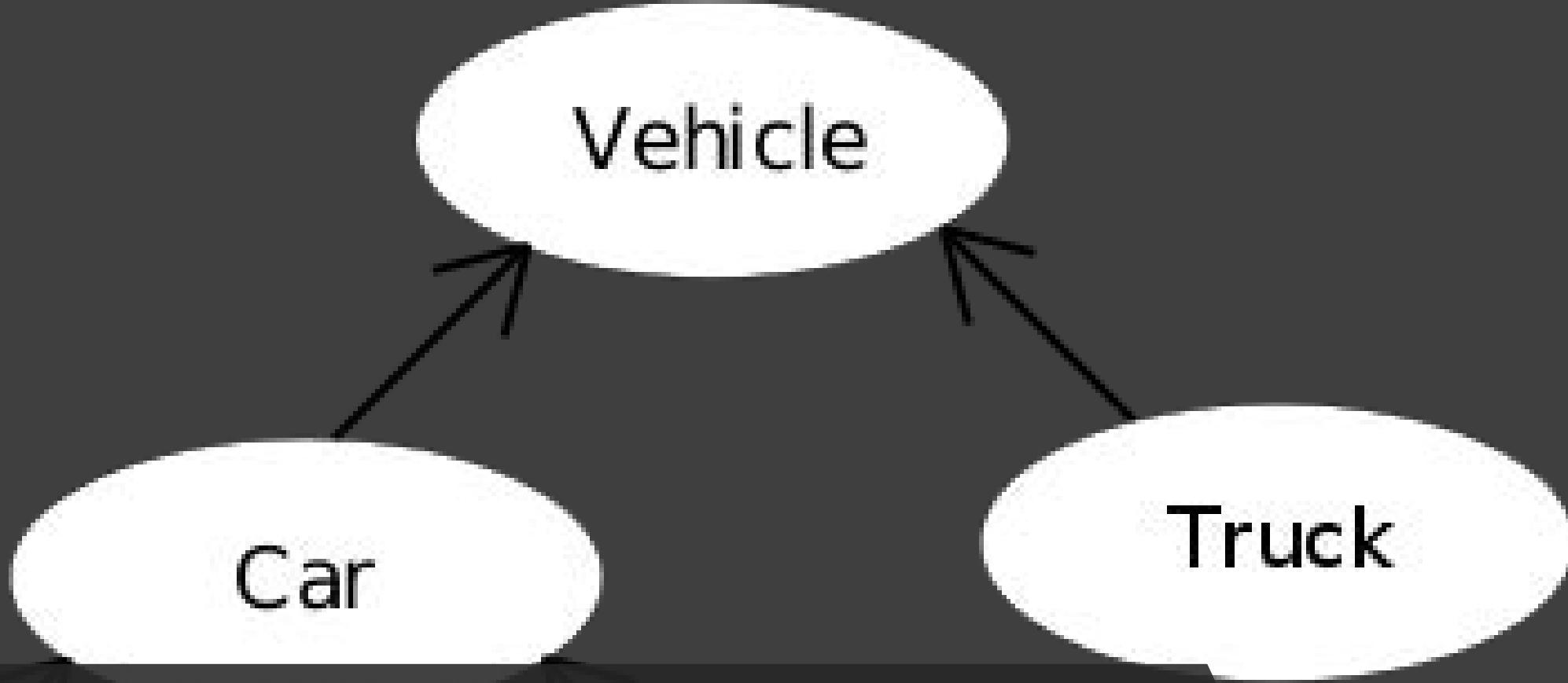
- Postgres  $\geq 12.0$  supports unique constraints **provided every column in the constraint is also a partition definition column**



# Inheritance

---

Object-Oriented Databases



## Inheritance and Postgres

- Postgres is an object-oriented database which supports *inheritance*
- Inheritance is a principle of object-oriented design which establish parent-child relationships between classes
- Sub-classes (children) are more specialised versions of their parents

## Adding Columns through Specialization

```
1 CREATE TABLE employees (
2     employee_number int,
3     first_name varchar(100),
4     last_name varchar(100)
5 );
6
7 CREATE TABLE researchers (
8     employee_number int,
9     first_name varchar(100),
10    last_name varchar(100),
11    area_of_expertise varchar(100)
12 );
13 |
14
```

# Inheritance

- All researchers are employees, but the researcher table gives more information specific to that job role
- We can say that the researcher table is a specialisation of the employee table
- If we create our tables like this it becomes difficult to maintain; any updates to the data definition of employee need to be carried over to researcher
- It's harder to query all employees, because now we need to search two tables

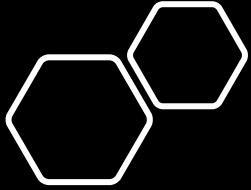
# Using Inheritance

```
14  
15  CREATE TABLE employees (  
16      employee_number int,  
17      first_name varchar(100),  
18      last_name varchar(100)  
19  );  
20  
21  CREATE TABLE researchers (  
22      area_of_expertise varchar(100)  
23  ) INHERITS (employees);  
24
```

# What does Inheritance Do?

- Inheritance creates a *link* between these tables
- Whenever I query the employees table, the researchers table will also be included
- I can query the researchers table separately if I'm only interested in researchers
- The ONLY keyword allows me to remove child tables from the query

```
26   SELECT * FROM ONLY employees;
```



# Caveats

- Unique constraints aren't supported with inheritance
- Foreign keys aren't supported with inheritance
- Not possible to have a foreign key from either child or parent

# Using Inheritance for Partitioning

- Inheritance was commonly used for partitioning before declarative partitioning became available
- More flexibility, but also more limited (constraints etc.)
- Inheritance can be used for partitioning, but it might be easier to rely on declarative partitioning and allow Postgres handle the details for you.

# Tablespaces

Working with the Filesystem

# What are Tablespaces?



A TABLESPACE IS A DATABASE OBJECT REPRESENTING A DIRECTORY ON YOUR FILESYSTEM



EVERY TABLE IN THE DATABASE HAS ITS DATA STORED IN A FILE ON DISK, THE *TABLESPACE* PROPERTY DETERMINES WHERE THAT FILE IS STORED



UNLESS OTHERWISE SPECIFIED, ALL TABLES WILL BE CREATED IN THE **PG\_DEFAULT\_TABLESPACE**.

# Why Use Tablespaces?



- Tablespaces allow you to control the disk layout of postgres
- If the postgres partition runs out of space, a new tablespace on another partition can be used as a temporary solution
- Secondly, if there are multiple storage media available, tablespaces can be used to put the most frequently used data on the fastest disk



# Importance of Tablespaces

- Tablespaces are an important part of the database, if the files are not accessible for any reason the database may fail
- Tablespaces should not be created on removable media (USB sticks, network mounts etc.)
- Tablespaces created on a temporary filesystem (like RAM) are a very bad idea

# Managing Tablespaces

- A tablespace is created as a *symlink* in the db root pointing to a directory on the filesystem
- To move a tablespace from one location to another, stop the server, update the symlink in pg\_tblspc, then restart the server
- Make sure all files are copied over to the new location before restarting the database

# Summary

---

- Partitioning can have similar results to indexing
- Partitioning can speed up queries with a lot of reads because of sequential access
- We can use RANGE or LIST partitioning
- It's important that our partitioning key is well chosen or we won't see much improvement
- We could partition manually, but it's difficult and comes with limitations

# Summary

---

- Inheritance allows us to create specialisations of tables
- Inheritance can be used to implement partitioning (though not recommended if avoidable)
- Tablespaces can be used to control where on disk a file lives
- This makes maintenance easier and can allow us to put certain objects on faster storage media
- Tablespaces are an integral part of the database and the data directories should never be touched while the database is running

The background of the slide features a subtle, abstract pattern of purple and green flowers and leaves, creating a soft, organic feel.

SYSTEMS AND DATABASE ANALYSIS

## Backup and Recovery

---



# When, not If

- A database can be thought of as a model of the universe from the point of view of an application, source of all knowledge
- Losing or corrupting this information can be catastrophic
- DBA's most important job is to ensure recovery is possible **when** disaster hits



# Stakeholders in Disaster Recovery Agreement

- Disaster recovery plans allow the business to plan for inevitable failure
- DBA is responsible for the technical side of disaster recovery
- Many other stakeholders, this is a business-wide issue

# Business Analysts / End users

- Work with business analysts and end users to agree an amount of downtime.  
Business analysts and end users state their requirements.  
DBA configures the database to match.
- Work with System Administrator and support staff to consider downtime
- What systems can be put in place, etc.





---

# Financial Department

- Building and maintaining disaster-recovery capabilities is expensive
- Less downtime = more expensive, zero downtime is extremely expensive
- End product is a compromise

---

# Service Level Agreements

- These are negotiated between the provider and the customer.
- Establish the needs, priorities and constraints of both parties.
- Quantify service outcomes and levels the customer will receive and cost justification of various service levels.





# Summary of Parties Involved

- In the provider organization

DBA has a role

- Optimise performance

Systems / Networks Administrator has a role

- Availability requires good systems and network administration.

Accounting division has a role.

- Zero downtime is expensive and may require a lot of extra equipment.

# DBA's Role

- To increase the MTBF
- To reduce the MTTR
- To reduce data loss.





# To Increase the MTBF

- MTBF – Mean Time Between Failures
- Some systems require 100% availability.  
E.g. satellite flight control, process control in an oil refinery, health monitors, etc.



## To Decrease the MTTR

- MTTR – Mean Time To Recover
- This refers to the downtime following a failure.
- Downtime can be more costly than data loss, because transactions are not being recorded.



## To Decrease the MTTR

- The database should be brought back up with minimal delay when it does fail.
- DBA must be very familiar with the procedure *before* the failure occurs, as it is difficult to look up procedures effectively while under pressure.
- Simulations should be run on all types of failure.

---

# Hardware

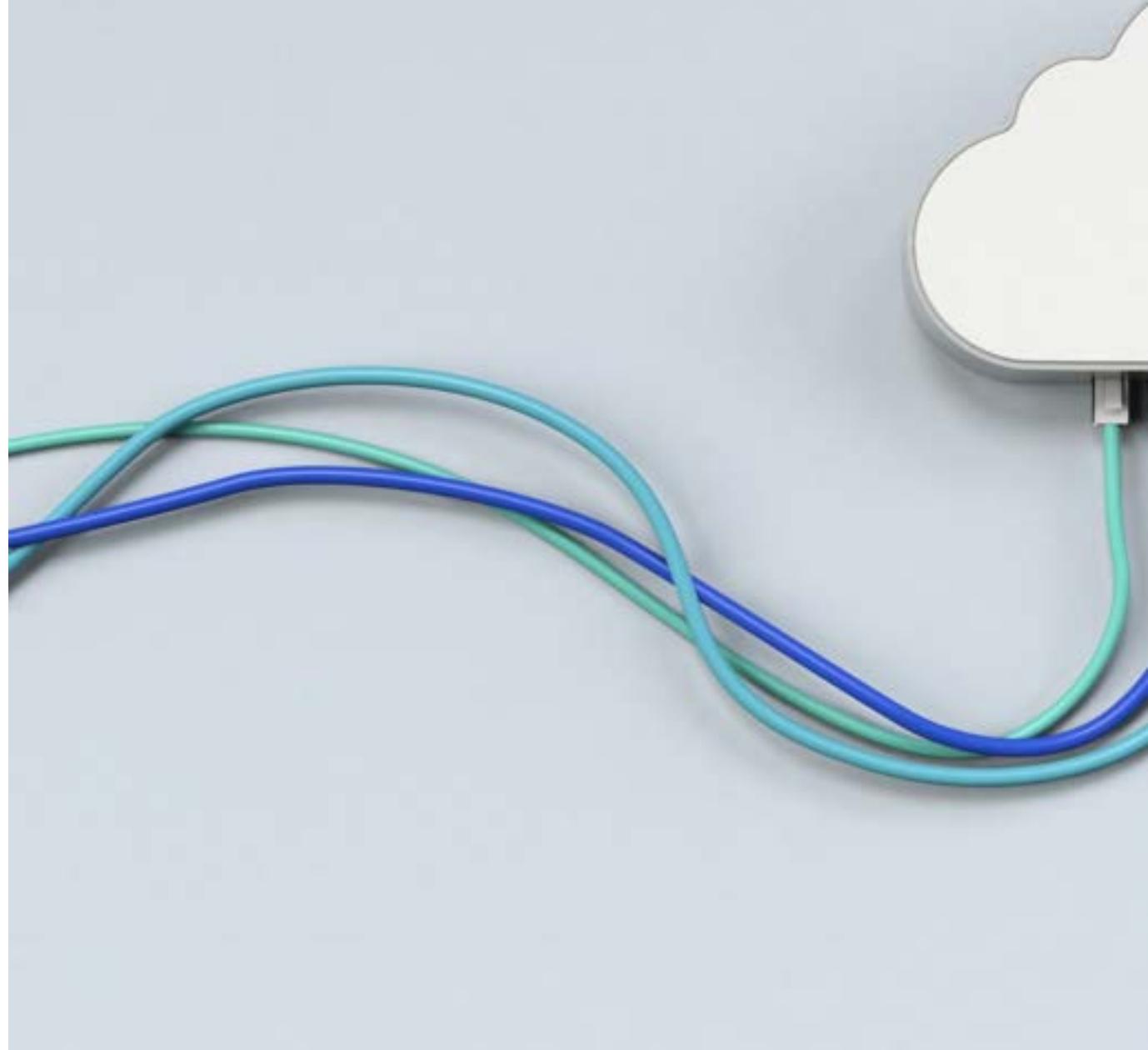


- Fault tolerance requires hardware redundancy.
- E.g. redundant datafiles may be held on separate disks. (`pg_receivexlog`)
- The system administrator must work with the DBA to ensure that disk redundancy and replacement is optimal.
- Warm standby can be maintained for instant failover (for another day)

---

# Hardware

- Network performance is critical  
The DBA must have agreed standards for uptime with the network administrators.
- The DBA should have a reasonable knowledge of the hardware, the network, the operating system, the programming language and the application.



# Categories of Failure

SYSTEM FAILURE



---

## Categories of failures

- Each category of failure should be covered by a service level agreement.
- The steps to be followed should be documented in a procedures manual.

---

# Types of failure

- Statement Failure
- Network Failure
- User Errors
- Media Failure
- Instance Failure



This Photo by Unknown Author is licensed under CC BY-NC

---

# Types of failure

- Statement Failure
- Network Failure
- User Errors
- Media Failure
- Instance Failure



This Photo by Unknown Author is licensed under CC BY-NC

---

# Statement Failure

- This is where an individual SQL statement fails.
- The server process executing the statement detects the failure and rolls back the ***statement***.
- If the statement is part of a multi-statement transaction, all statements that have already succeeded are kept, but not committed.
- Programmers should include exception statements in their code to ensure that the error is trapped.

---

# Statement Failure

## Causes of statement failure

- Invalid data; poor data entry validation, integrity / constraint violations.
- Logic errors; semantic errors in program code – e.g. deadlock, insert child before parent, etc.
- Insufficient Privilege; security structures are inappropriate.
- Space Management



# Avoiding failure due to space

- Ensure database partitions do not fill up
- Create new tablespaces on additional disks/partitions as necessary
- Use the VACUUM command to clean up old data in tables

---

# Types of failure

- Statement Failure
- Network Failure
- User Errors
- Media Failure
- Instance Failure



This Photo by Unknown Author is licensed under CC BY-NC



NETWORK CAN FAIL THROUGH THE LISTENER, THE  
NETWORK INTERFACE CARDS AND THE ROUTES.

## Network Failure

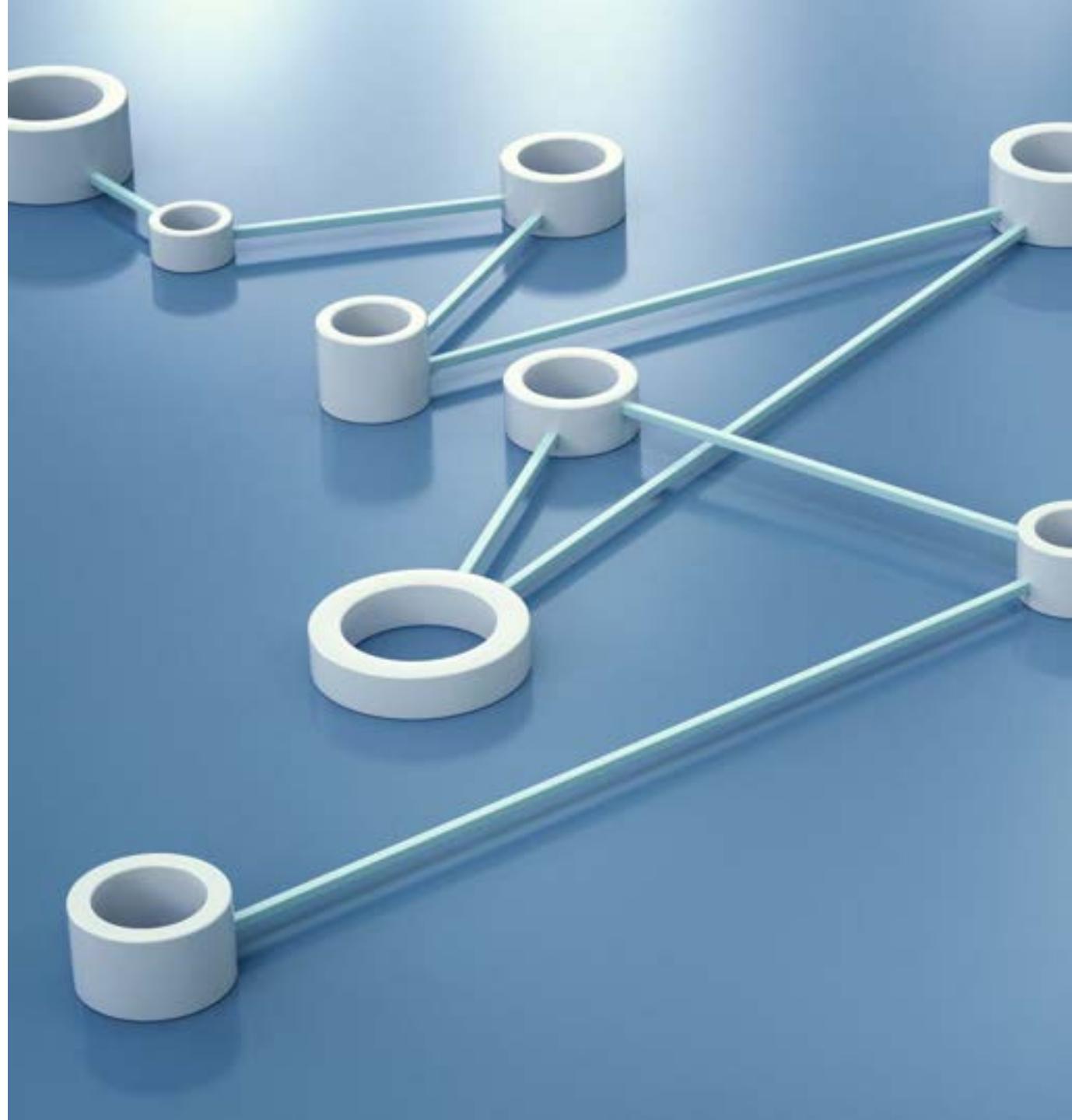
---

---

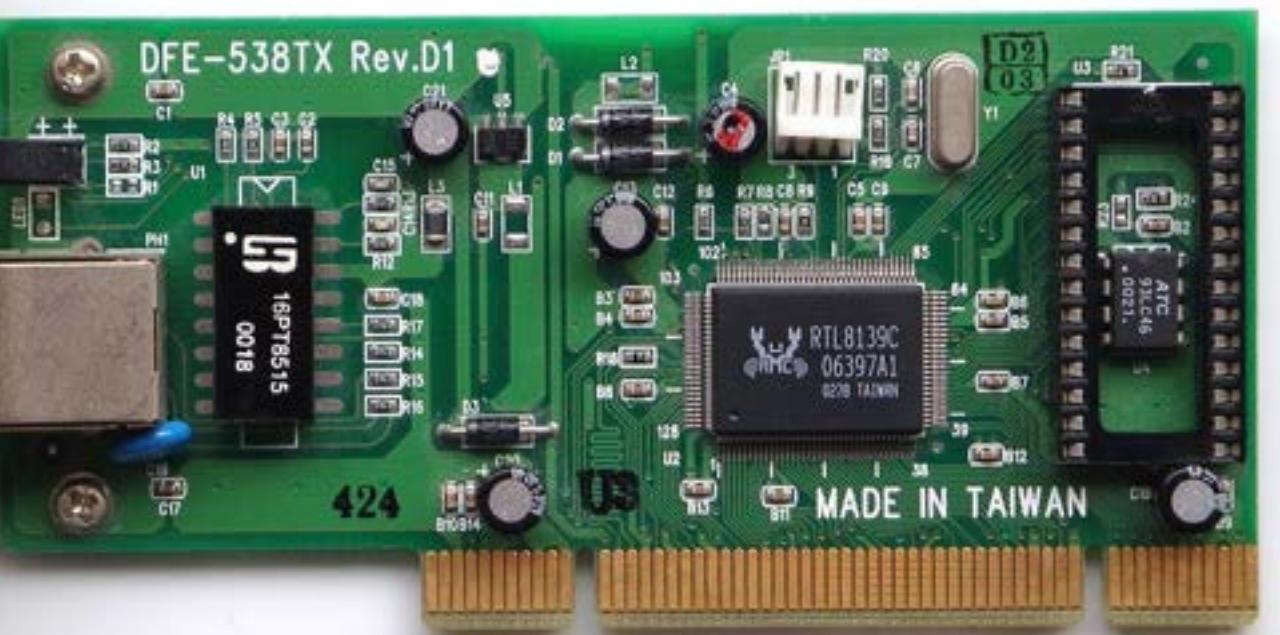
# Network Failure

Listener can Fail

- May not crash, but can be overloaded, causing it to timeout.
- High volumes of concurrent connection requests may result in errors.
- Configure multiple databases and make use of load balancing (usually requires commercial solution)



# Network Failure



## Network Interface Cards can Fail

- The server should have 2, for redundancy and performance, should listen on both.
- Each network card should be configured on a different physical subnet.
- Clients should be configured for connect time fault tolerance and load balancing

---

# Types of failure

- Statement Failure
- Network Failure
- User Errors
- Media Failure
- Instance Failure



This Photo by Unknown Author is licensed under CC BY-NC

# User Errors

- This is where the user does something syntactically correct, semantically disastrous.
- If the code is DML, and not committed, it can be rolled back.



---

## Point-in-time Recovery (PITR)

- If the database has been configured for it, point-in-time recovery can restore the database to an earlier state
- This requires downtime and often data loss, last resort
- More on this later

---

# Types of failure

- Statement Failure
- User Process Failure
- Network Failure
- User Errors
- Media Failure
- Instance Failure



This Photo by Unknown Author is licensed under CC BY-NC



---

## Media Failure

- Caused by
  - The disk is damaged, meaning the file is gone.
  - The system / database administrator deleted the file.
- Data loss can be prevented by proper configuration
  - Datafiles can be multiplexed
  - Warm standby on another disk/machine
  - Continuous archiving

# Recovery

---



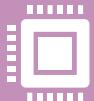
Uses archived WAL logs



The changes are extracted, in chronological order, from the archive logs generated since the backup was taken.



Archived WAL files should be multiplexed over different devices.



Datafiles must be protected by hardware redundancy: convention RAID systems or Oracle's ASM (automatic storage management).

---

# Types of failure

- Statement Failure
- User Process Failure
- Network Failure
- User Errors
- Media Failure
- Instance Failure



This Photo by Unknown Author is licensed under CC BY-NC



---

## Instance Failure

- This is a disorderly shutdown or crash.
  - Power cut
  - Switching off / rebooting the server
  - Hardware problems
- The database may be corrupted.



# Instance failure process

- The database may be
  - Missing committed transactions
  - Storing uncommitted transactions.
- The server tries to recover on start-up by replaying the transaction logs
- Index files may need to be re-built by VACUUMing the table or using the REINDEX command (indicated by log message)



# Durability and the WAL

---

---

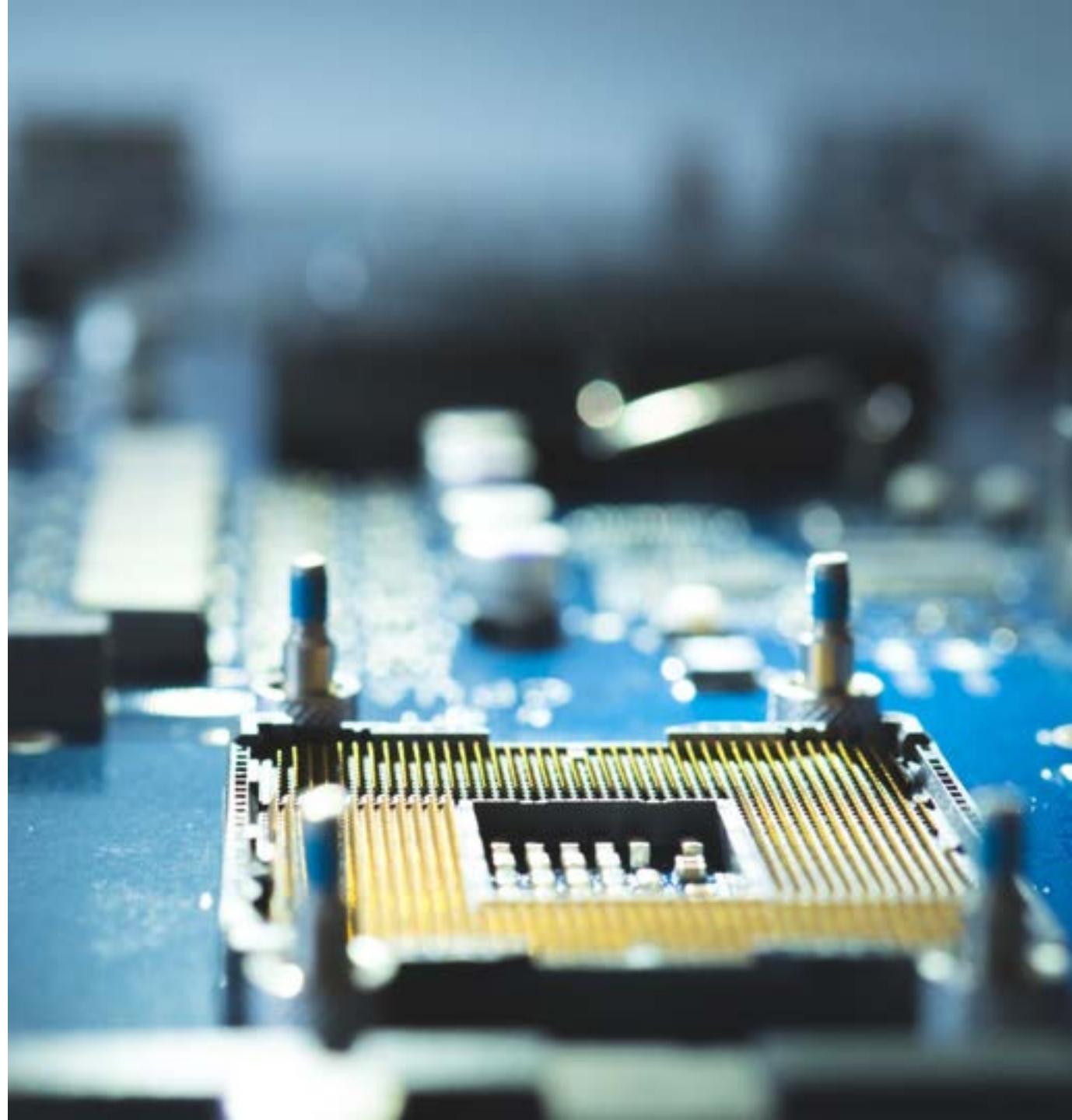
# Durability in PostgreSQL

- Relational databases provide the ACID guarantee
- With Durability, a database guarantees that *once a transaction is committed* it will not be lost in the event of a system failure
- PostgreSQL ensures this through Write-Ahead-Logs (WAL logs)

---

# RAM and Hard Disk

- Hard disk access is slow
- Performance can be increased by keeping tables in RAM where possible (caching)
- PostgreSQL doesn't necessarily write every insert/update/delete back to disk
- All changes written to disk on server shutdown



---

# RAM and Volatility

- Ram is volatile
- If the system crashes unexpectedly any changes held in RAM will be lost
- Cannot rely on a safe shutdown



# WAL and Durability

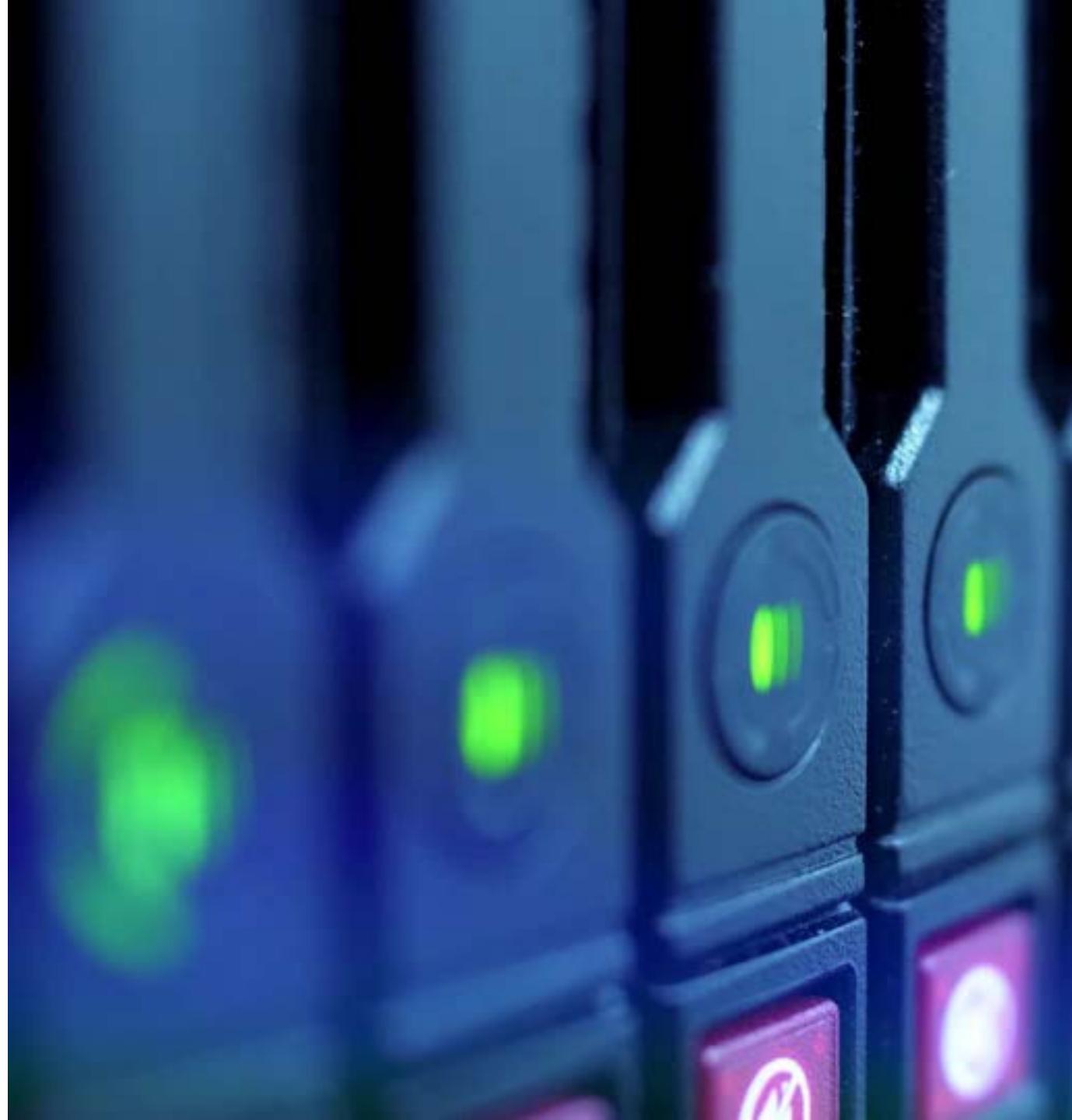


- Postgres maintains a Write-Ahead-Log (WAL)
- Any changes are written to this log file on commit (and is required for the transaction to complete successfully)
- In the event of a systems failure, any updates which hadn't been written to disk can be *replayed* from the WAL log to bring the database up-to-date

---

# Data Data Everywhere

- WAL files take up quite a lot of room
- A postgres server running indefinitely will produce an **infinitely long** WAL file.
- Large WAL files = more disk space needed + longer recovery times
- A WAL file is only needed for transactions which haven't yet been written to disk so we can clear our current WAL as soon as we know our hard disk is up to date

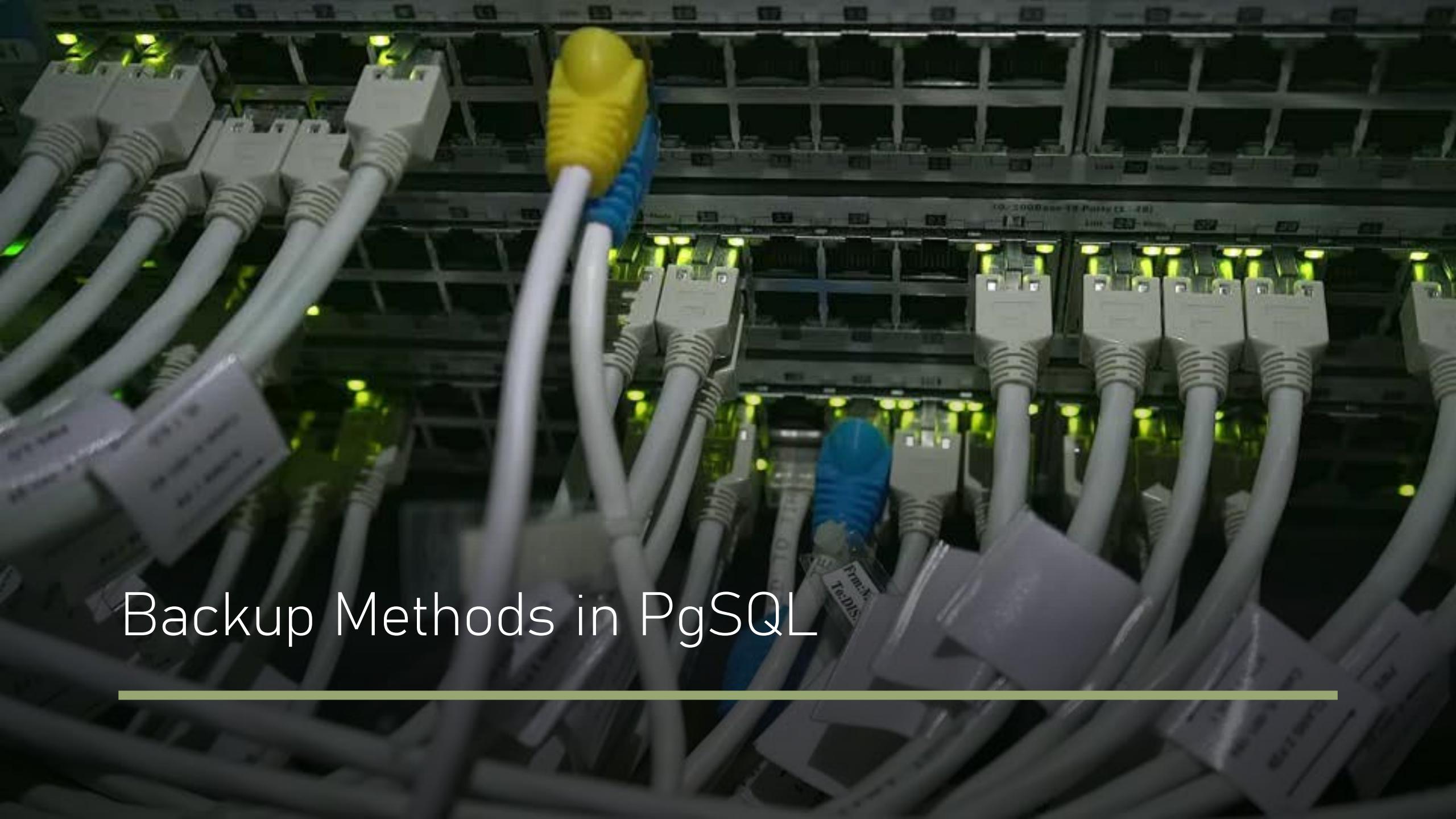


---

# The CHECKPOINT command

- The *CHECKPOINT* command forces postgres to write all committed transactions to disk, allowing us to clear down the WAL file
- The CHECKPOINT command is IO-intensive so will degrade database performance while running
- This can be run manually by any superuser but is also called regularly by the system
- `Checkpoint_timeout` and `max_wal_size` can be configured to adjust how often this is called





# Backup Methods in PgSQL

---



# Backup Methods in Postgres

- SQL Dump
- File System Backup
- WAL Log Archiving

---

# PG Dump

- The pg\_dump tool generates the SQL statements needed to recreate the database at a point in time
- Uses *transaction isolation* (more on this next week) to ensure a consistent snapshot is taken of the database
- Easy to run and easy to restore

```
pg_dump -h host1 dbname > outfile
psql -f infile postgres
```



# Filesystem Level Backup

- All required database files are stored in the postgres root directory
- We can in theory just copy the data directory to take a full backup of the database

```
tar -cf backup.tar /usr/local/pgsql/data
```



# Filesystem Level Backup

## Two Restrictions

1. All or nothing. Although we can dig out the individual table files they will only work in the context of the whole cluster. No easy way to do a partial backup
2. The database server must be shut down to ensure that the snapshot remains consistent



# Minimizing Disruption with rsync

- The **rsync** command is a useful tool for backups and archiving
- Allows incremental backups
  - Run rsync once with the database running
  - Stop database
  - Run again with --checksums flag to only update changed files
- This two-stage backup requires less downtime than a single tar command



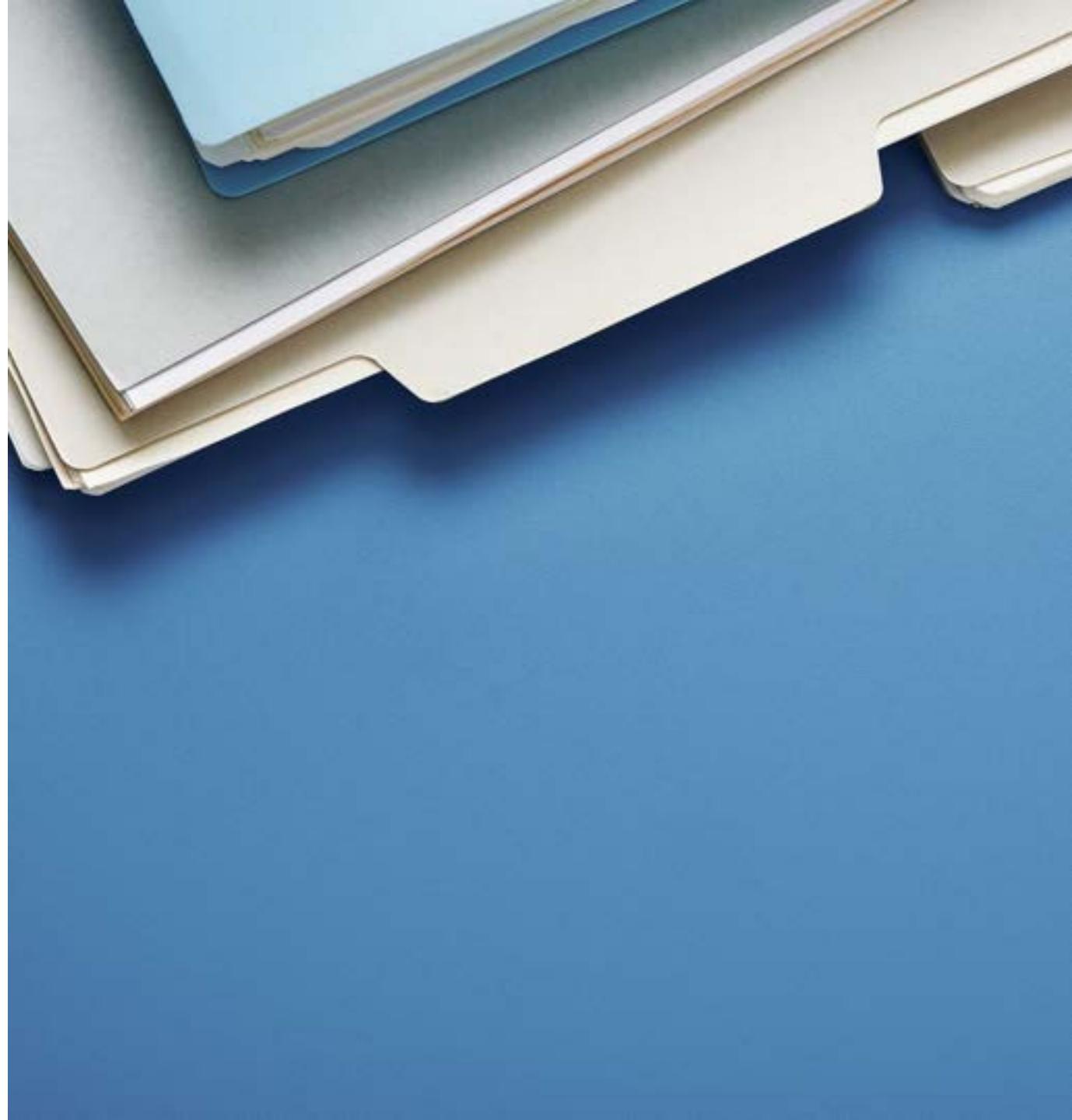
# Filesystem Level Backup

- Although supported, filesystem level backup is not recommended.
- Probably larger than SQL dump
- More disruptive than SQL dump
- More error-prone than SQL dump

---

# WAL File Recovery

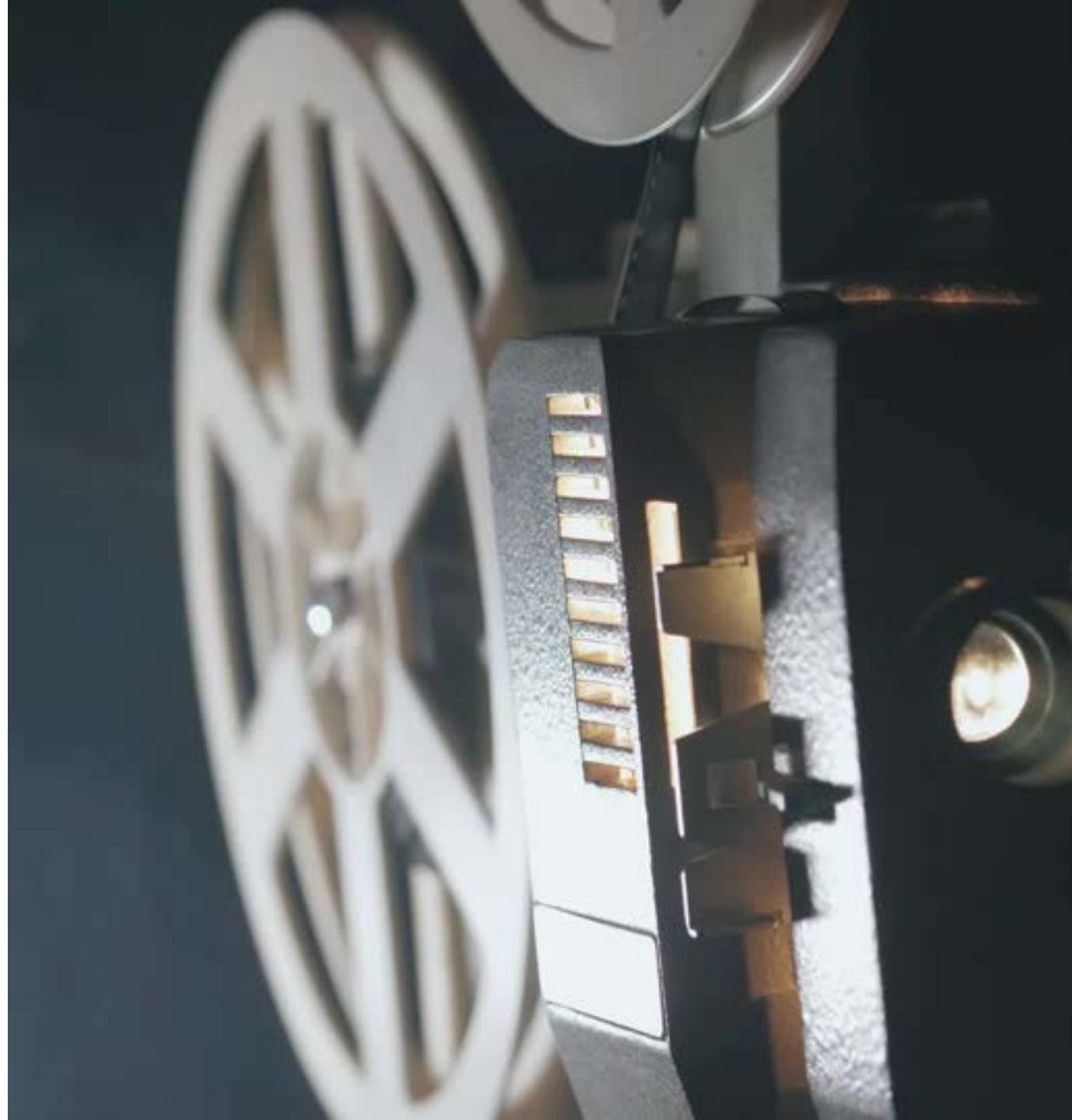
- WAL files are rotated as they are used, and old files are overwritten
- If we keep a copy of these files we can rebuild the database to any point in time
- WAL files usually need to be stored on another disk due to size constraints



---

# WAL Segments

- A WAL segment is a single unit of file storage
- Once a segment is filled, it will be swapped out for another segment
- Eventually the segment will be recycled
- Segment size can be controlled in PgSQL, 16MB default





# Archiving WAL files

- When a segment is filled, we have the option of archiving it
- The *archive\_command* parameter allows us to specify a command which will be run when a segment is complete
- We can use this to copy our file to a different location or do anything we want

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p  
          /mnt/server/archivedir/%f' # Unix
```



# Forcing a WAL switch

- By default the archive command will only run when a segment is full
- If database is low traffic we can set the `archive_timeout` parameter to force a switch at regular intervals
- We can execute the `pg_switch_xlog()` function in the database to manually force a switch

```
SELECT pg_switch_xlog();
```



# Performing a WAL-backup (Overview)

- Ensure WAL-archiving is working
- Put the database in backup mode `SELECT pg_start_backup('label');`
- Perform a filesystem backup as usual
- Stop the backup `SELECT pg_stop_backup();`
- Ensure the last WAL archive is copied into the backup and you're done

---

# Restoring the Database

- A WAL archive can be restored by stopping the server, cleaning out the **data directory** and replacing with the data files taken during the filesystem backup
- Next step is to replace the contents of pg\_xlog with the archive files generated during the backup
- By putting the database into recovery mode and restarting it, PostgreSQL will re-run any missing transactions and automatically correct the filesystem backup

---

# Summary

- One of the key responsibilities of a DBA is to plan for disaster recovery
- Disaster-recovery-capability is expensive, trade-offs are involved
- DBAs focus on increasing the MTBF and decreasing the MTTR
- Redundancy is very important for fault tolerance and avoiding unnecessary failure



# Summary

- There are 5 main categories of failure
- Each category has different causes, impacts and remedies
- Some of these are outside the DBA's control



# Summary

- Instance failure is particularly dangerous because databases often keep important information in RAM
- Durability is ensured using a Write-Ahead-Log
- This keeps track of all transactions in the database and allows recovery by replaying all transactions since the last backup

---

# Summary

- There are 3 different types of database backup in PostgreSQL
  1. `sql_dump` is simple, flexible and effective
  2. filesystem backups can be done but are difficult, error-prone and not generally recommended
  3. `pg_start_backup` leverages the WAL in conjunction with a filesystem backup to ensure a consistent snapshot is taken

# Transactions and Isolation Levels

Systems and Database Administration

# Databases and Concurrency

- Databases allow multiple users to connect at the same time (concurrently)
- Shared concurrent access to resources is one of the oldest problems facing software engineers



# What Kind of Problems?

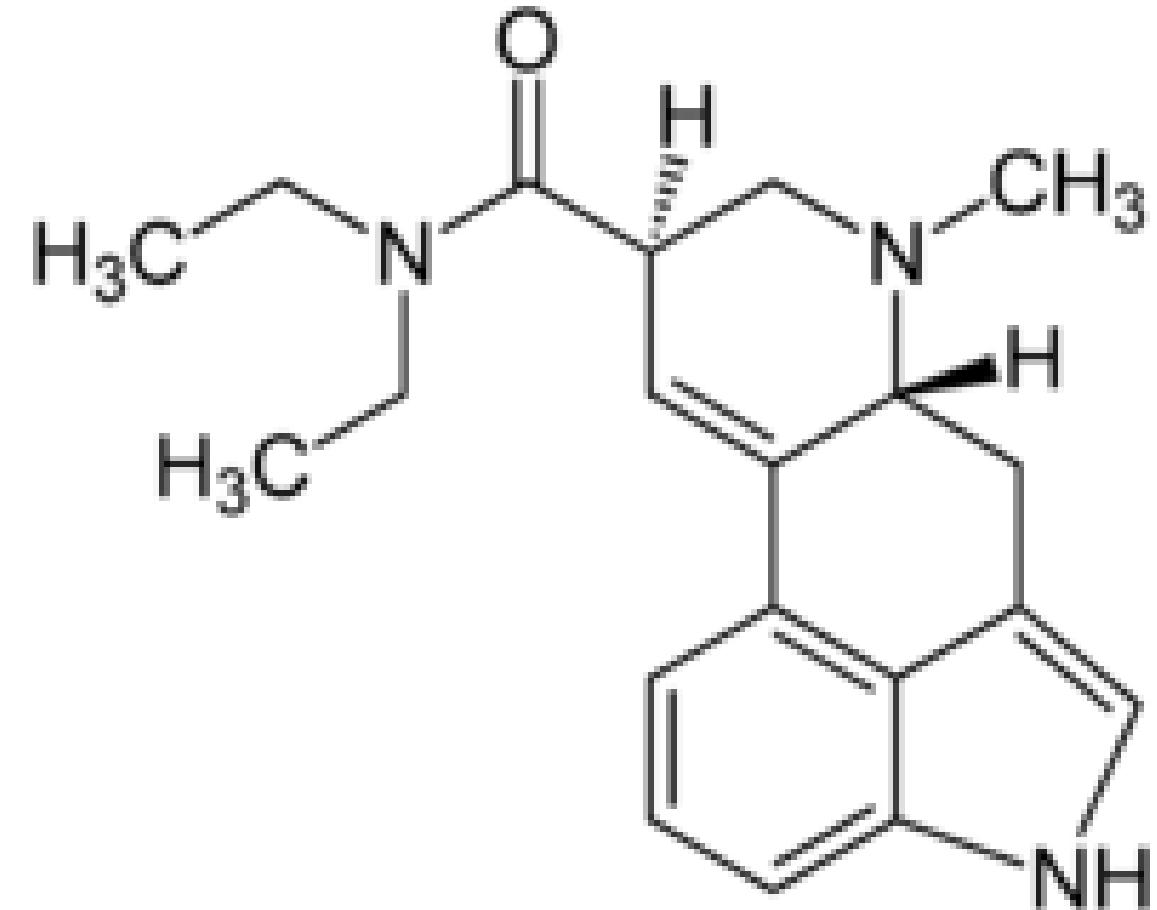
---

```
1 do $$  
2 declare  
3     act_balance numeric(8,2);  
4     new_balance numeric(8, 2);  
5 begin  
6     select balance into act_balance  
7     from current_accounts  
8     where act_no = '5524913'; -- RETURNS 520.50  
9  
10    select apply_interest(act_balance)  
11    into new_balance; --RETURNS 522.50  
12  
13    update current_accounts  
14    set balance = new_balance  
15    where act_no = '5524913';  
16  
17    commit;  
18 end;  
19 $$;  
20
```

```
22 do $$  
23 begin  
24     update current_accounts  
25     set balance = balance - 500.00  
26     where act_no = '5524913'  
27 end;  
28 $$;  
29
```

# Database Guarantees

- Atomicity
- Consistency
- Isolation
- Durability



A photograph of a person wearing a bright yellow waterproof jacket and black gloves, standing in a heavy downpour. The person is pointing their right arm out towards the right side of the frame. The background is blurred by the rain, showing what appears to be a city street or park area.

# Locks and Mutexes

Pessimistic Isolation Strategies

# Locks and Mutexes

- Earliest examples involved **Mutually Exclusive** locks (mutexes)
  - Similar to a library system, access must be *checked out* and *checked back in*
  - Boolean variable records whether the resource is in use
  - Requesters essentially form a queue waiting for the resource to become available



# Locks and Mutexes - Databases

- Most database systems allow **table-level** locks and **row-level** locks
- A lock prevents other sessions from accessing data until it is released
- Trade-off between guarantees of consistency and performance

```
1 do $$  
2 declare  
3     act_balance numeric(8,2);  
4     new_balance numeric(8, 2);  
5 begin  
6  
7     select balance into act_balance  
8     from current_accounts  
9     where act_no = '5524913' --RETURNS 520.50  
10    FOR UPDATE; -- Takes out a lock on this row  
11  
12    select apply_interest(act_balance)  
13    into new_balance; --RETURNS 522.50  
14  
15    update current_accounts  
16    set balance = new_balance  
17    where act_no = '5524913';  
18  
19    COMMIT; -- Releases all locks held by txn  
20 end;  
21 $$;
```

# Locks and Mutexes

- Waiting for transactions to complete can cause major performance issues
- Bottlenecks quickly build up, just like with traffic jams
- Locks must be used as sparingly as possible to avoid dragging the system to a halt

# SQL Lock Types

---

- **Access Share:** conflicts only with access exclusive – read-only queries
- **Share Row Exclusive:** prevents concurrent updates to table
- **Access Exclusive:** no other process is allowed access to the table



# Acquiring Locks

- The FOR SHARE and FOR UPDATE acquire row-level locks on any rows returned by the query.
- This row-level lock prevents any other process from updating that row
- FOR SHARE allows other processes to read the data
- FOR UPDATE prevents other processes from reading the data (it's about to change)



# Deadlocks

Usually if a row is locked, the requesting transaction will wait for the lock to be released

It is possible (and often happens) that two transactions each hold locks that the other needs (circular dependency)

In this case, neither transaction will finish, this is called deadlock, and causes one or both transactions to fail

```
1 do $$  
2 declare  
3   sender_balance      numeric(8,2);  
4   receiver_balance    numeric(8, 2);  
5   transfer_amt        numeric(8, 2);  
6 begin  
7  
8   select balance into sender_balance  
9   from current_accounts  
10  where act_no = '5524913' --RETURNS 520.50  
11  FOR UPDATE; -- Takes out a lock on this row  
12  
13  select balance into receiver_balance  
14  from current_accounts  
15  where act_no = '729968'  
16  FOR UPDATE;  
17  
18  update current_accounts  
19  set balance = sender_balance - transfer_amt  
20  where act_no = '5524913'  
21  
22  update current_accounts  
23  set balance = receiver_balance + transfer_amt  
24  where act_no = '729968'  
25  
26  
27  COMMIT; -- Releases all locks held by txn  
28 end;  
29 $$;
```

# Locks and Mutexes - Summary

- Conceptually easy to understand and implement
- Effective at preventing concurrency issues
- Bit of a Sledgehammer approach
- Possibility of slowing down system
- Possibility of deadlocks

The background image shows a large, reddish-brown rock formation with distinct horizontal layers, characteristic of sedimentary rock. It rises from a field of tall, dry, golden-brown grass. The sky above is overcast with heavy, grey clouds.

# Isolation Levels

Isolation Levels and the SQL Standard

# Explicit Locking

- We can explicitly take out locks on the database to protect ourselves from interference with other sessions
- This is complex and error-prone
- Better to ask the SQL engine to do it for us



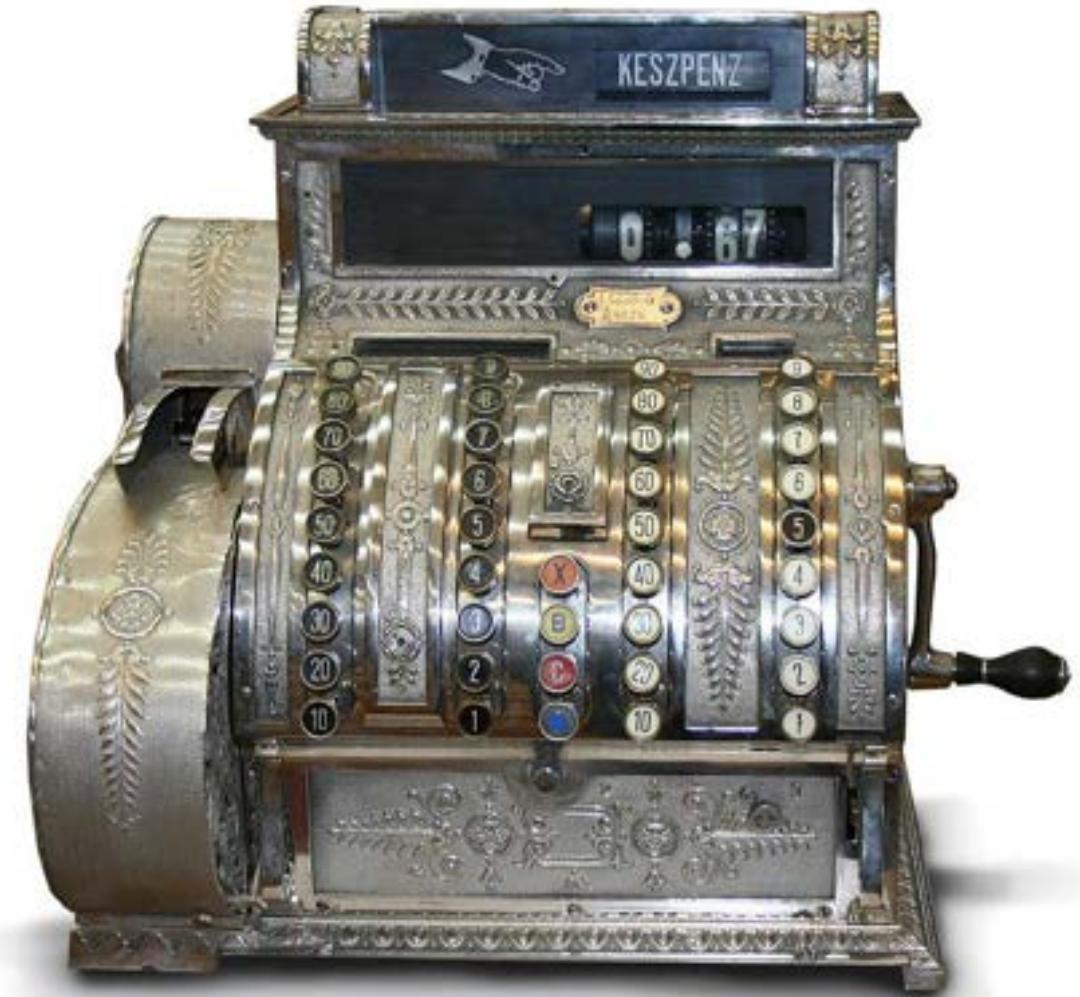
# The Isolation Trade-off

- Isolation is the phenomenon of each session *appearing* to be the only session operating on the database
- More isolation provides stronger guarantees but imposes bigger performance hits e.g. do we bother locking this table or not?
- The SQL standard defines 4 levels of isolation
  1. Read Uncommitted
  2. Read Committed
  3. Repeatable Read
  4. Serializable

# Transactions

---

- A SQL transaction is a group of queries which are executed as a **single atomic unit**
- In reality, each transaction has to be processed separately, but no query is considered complete until it is **committed**.
- Each transaction needs to *behave* as if no other session was operating when the transaction ran



# Isolation Levels – Read Uncommitted

- Even though transactions technically haven't happened until they're committed, the database will store the changes made as they happen and undo them if they're rolled back.
- The database will only check that the data is valid once it goes to commit, so there's a chance that the updates temporarily put invalid data into the database
- This can cause problems for other transactions querying the data after it's been stored in the database but before it's committed

```
isolation=# select * from marbles;  
id | colour  
---+-----  
 1 | White  
 2 | White  
 3 | Black  
 4 | Black  
(4 rows)
```

# Isolation Levels – Read Uncommitted

## Dirty Reads

```
1  
2  
3  
4 update marbles  
5 set colour = 'Black'  
6 where id = 1;  
7  
8  
9 --How many will I update?  
10 commit;
```

```
1 update marbles  
2 set id = 1  
3 where id = 2;  
4  
5  
6  
7  
8 commit; -- ERROR: ROLLBACK!  
9  
10
```

# Isolation Levels – Read Uncommitted

- A query will see the latest value in the database, even if that value isn't committed
- Good for performance, no checks, ignores isolation
- Bad for reliability, high chance that information never makes it into the database
- **Read Uncommitted** isolation level allows dirty reads
- Postgres provides **read uncommitted** for compatibility but never allows dirty reads
- The **read committed** isolation levels prevents dirty reads

# Isolation Levels – Read Committed

- We can fix the issue of dirty reads by ensuring that a query will only see an update IF that update has been committed
- With locks, this can be done by holding all write locks until the transaction is completed, but releasing read locks immediately
- Significant improvement in isolation over read uncommitted, but still leaves edge cases

```
isolation=# select * from marbles;  
id | colour  
---+-----  
 1 | White  
 2 | White  
 3 | Black  
 4 | Black  
(4 rows)
```

# Isolation Levels – Read Committed

## Non-Repeatable Read

```
1 select colour from marbles  
2 where id = 2;  
3  
4  
5  
6 --what colour is it?  
7 select colour from marbles  
8 where id = 2;  
9 |
```

```
1  
2  
3 update marbles  
4 set colour = 'Black'  
5 where id = 2;  
6 commit;  
7  
8  
9
```

# Isolation Levels – Read Committed

- A non-repeatable read is when a transaction re-reads data it has previously read and finds that the data has been modified by another transaction
- Not solved by **read committed** because the other transaction has committed by the time of the second read
- Solved by setting the isolation level to **repeatable read**
- **read committed** is the default isolation level in postgres

# Isolation Levels – Repeatable Read

- We can fix the issue of non-repeatable reads by ensuring that no updates are made to a row which has been selected as part of a running transaction
- With locks we can do this by holding onto row-level read *and* write locks until the transaction completes
- Significant increase in lock usage but provides additional guarantees
- Still leaves some edge cases

```
isolation=# select * from marbles;  
id | colour  
---+-----  
 1 | White  
 2 | White  
 3 | Black  
 4 | Black  
(4 rows)
```

# Isolation Levels – Repeatable Read

## Phantom Read

```
1 select * from marbles
2 where colour = 'White';
3
4
5
6 -- how many are returned?
7 select * from marbles
8 where colour = 'White';
9
10 commit;
```

```
1
2
3 update marbles
4 set colour = 'White'
5 where id = 3;
6
7 |
8
9
10 commit;
```

# Isolation Levels – Repeatable Read

- A phantom read is where the set of rows returned by a query changes during the course of a transaction
- Because these rows weren't seen by the original query they would not be affected by any row-level read locks.
- We can fix this by taking out *range* locks
- Provides very strong guarantees but at cost of large performance hit

# Isolation Levels - Serializable

- ***Serializable*** fixes the issue of phantom reads by taking out range locks in addition to row-level locks
- Serializable also provides the guarantee, that for any set of overlapping transactions it doesn't matter what order they finished in (more on this later)
- This is the strongest isolation level, providing strictest guarantees

```
isolation=# select * from marbles;  
id | colour  
---+-----  
 1 | White  
 2 | White  
 3 | Black  
 4 | Black  
(4 rows)
```

# Isolation Levels - Serializable

- What's the result?

```
1 update marbles
2 set colour = 'Black'
3 where colour = 'White';
4
5
6
7
8
9 commit;|
```

```
1
2
3
4
5 update marbles
6 set colour = 'White'
7 where colour = 'Black';
8
9 commit;|
```

# Serializable – Serial Transactions

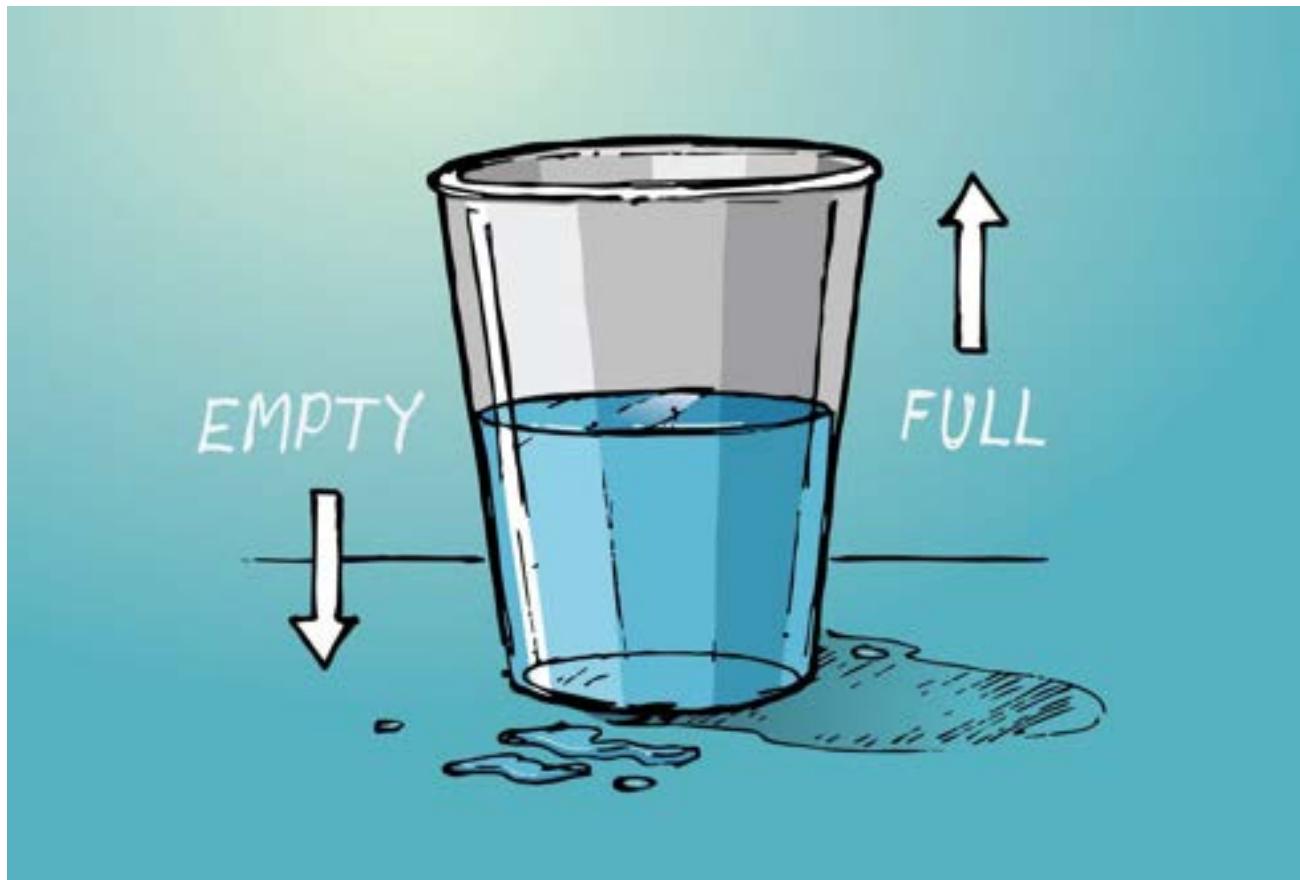
- Serial means "one after another"
- Transactions often overlap in the database
- Race conditions can cause problems where the order of execution is important
- Serializable will detect whenever race conditions could cause problems and will roll back transactions to prevent it.
- Transactions which fail due to serialization can usually be re-run immediately

# Isolation in SQL/PgSQL

---

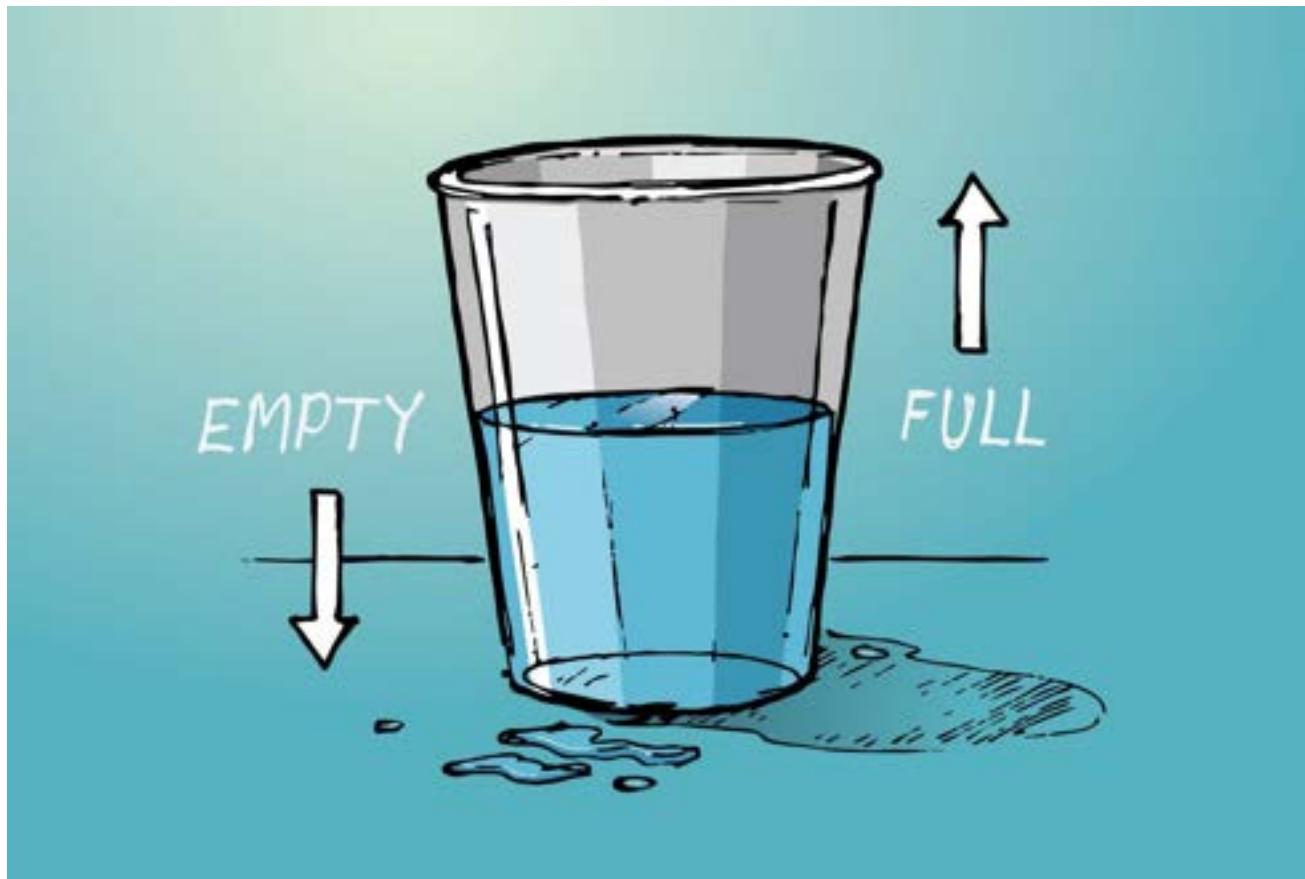
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

# Locking – Do We Need it?



- Locking adds latency to the database
- Problems solved are serious but conflicts are rare
- Locking is a form of **pessimistic** concurrency control
- Fixes problems before they happen

# Optimistic Concurrency Control



- Locking assumes conflicts will happen
- Optimistic concurrency control (OCC) assumes everything will be OK
- With OCC we don't bother with locks, but right before we commit we check that nothing has changed
- If some problem is detected then the transaction is aborted

# Snapshot Isolation



Time Travelling Databases

# Problems with Lock-Based Concurrency Control

- Concurrency control solves a problem which is potentially disastrous but generally rare
- Locks mean that reads block writes and writes block reads, even when there wouldn't be any issue allowing them
- In high-throughput databases this can essentially cause a *traffic jam* and bring the database grinding to a halt

# Multi-Version Concurrency Control



- Postgres, along with most other major database systems, has moved from lock-based isolation to **snapshot isolation**
- Snapshot Isolation is a nice improvement over locks which reduces database congestion
- Snapshot Isolation is a form of optimistic concurrency control and is implemented in postgres using multi version concurrency control (MVCC)

# Multi-Version Concurrency Control



- Snapshot Isolation is a guarantee that all transactions will see a *consistent* version of the database
- If a database implements some system of versioning, it's possible to *rewind time* to when a transaction started
- This means all queries in the transaction will behave as if no other transactions were taking place

# Postgres DB Versioning

- Postgres stores versioning data internally in its tables
- Old values are retained and versioned when changed
- This allows a query to return to a point-in-time
- If a transaction tries to write any data which has changed since the transaction began it will be aborted by the system and must be re-run

```
1 select * from marbles
2 where colour = 'White';
3
4
5
6 -- how many are returned?
7 select * from marbles
8 where colour = 'White';
9
10 commit;
```

```
1
2
3 update marbles
4 set colour = 'White'
5 where id = 3;
6
7 |
8
9
10 commit;
```

# Cleaning Up

---

- All of the old versioning data is kept by postgres in the tablespace file
- This needs to be periodically cleaned out
- Postgres will do this automatically but it can be forced by the DBA using VACUUM





# Summary

- Locks are a way to share access to resources in computer systems
- Databases have long used locks to enable multiple users to access the database simultaneously
  - Read Locks
  - Write Locks
  - Row-Level Locks
  - Table-Level Locks (range locks)
- Locks make bugs less likely but can dramatically decrease performance
- Deadlocks can cause transactions to fail

# Summary

- The SQL standard provides isolation levels to allow the user to determine how much isolation they need
- Lower isolation levels = better performance
- Higher isolation levels = stronger guarantees
- Each isolation level has a corresponding read phenomenon it prevents
- Locking is a form of *pessimistic concurrency control*, it assumes the worst

# Summary

Many modern databases have moved away from locks and towards snapshot isolation

Snapshot isolation allows a transaction to view the database at a previous point in time

Prevents many of the issues tackled by lock-based isolation

Optimistic concurrency control, will ensure that a txn does not write to any data that was updated since it began, otherwise roll back

The VACUUM command allows the DBA to clean up old versioning data which is no longer needed

# All things distributed

Adapted from 'Eventually consistent – revisited' by werner vogels

[HTTPS://WWW.ALLTHINGS DISTRIBUTED.COM/2008/12/EVENTUALLY\\_CONSISTENT.HTML](https://www.allthingsdistributed.com/2008/12/EVENTUALLY_CONSISTENT.HTML)

# Eventually Consistent

- ▶ Desirable features in an information system:
  - ▶ security,
  - ▶ scalability,
  - ▶ availability,
  - ▶ performance, and
  - ▶ cost effectiveness

while continuously serving millions of customers globally.

# How many...

Patricia O'Byrne

- ▶ ...Tweets per day...
  - ▶ 500 million
- ▶ ...customers on Amazon.com
  - ▶ 310 million
- ▶ ...active Facebook users
  - ▶ 2.13 billion use it monthly
  - ▶ 1.15 billion mobile daily active users
- ▶ Coursera
  - ▶ 30 million users.
  - ▶ Introduction to Mathematical Thinking (Standord University)  
– 86,230 enrolments.

# Scale issues

- ▶ These systems process trillions of requests
- ▶ Low probability events are guaranteed to happen
- ▶ Replication guarantees consistent performance and high availability.
  - ▶ But not without consequences

# Consistency - guiding principles

- ▶ Transaction protocols
  - ▶ Such as two-phase commit. These can achieve data consistency and availability, but don't tolerate network partitions.
- ▶ ACID consistency
  - ▶ When a transaction is finished the database is in a consistent state.
  - ▶ Often the responsibility of the developer writing the transaction but can be assisted by the database managing integrity constraints.
- ▶ Larger scale – partitions will happen.
  - ▶ Must choose between consistency and availability

# Developer conundrum

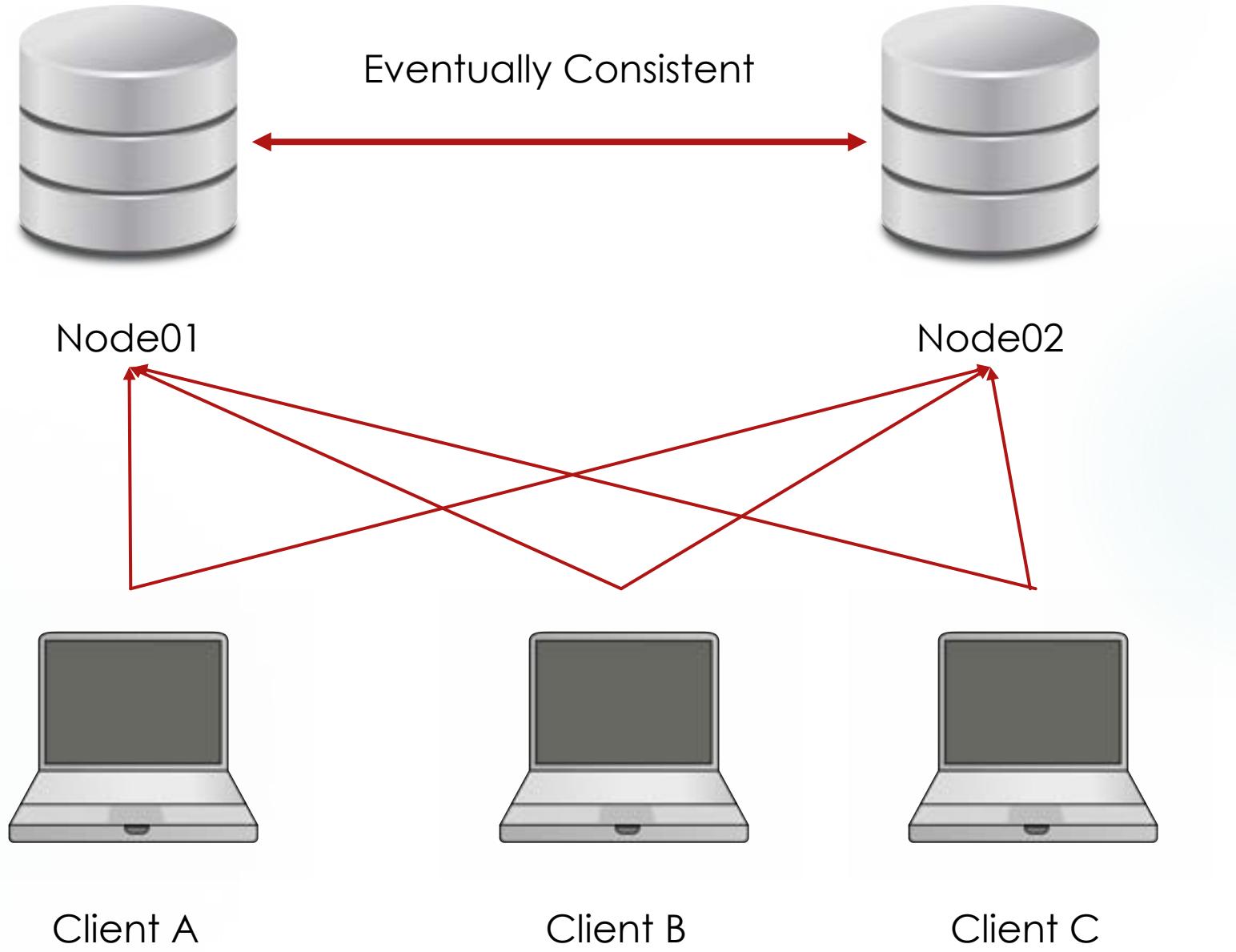
- ▶ If consistency is needed
  - ▶ the system may not be available to take a write.
  - ▶ If this write fails the developer must deal with what to do with the data to be written.
- ▶ If availability is needed,
  - ▶ The system may always accept the write,
  - ▶ BUT - a read may not reflect the result of a recently completed write.
  - ▶ The developer then has to decide whether the client requires access to the absolute latest update all the time.
  - ▶ There is a range of applications that can handle slightly stale data.

# Consistency—Client and Server

- ▶ How does the client observe updates?
- ▶ From the server side:
  - ▶ how do updates flow through the system?
  - ▶ what guarantees can be given with respect to updates?

# Client-side Consistency

- ▶ The client side has these components:
  - ▶ A storage system.
  - ▶ Process A.  
This is a process that writes to and reads from the storage system.
  - ▶ Processes B and C.  
These two processes are independent of process A and write to and read from the storage system.  
They are independent and need to communicate to share information.



# Client-side consistency

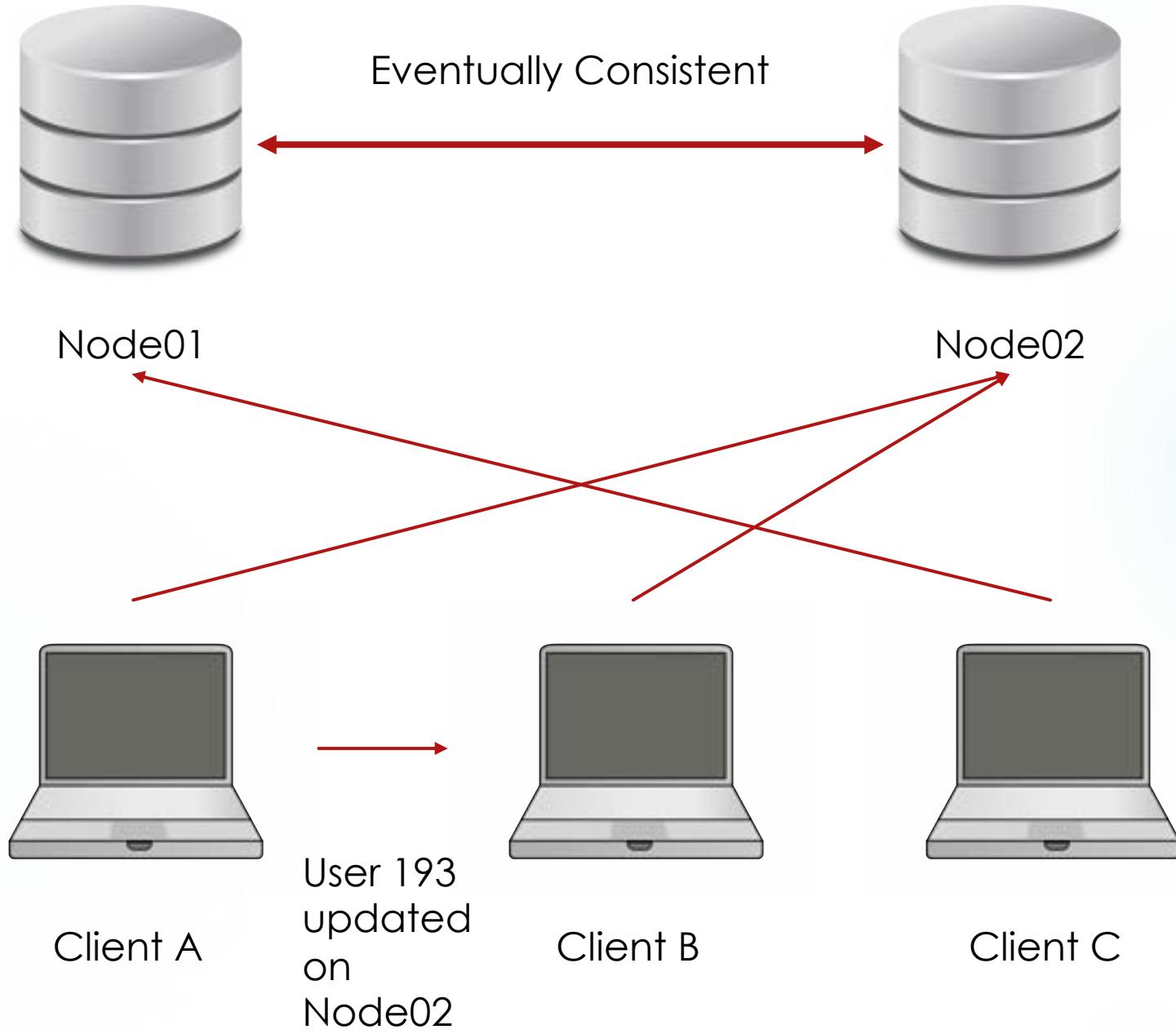
- ▶ How and when do observer processes (A, B, or C) see updates made to a data object in the storage systems?
- ▶ Assume process A has made an update to a data object:
- ▶ Strong consistency.
  - ▶ After the update completes, any subsequent access (by A, B, or C) will return the updated value.
- ▶ Weak consistency.
  - ▶ The system does not guarantee that subsequent accesses will return the updated value.
  - ▶ A number of conditions need to be met before the updated value will be returned.
  - ▶ This is the ***inconsistency window***.

# Eventual consistency

- ▶ The storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value.
- ▶ If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme.

# Flavours of eventual consistency

- ▶ Causal consistency.
  - ▶ If process A has told process B that it has updated a data item, a subsequent access by process B will return the updated value, and a write is guaranteed to supersede the earlier write.
  - ▶ Access by process C that has no causal relationship to process A is subject to the normal eventual consistency rules.



# Flavours of eventual consistency

- ▶ Read-your-writes consistency.
  - ▶ If process A has updated a data item, it always accesses the updated value and will never see an older value. This is a special case of the causal consistency model.
- ▶ Monotonic read consistency.
  - ▶ If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.
- ▶ Monotonic write consistency.
  - ▶ In this case the system guarantees to serialize the writes by the same process. Systems that do not guarantee this level of consistency are notoriously hard to program.

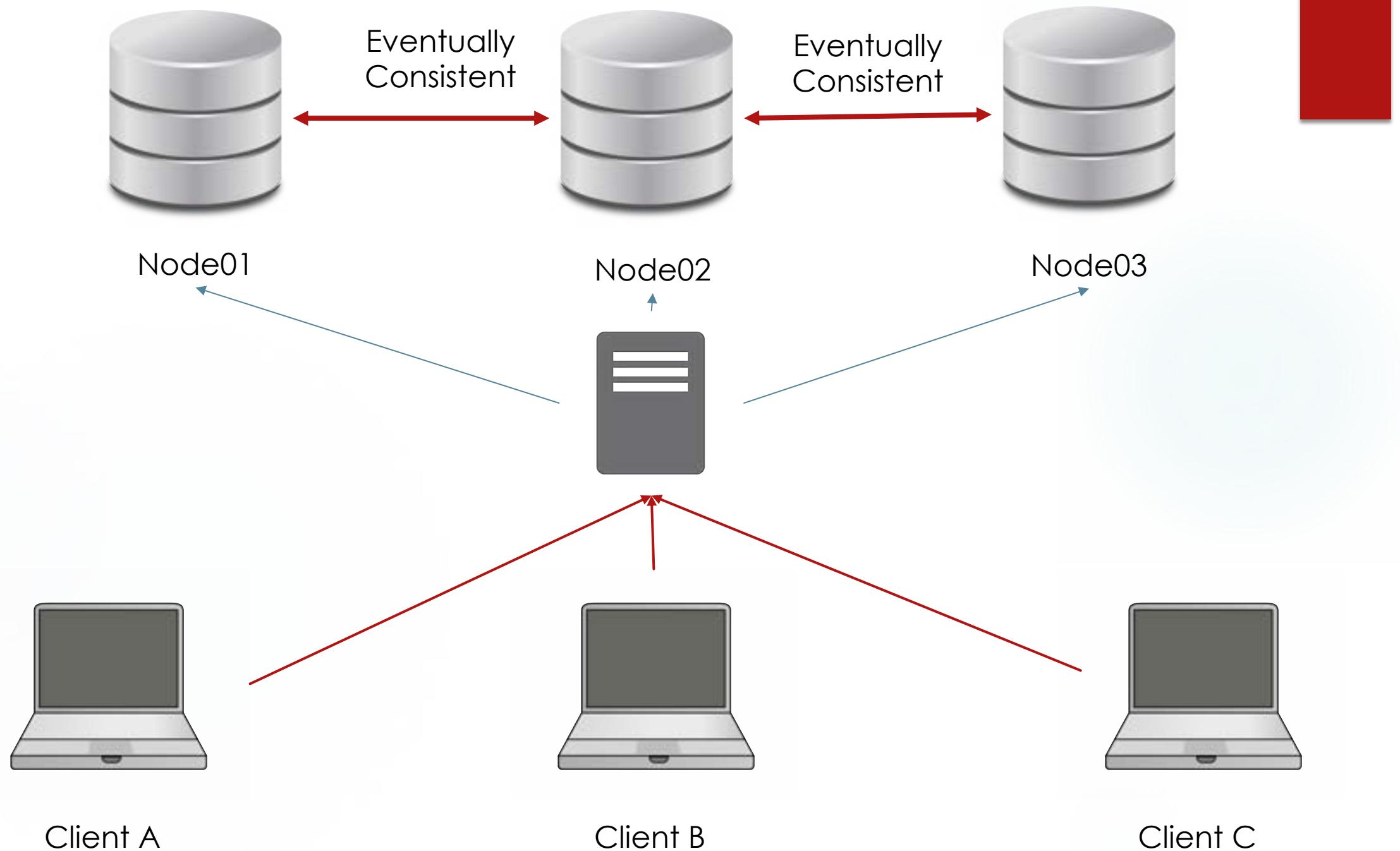
# Flavours of eventual consistency

- ▶ Session consistency.
  - ▶ If a process accesses the storage system in the context of a session, as long as the session exists, the system guarantees read-your-writes consistency.
  - ▶ If the session terminates because of a certain failure scenario, a new session needs to be created and the guarantees do not overlap the sessions.

# Server-side Consistency

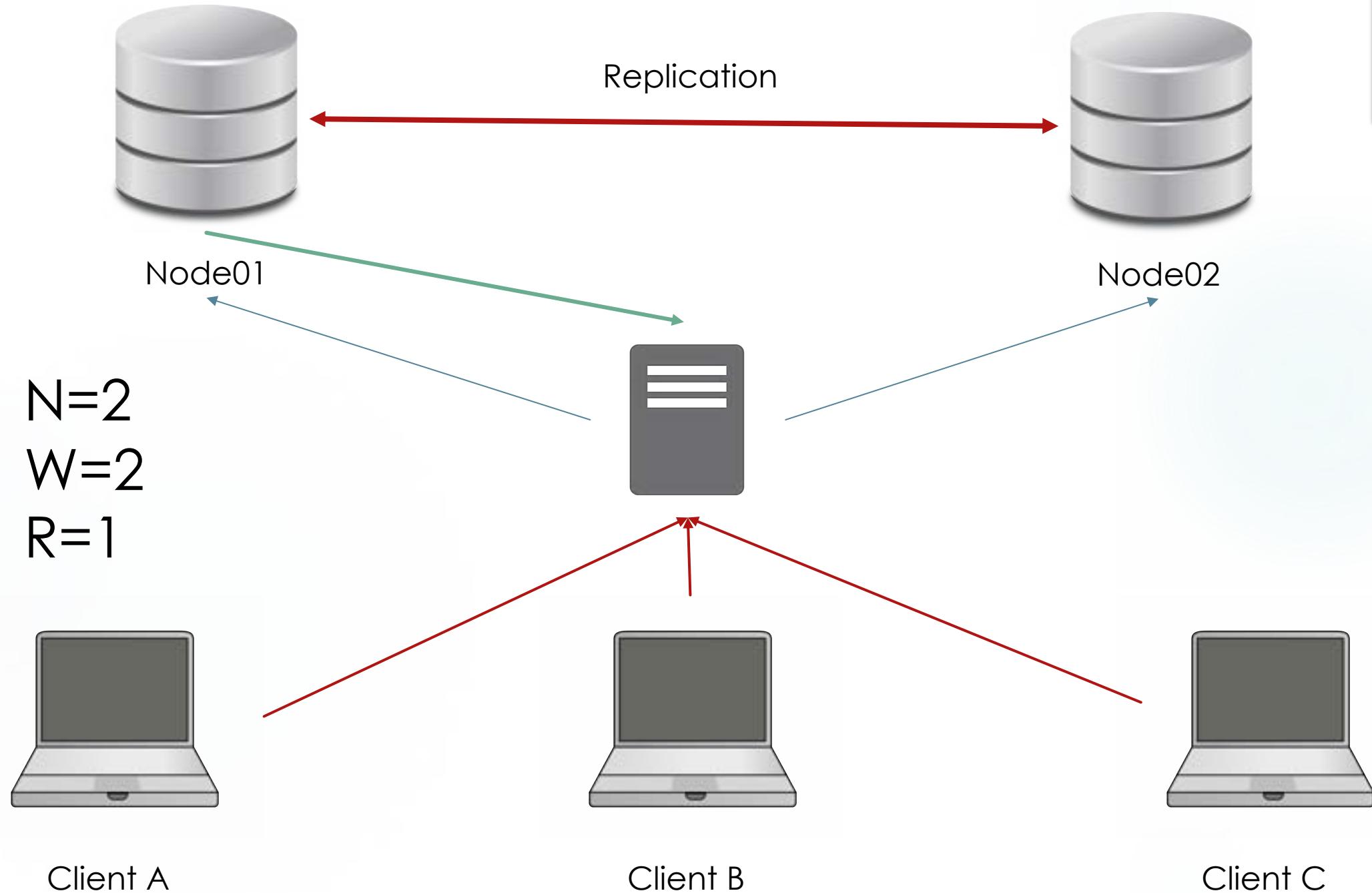
ON THE SERVER SIDE :

How do updates flow through the system ?



# Definitions

- ▶ Let:
  - ▶  $N$  = number of nodes replicating the data.
  - ▶  $W$  = number of replicas that need to acknowledge the receipt of the update before the update completes.
  - ▶  $R$  = the number of replicas that are contacted when a data object is accessed through a read operation.





Node01

Node02

Node03

$N=3$

$W=2$

$R=?$



Client A



Client B



Client C

# Scenario

- ▶ If  $W+R > N$ , then the write set and the read set always overlap and consistency is guaranteed.
- ▶ In the primary-backup RDBMS scenario, which implements synchronous replication,
  - ▶  $N=2$ ,  $W=2$ , and  $R=1$ .
  - ▶ No matter from which replica the client reads, it will always get a consistent answer.
- ▶ In asynchronous replication with reading from the backup enabled,
  - ▶  $N=2$ ,  $W=1$ , and  $R=1$ .
  - ▶ In this case  $R+W=N$ , and consistency cannot be guaranteed.

# Scenario

- ▶ The problem is that
  - ▶ when the system cannot write to  $W$  nodes because of failures, the write operation has to fail, marking the unavailability of the system.
- ▶ With  $N=3$  and  $W=3$  and only two nodes available, the system will have to fail the write.

# If High Availability paramount

- ▶ In distributed-storage systems that need to provide high performance and high availability,
- ▶ the number of replicas is in general higher than two.
- ▶ Systems that focus solely on fault tolerance often use  $N=3$  (with  $W=2$  and  $R=2$  configurations).
- ▶ Systems that need to serve very high read loads often replicate their data beyond what is required for fault tolerance;
  - ▶  $N$  can be tens or even hundreds of nodes,
  - ▶ with  $R$  configured to 1 such that a single read will return a result.
  - ▶ If  $W=N$  this decreases the probability of the write succeeding.
  - ▶ If  $W=1$  the update gets minimal durability and a lazy (epidemic) technique is used to update the other replicas.

# Configuring N, W and R

- ▶ Depends on what the common case is and which performance path needs to be optimized.
  - ▶  $R=1$  and  $N=W$  we optimize for the read case, and in
  - ▶  $W=1$  and  $R=N$  we optimize for a very fast write
  - ▶  $W=1$ ,  $R=1$   $N>2$  we optimise read/write but no consistency guarantee
  - ▶  $W < (N+1)/2$ , there is the possibility of conflicting writes when the write sets do not overlap.
- ▶ Weak/ eventual consistency arises when  $W+R \leq N$ 
  - ▶ So the read and write set may not overlap.
  - ▶ This can be done deliberately by setting R to 1.

# Network partitioning

- ▶ This happens when some nodes in the system cannot reach other nodes, but both sets are reachable by groups of clients.
  - ▶ If you use a classical majority quorum approach, then the partition that has  $W$  nodes of the replica set can continue to take updates while the other partition becomes unavailable. The same is true for the read set. Given that these two sets overlap, by definition the minority set becomes unavailable.
  - ▶ Partitions don't happen frequently, but they do occur between data centers, as well as inside data centers.

# When unavailability is not acceptable

- ▶ In this case both sides assign a new set of storage nodes to receive the data, and a merge operation is executed when the partition heals.
- ▶ E.g. Amazon's shopping cart
  - ▶ uses a write-always system;
  - ▶ in the case of partition, a customer can continue to put items in the cart even if the original cart lives on the other partitions.
  - ▶ The cart application assists the storage system with merging the carts once the partition has healed.

# Summary

- ▶ Eventual consistency vs strict consistency
- ▶ Client-side consistency vs server-side consistency
- ▶ Read/write optimisation with N, W and R

# Scaling a Postgres System

Systems and Database Analysis Week 10

# Tuning the Database



# Performance Issues Over Time



Database performance will decrease over time unless properly managed



Databases will become less responsive as the hardware hits its limits

# Dead Tuples

- Postgres uses MVCC to enforce isolation
- When we *delete* a row it isn't physically removed
- These are known as *dead tuples*
- Dead Tuples take up disk space and slow down table scans
- The VACUUM command removes dead tuples

# Query Planner (Analyze)



The query planner relies on statistics about the data



This goes out of date as the database is updated



Stale data -> sub-optimal queries



We can run the ANALYZE function to manually rebuild statistics

# Autovacuum

- Running these commands manually is impractical
- The autovacuum utility runs in the background vacuuming and analyzing the database
- Balancing Act: Maintain database vs Minimize Cleanup Impact

# Tuning Autovacuum

- Frequency of autovacuum is controlled by two main parameters
  - **autovacuum\_scale\_factor**
  - **autovacuum\_vacuum\_threshold**
- Vacuum runs when the number of dead tuples in a table exceeds

$$Threshold + (Scale\ Factor\ * RowCount)$$

- PgSQL defaults to threshold 50 and scale factor 0.2
- On a 1TB table this performs 200GB worth of clean-up!

# Side Note: VACUUM FULL

- Postgres provides the VACUUM FULL command
- This releases any memory reclaimed through the VACUUM command to the OS
- VACUUM FULL takes out an exclusive lock on the table
- This can bring a database down while the system waits for the operation to complete
- Approach with caution!

# Throttling

- Throttling forces the vacuum process to sleep intermittently when a certain limit of resources has been reached
- We can define how expensive an operation is using configuration
  - `vacuum_cost_page_hit = 1`
  - `vacuum_cost_page_miss = 10`
  - `vacuum_cost_page_dirty = 20`
- The `autovacuum_vacuum_cost_delay` parameter controls how long the process sleeps for
- The `autovacuum_vacuum_cost_limit` parameter controls how many *units* of work can be done before the process sleeps

# Tuning Guidelines

- The PgSQL defaults for tuning work well for small and medium tables, not for large tables
- Scale factor should be reduced for very large tables (can be set on a per-table basis)
- Throttling defaults were calibrated on older machines and should be increased
- All of these parameters can be set in postgresql.conf or configured on a per-table bases using ALTER TABLE

# Scaling the Hardware



# Theoretical Limits of Postgres

---

Database size: unlimited

---

Table Size: 16TB - 64TB

---

Field Size: 1GB

---

Columns in a Table: 250+

---

Row Size: No Limit

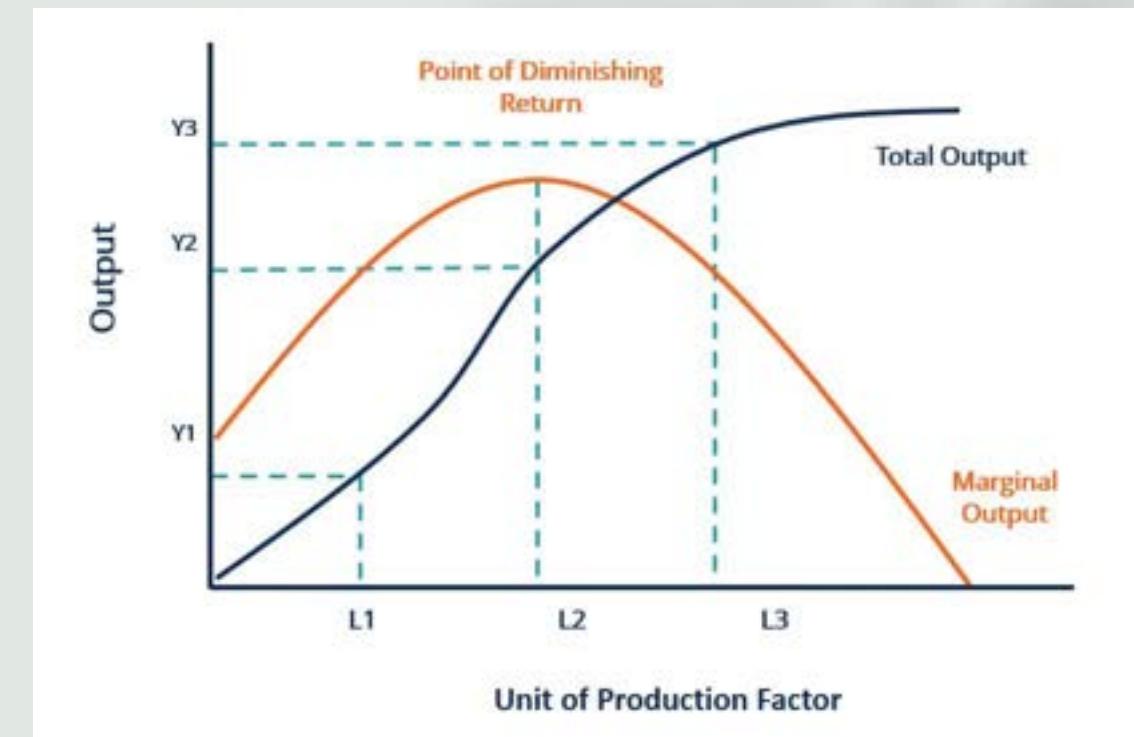
# Practical Limits of a System



- Theoretical limits rarely an issue.
- Practical limits imposed by the hardware in terms of
  - RAM
  - Disk space
  - Processing Power
  - Bandwidth
- Scaling a database usually involves scaling the hardware

# Diminishing Returns

- When buying a computer the *sweetspot* for value is somewhere in the middle
- The cost of each additional GB of RAM goes up as you add more to a system
- This is an example of the **law of diminishing returns**



# Scaling Up (**Vertical Scaling**)

- Scaling up increases the performance of a system by upgrading the hardware
- Works well for low-to-medium end systems
- Becomes expensive as requirements increase

# Scaling Out (**Horizontal Scaling**)

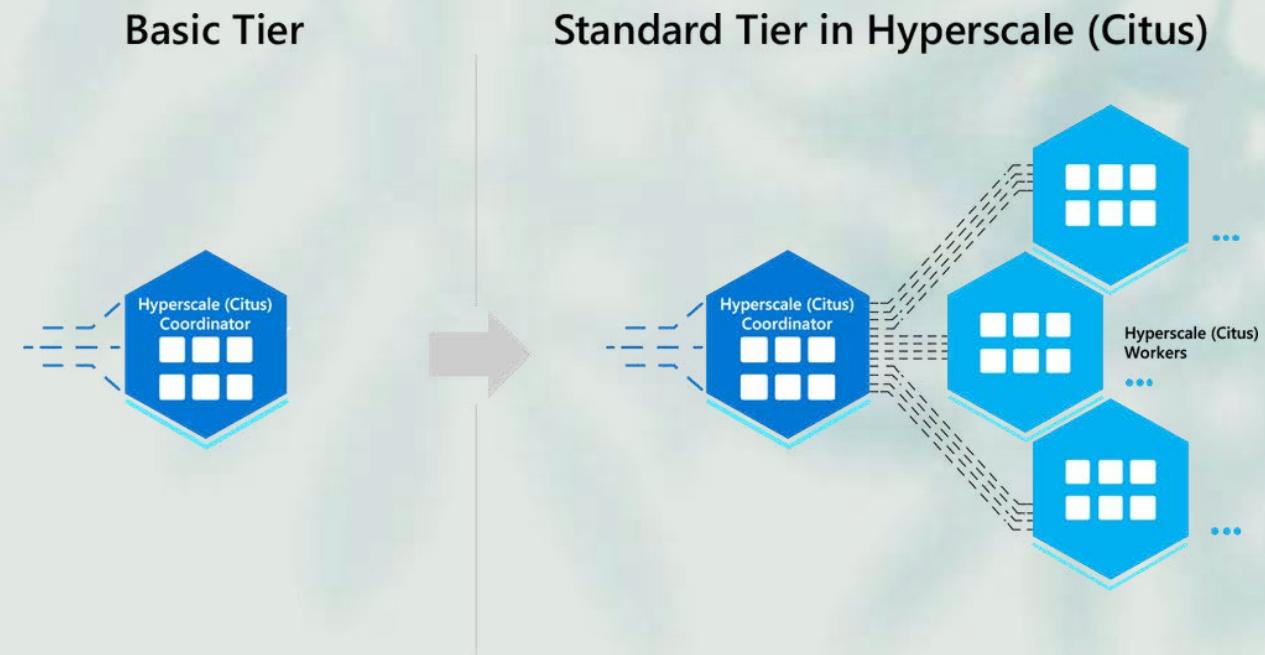
- Scaling out increases the performance of a system by adding additional machines (nodes)
- Adds to the system complexity and creates additional overheads
- Works well for systems with very high performance requirements

# Increasing Complexity

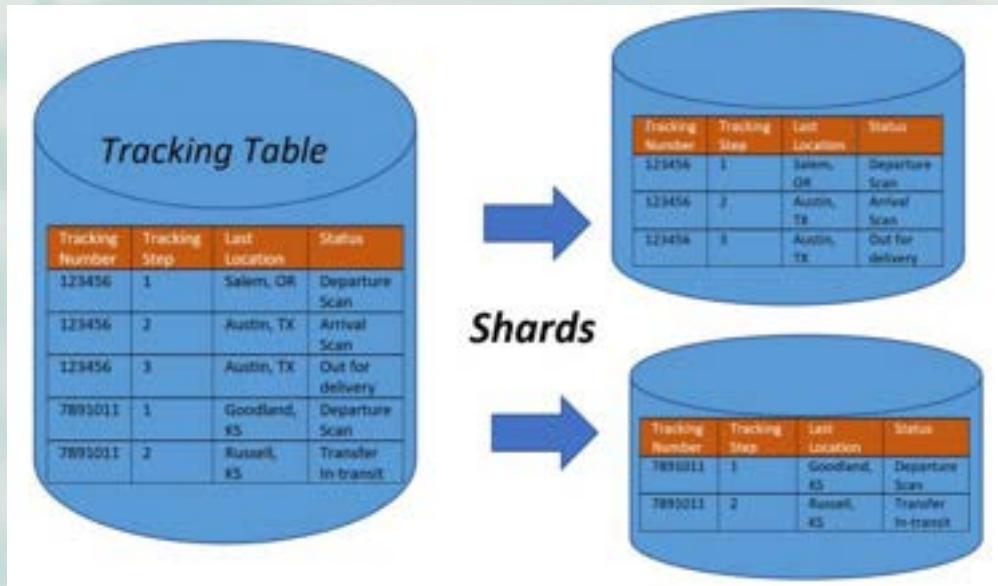
- Running a database on a single node is complex
- Additional complications running database on multiple nodes
  - Transaction management
  - Locating data
  - Network partitions

# Scaling out with Postgres: **Citus**

- Many commercial 3<sup>rd</sup> party solutions to use scale-out architecture with Postgres
- **Citus** is an open-source postgres extension which allows the creation of **distributed tables (shards)**
- Shards are essentially table partitions spread across multiple nodes



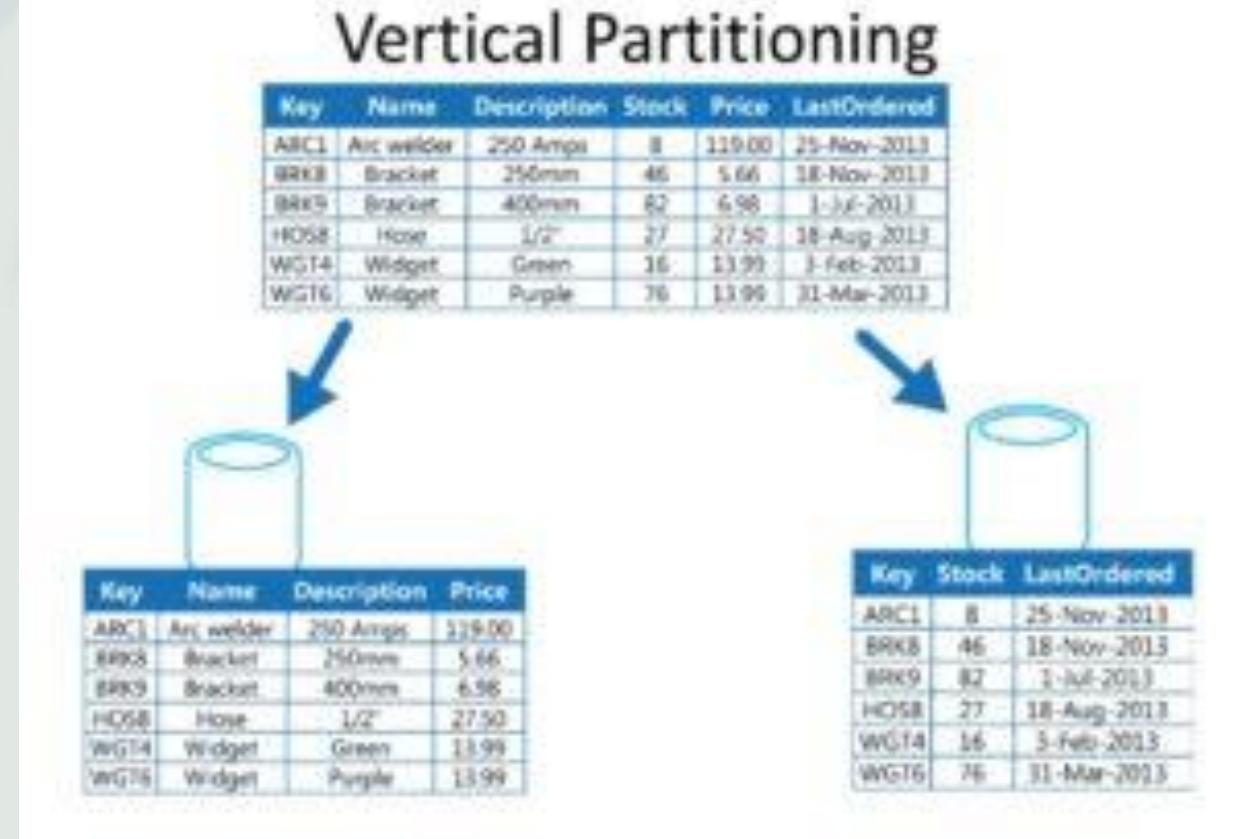
# Horizontal Sharding



- Most common sharding scenario
- Tables are split up and distributed by **row**
- Presented to the user as a single table
- Allows infinite rows to be added

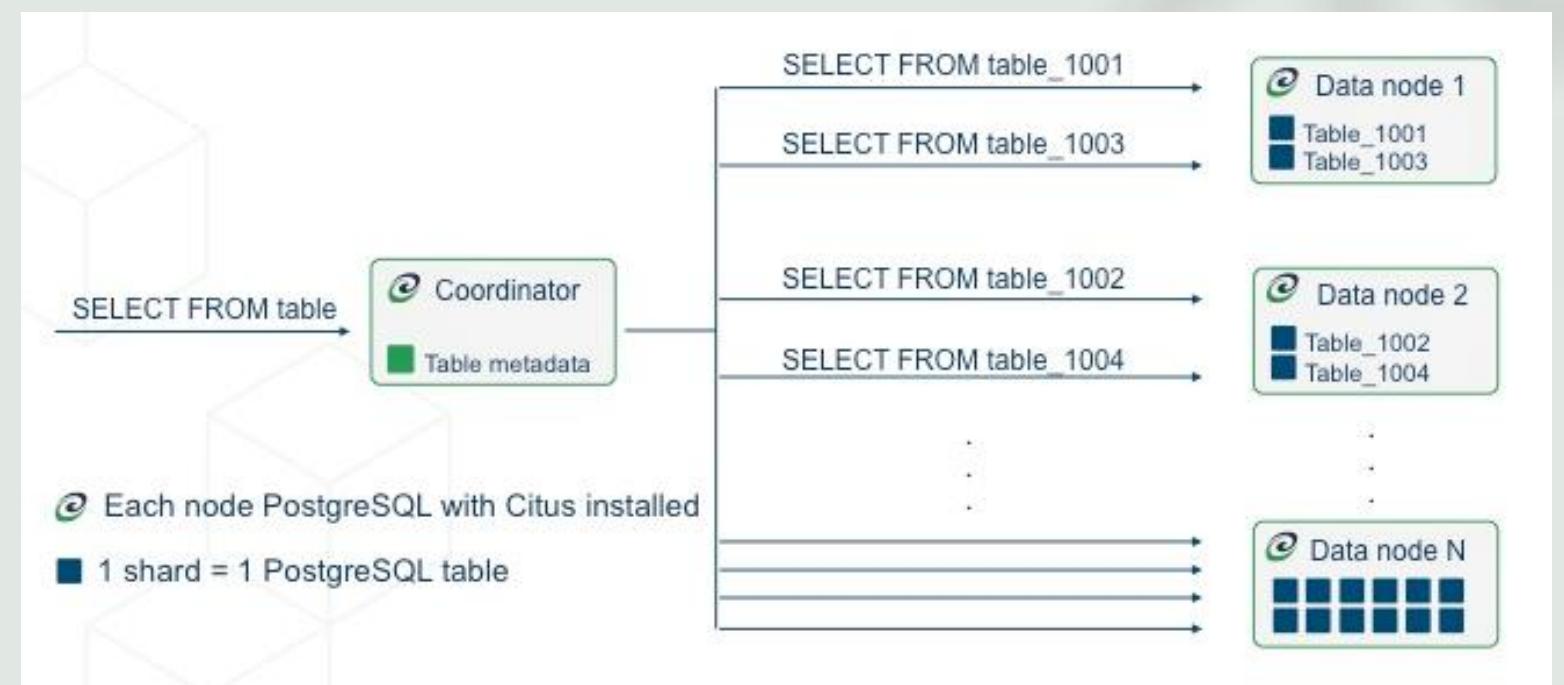
# Vertical Sharding

- Tables are split by column
- Useful when some columns are not used very often
- Each node must have space to hold all rows (for its columns)
- Not as common as horizontal sharding



# Citus Architecture

- Each cluster contains one special **co-ordinator** node
- The cluster takes all incoming queries and re-routes them to the **worker** nodes



# Parallelism: Reading Data

- Database queries involve some combination **reading data** and **computational processing**
- Data read from worker nodes needs to be sent over the network to the co-ordinator
- No gains from parallel processing

```
SELECT Salary, Role  
FROM Employee
```

# Parallelism: Processing Data

- Many analytical queries perform significant computations on the data
- Usually these queries take in lots of rows and output very few
- Processing happens on each node in parallel with very little data sent over the network
- Big gains from parallel processing

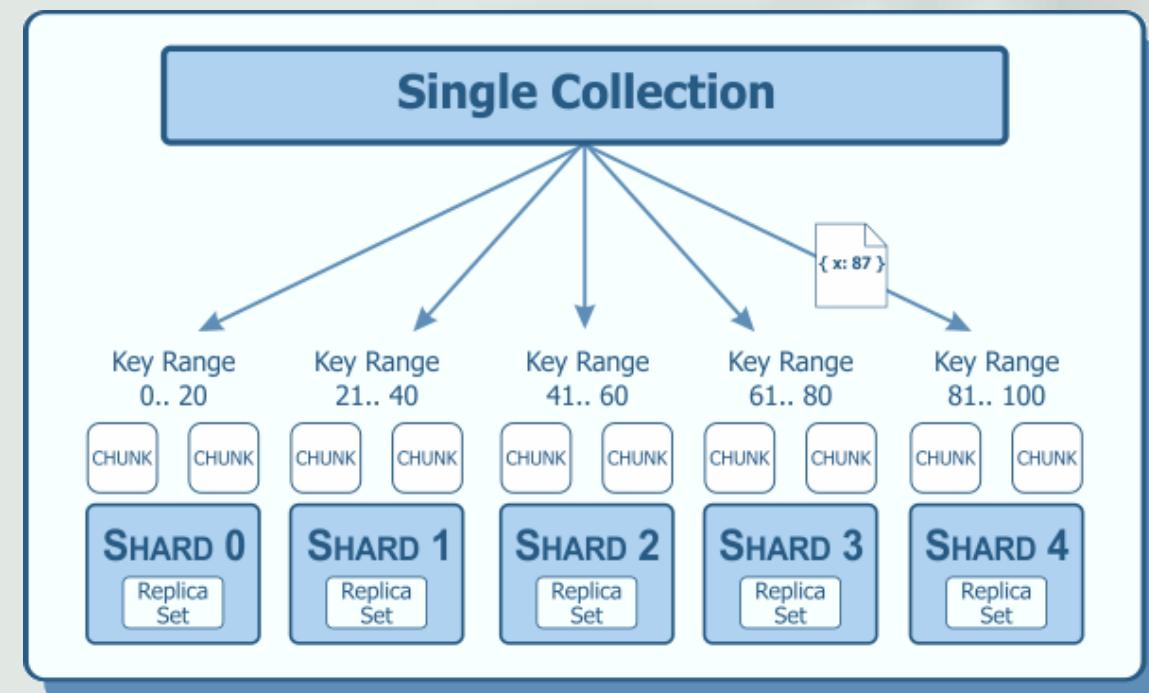
```
SELECT AVG(Salary)  
FROM Employee  
GROUP BY Role;
```

# Sharding Strategies

Fan-out-on-Read vs Fan-out-on-Write

# Shard Keys

- Sharding uses a **shard key** to determine which node will store a row
- Multi-shard operations are generally slower
- Choosing a good shard key is essential to an efficient system

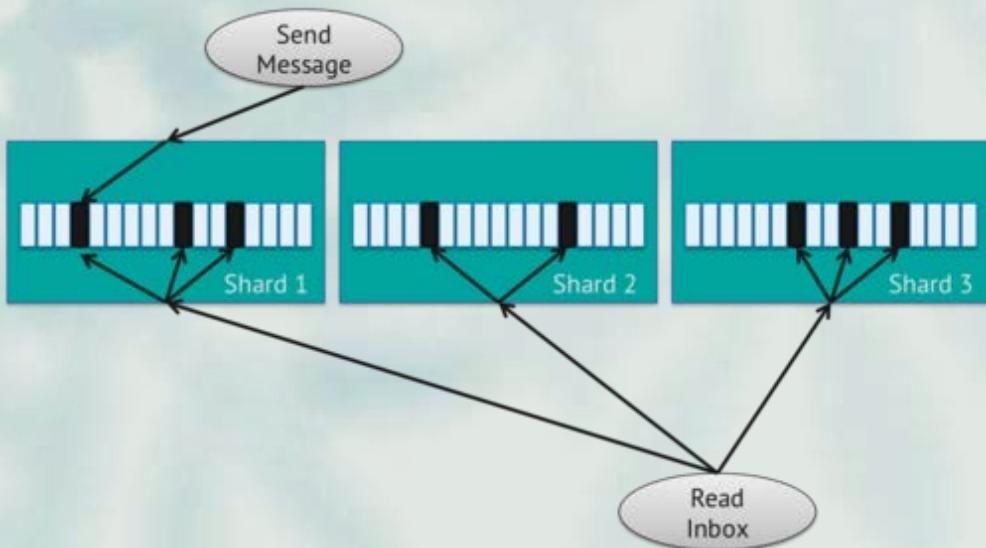


# Choosing a Shard Key: Messaging App

- We want our queries to read/write all of our data to a single node if possible
- The shard key determines whether we can do this
- Which column is most suitable for use as a shard key?

MessageID	SenderId	RecipientID	Content
10045309	10	5	Hi
10045310	14	26	Be right there
10045311	10	41	Sure thing
10045311	5	26	np

# Fanning Out



- We can usually write all of our data to a single shard, or read all of our data from a single shard, but not both
- Hitting multiple nodes is known as **fanning out**
- Fan-out-on-read means we write all of our data to a single node
- Fan-out-on-write means we read all of our data from a single node

# Choosing a Shard Key: Messaging App

- Which column results in fan-out-on-read?
- Which column results in fan-out-on-write?

MessagelD	SenderId	RecipientID	Content
10045309	10	5	Hi
10045310	14	26	Be right there
10045311	10	41	Sure thing
10045311	5	26	np

# Twitter: Fan-out-on-Write

- Tweets are read far more often than they're written
- When a tweet is created, multiple copies are created across multiple shards for each follower who will be notified
- This makes writing slow, but populating a timeline very fast



# Facebook: Fan-out-on-Read

- Facebook has more complex logic around visibility and privacy than twitter
- Facebook engineers found it impractical to work out who would see a post at the time it's created: all of that logic handled at read-time
- Facebook therefore went the other way and implemented fan-out-on-read



# Fan-out-on-read vs Fan-out-on-write

- Which is better? It depends
- Fan-out-on-write is more popular (a system usually requires more reads than writes)
- Fan-out-on-read can make more sense depending on the business case and context

The background of the image is a dense, abstract pattern of thin, glowing lines in various colors, primarily shades of pink, red, and orange, set against a dark, almost black, background. The lines are highly saturated and appear to be moving or flowing across the frame. A few thicker, more prominent lines in orange and yellow are visible, particularly on the left side.

Recap

# Recap – Performance Tuning

- Performance of a db degrades over time
- Dead tuples take up space and slow down table scans
- Query planner statistics go stale and need to be refreshed
- The autovacuum utility looks after this for us
- Needs to be configured to avoid major impact on DB performance
- Postgres defaults should be changed as hardware has improved

# Recap – Scaling Hardware

- When improving our hardware we can scale vertically or horizontally
- Vertical scaling adds resources to a single server
- Horizontal scaling adds additional (cheaper) nodes to the system
- Horizontal scaling is cheaper for systems with extremely high demands
- Horizontal scaling adds complexity and overhead
- The Citus extension allows us to horizontally scale our databases

# Recap

- Horizontal scaling usually involves sharding our tables
- Sharding is essentially distributed partitioning
- Sharding allows us to speed up certain processing-heavy queries through parallel execution
- Usually, however, it's best to keep queries to a single shard
- Fan-out-on-read and fan-out-on-write are two sharding strategies to achieve this aim