Getting Started with PostgreSQL

O'Reilly Online Live Training

👋 Hey

I'm **Haki Benita**

A software developer and a technical lead.

I'm interested in databases, web development, software design and performance tuning.

hakibenita.com

About Me

- Started as an Oracle DBA
- Worked on DWH and DSS systems
- Lead a team developers and DBAs
- Got into web development, mainly Python and Django
- Leading the development of a large ticketing platform

POLL

What is your job title?

- Developer
- Sysadmin / SRE
- Analyst / Data ops / Data scientist
- Student
- Management
- Other

POLL

What is your main use of data / SQL?

- Ad-Hoc Reports (sales reports, operational reports, etc.)
- Research / Analysis (BI / DWH / Dashboards/ Data exploration etc.)
- Development (Backend Development / ETL / Data pipelines etc.)

POLL

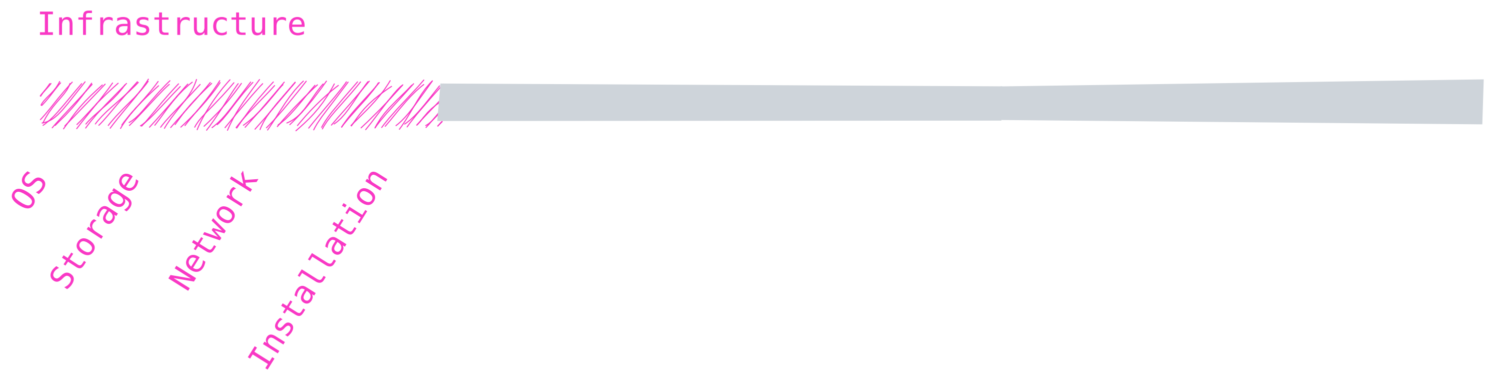
How would you rate your level of proficiency with databases?

- Novice
- Intermediate
- Advanced

The DBA Spectrum

Infrastructure

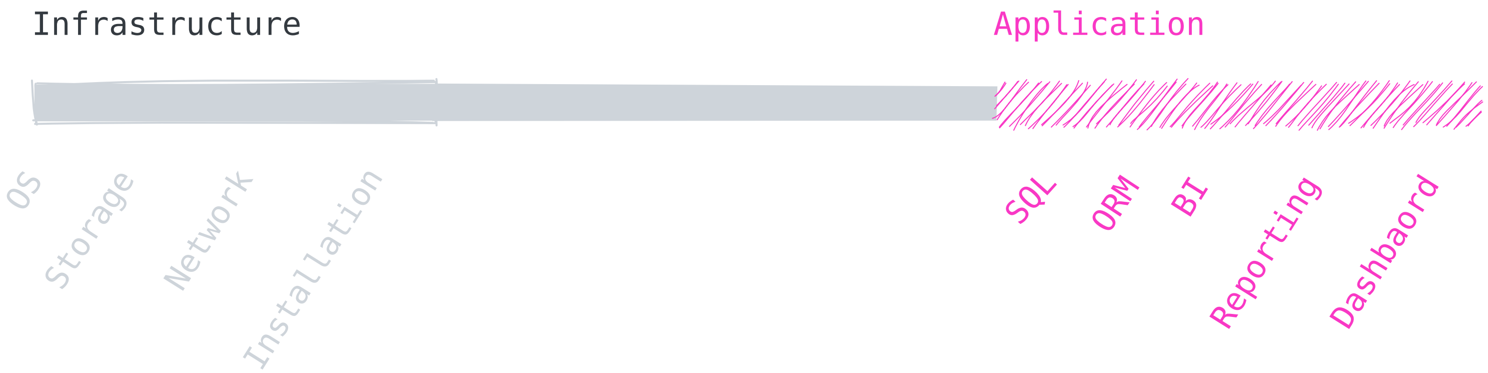
- Operating system, storage, network, installation...
- Infra DBA, SRE, Sysadmin



The DBA Spectrum

Application

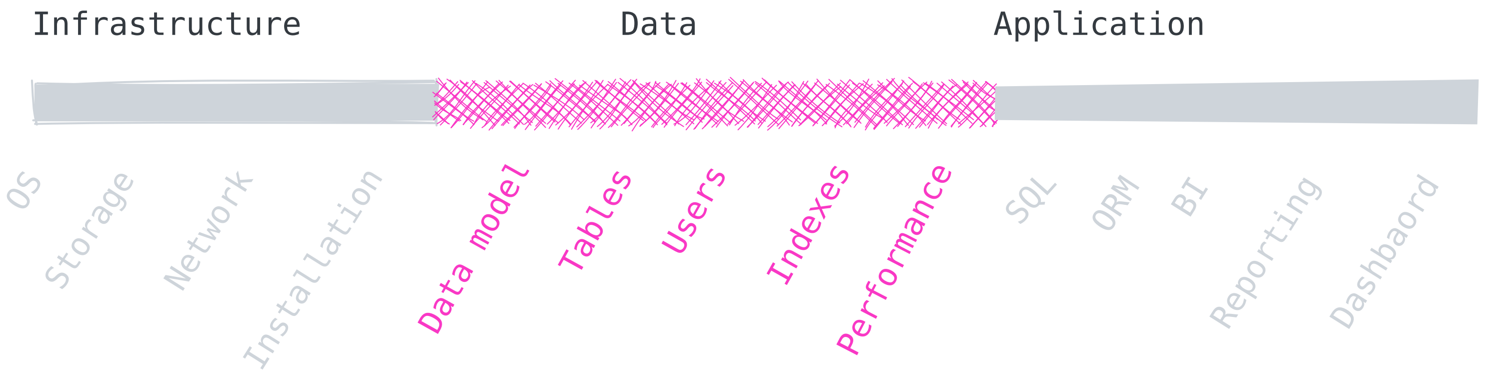
- SQL, ORMs, BI, dashboards, reporting...
- Developers, data scientists, analysts



The DBA Spectrum

Data

- Data model, tables, indexes, users, performance tunning
- Application DBAs, developers, data ops



What you’ll gain from this training

- Get comfortable with PostgreSQL
- Perform basic administrative tasks
  - Create and manage users and permissions
  - Create and manage tables
  - Evaluate query performance
  - Create indexes to speed up query execution

Just the tip of the iceberg...

> “ *Give a man a fish and you feed him for a day*
> *Teach a man to fish and you feed him for a lifetime* ”

In the process you will also

- Get comfortable with **PostgreSQL CLI and documentation**
- Figure out how to **find answers on your own**
- Evaluate different aspects of **database performance**

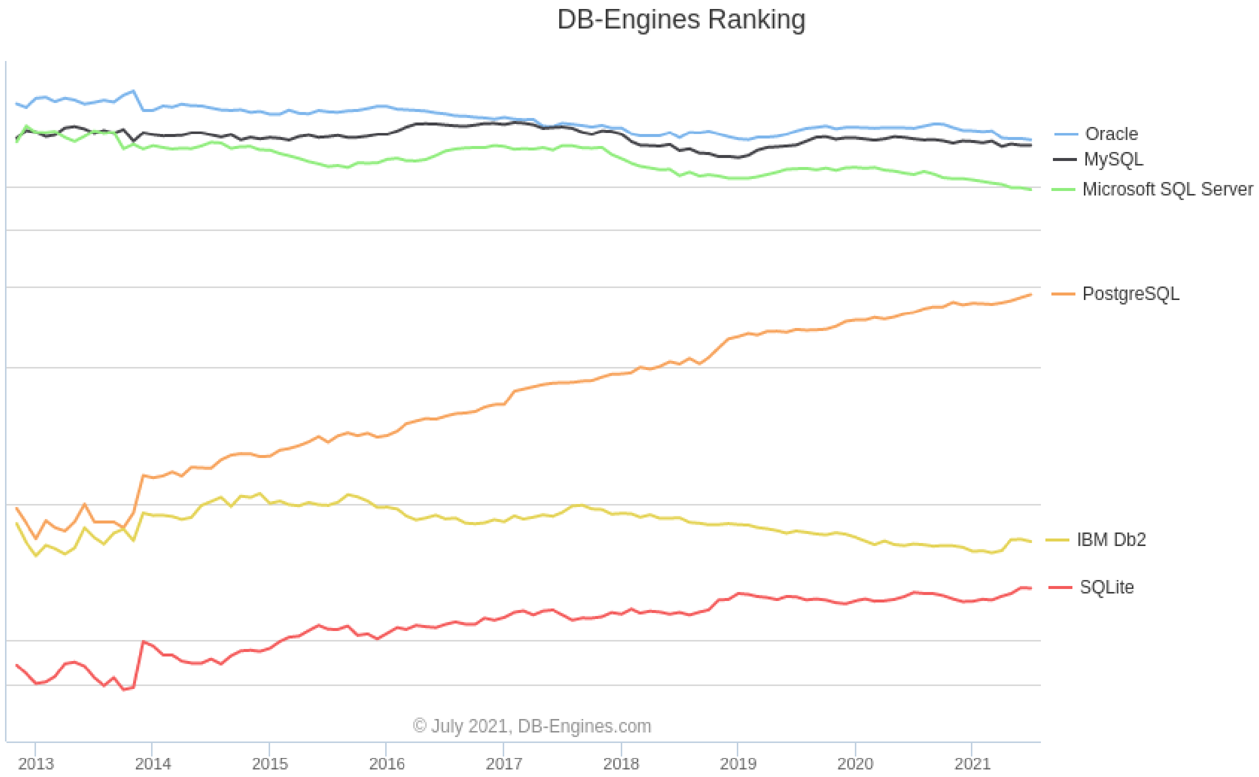
History

SQL

- **S**tructured **Q**uery **L**anguage
- Used for interacting with relational databases (RDBMS)
- Invented in the early 70s at IBM based on work by Edgar F. Codd
- Became a standard in 1986 (ANSI-86)
- Standard revised in 1992 (ANSI-92)

PostgreSQL

- Based on the Berkeley POSTGRES project from 1986
- Originally named Postgres95
- Free Open Source
- Growing fast!



Tools

PostgreSQL Documentation

- Excellent resource
- Suited for both experts and beginners
- Use it! Bookmark it!

https://www.postgresql.org/docs/current/

psql

- PostgreSQL interactive terminal
- Comes with PostgreSQL installation

```
$ psql
psql (13.2)
Type "help" for help.

postgres=#
```

https://www.postgresql.org/docs/13/app-psql.html

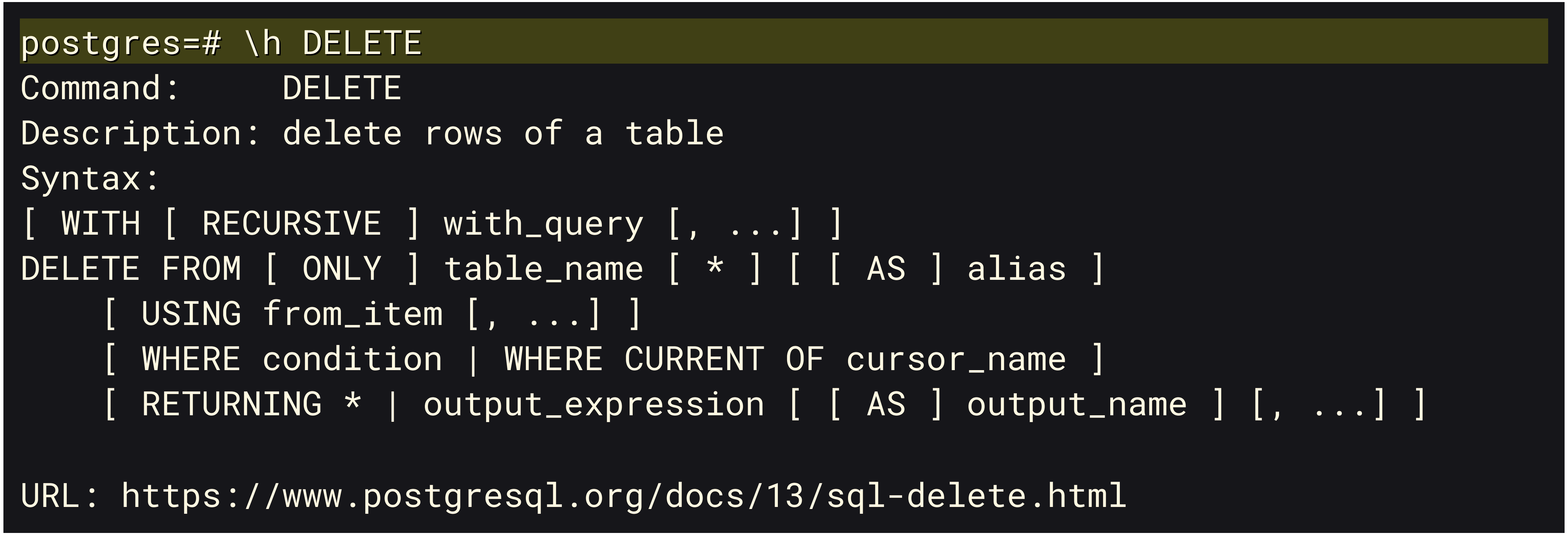
Don't memorize anything!

List all commands

```
postgres=# \?
General
  \copyright             show PostgreSQL usage and distribution terms
  \crosstabview [COLUMNS] execute query and display results in crosstab
  \errverbose            show most recent error message at maximum verbosity
  \g [(OPTIONS)] [FILE]  e
  ....
```

Don't memorize anything!

Get command syntax



```
postgres=# \h DELETE
Command:     DELETE
Description: delete rows of a table
Syntax:
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]

URL: https://www.postgresql.org/docs/13/sql-delete.html
```

Don't memorize anything!

Use auto complete

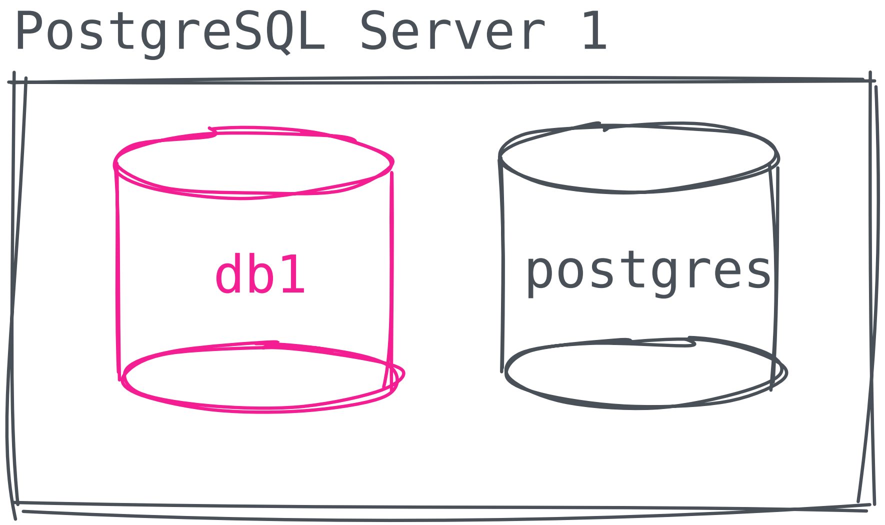
```
postgres=# \set E<tab>
ECHO          ECHO_HIDDEN          ENCODING
```

Also works with column and table names

Architecture

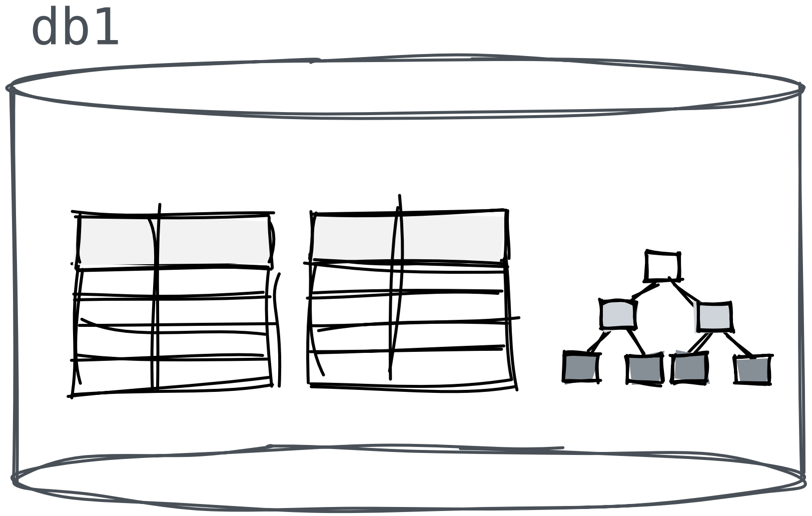
Cluster

- PostgreSQL installation
- Define user / roles and their permissions
- Can contain multiple named databases



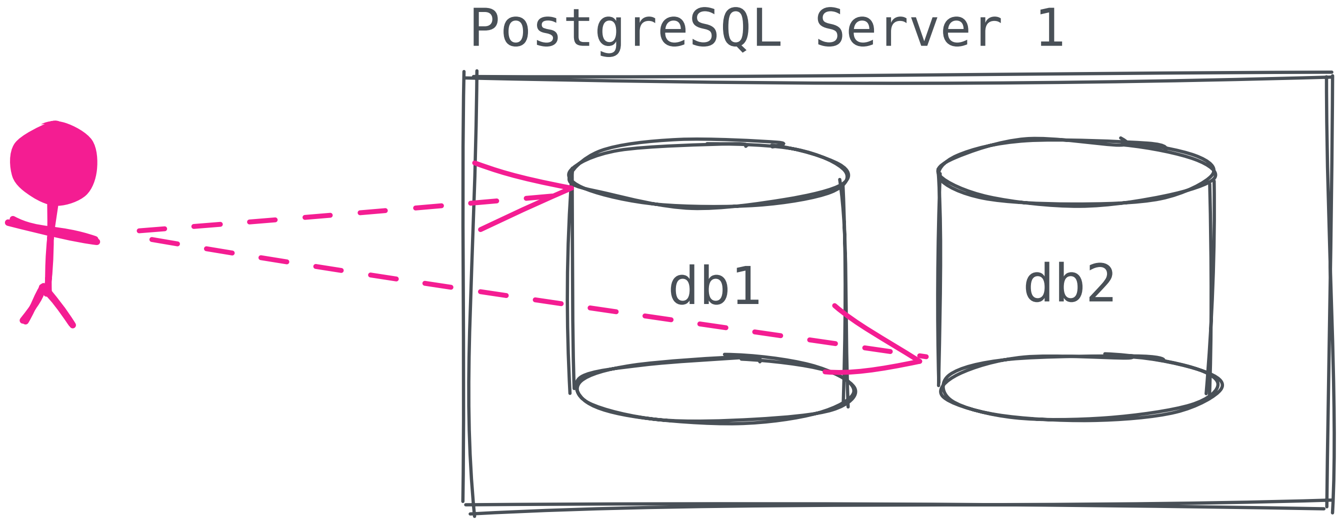
Database

- Has a designated storage area on the file system
- Contains database objects and definitions such as tables, indexes and sequences
- Owned by a user



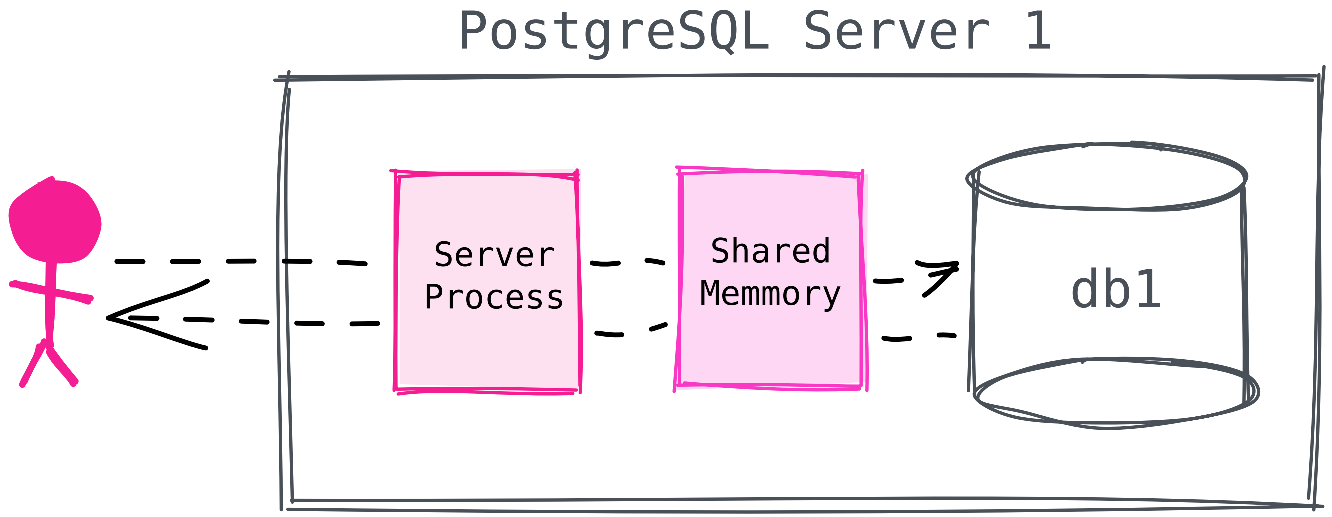
User

- Granted object and system permissions
- Used to authenticate a client application



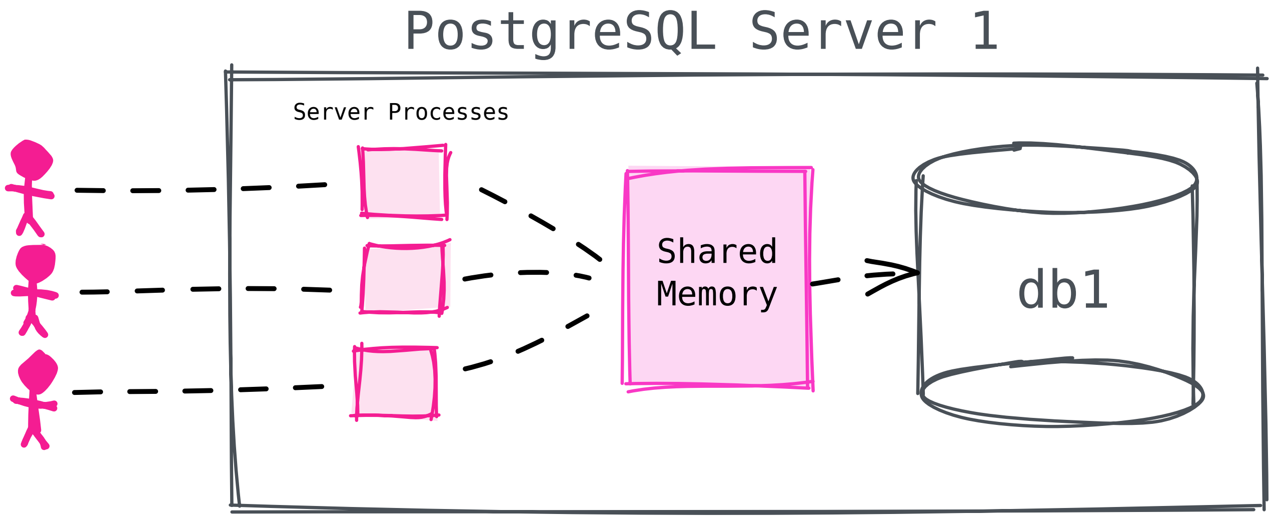
Client Server

- Client application establish connection with database server
- Database server allocate a server process for the session
- Can have many concurrent active connections (`show max_connections`)



Memory

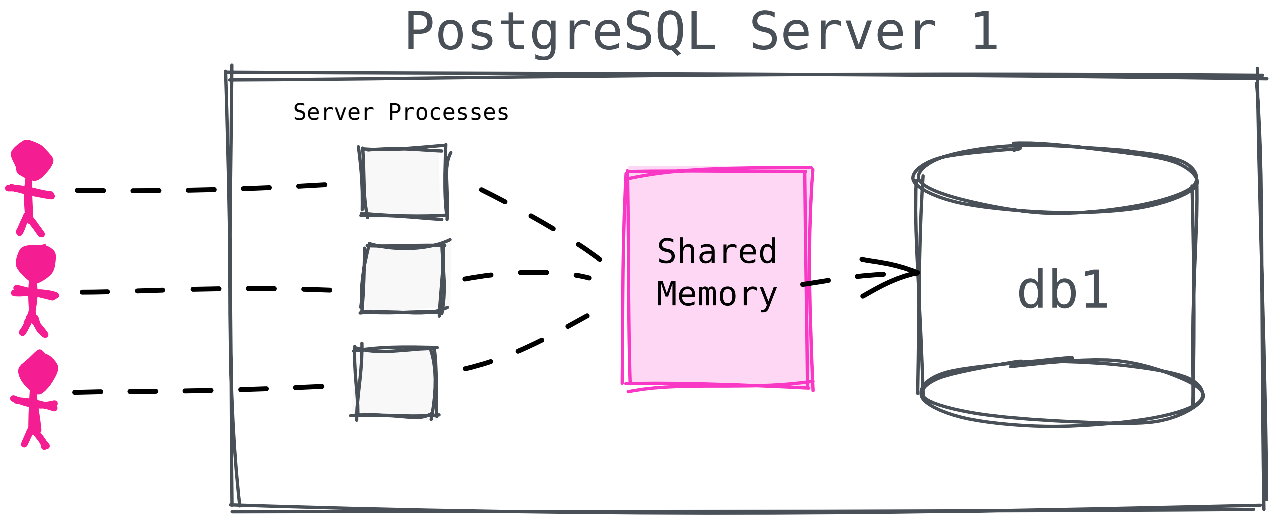
- **Shared memory**: Shared by all server processes
- **Local memory**: Used by a single server process



Memory

Shared Buffers

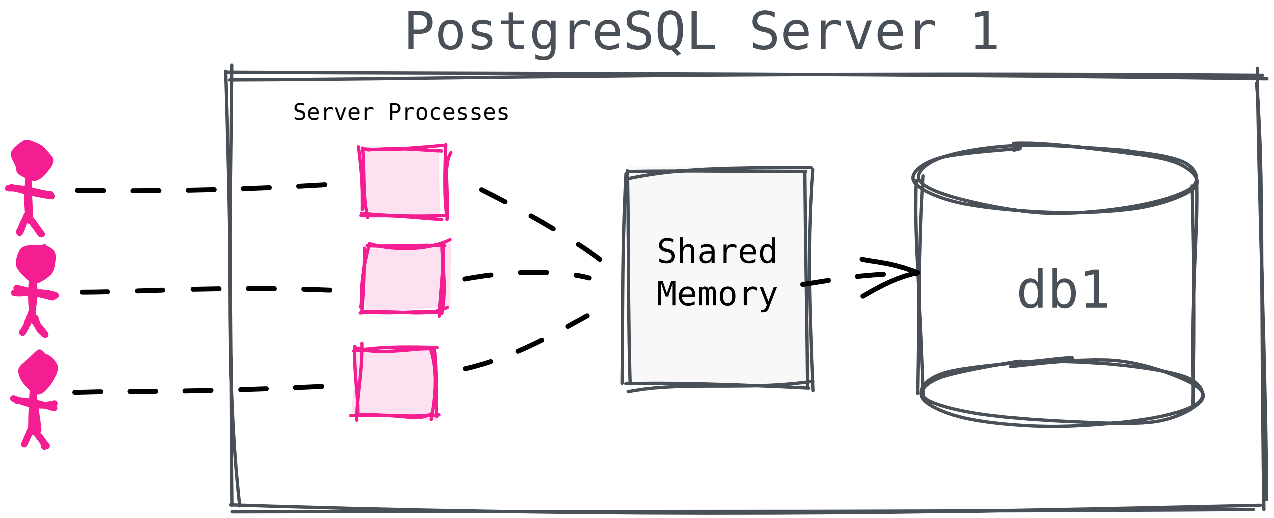
- Keep frequently accessed objects in memory for fast retrieval
- Shared by all server processes
- Default 128MB
- `show shared_buffers`



Memory

Work Mem

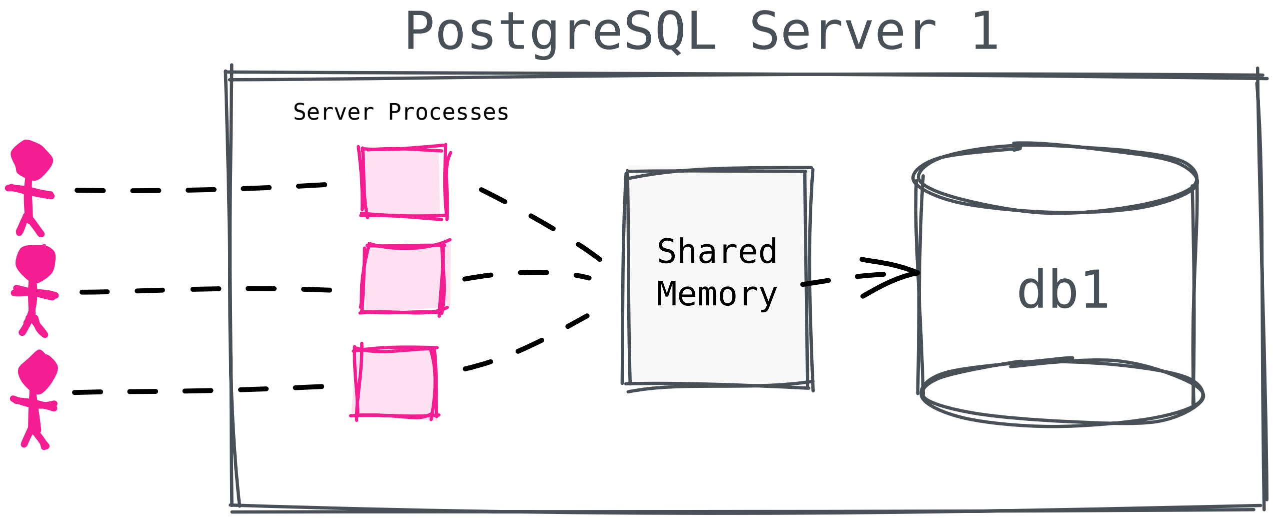
- Allocated for each backend process
- Used for sorts and hash tables
- Default 1MB
- `show work_mem`



Memory

Temp Buffers

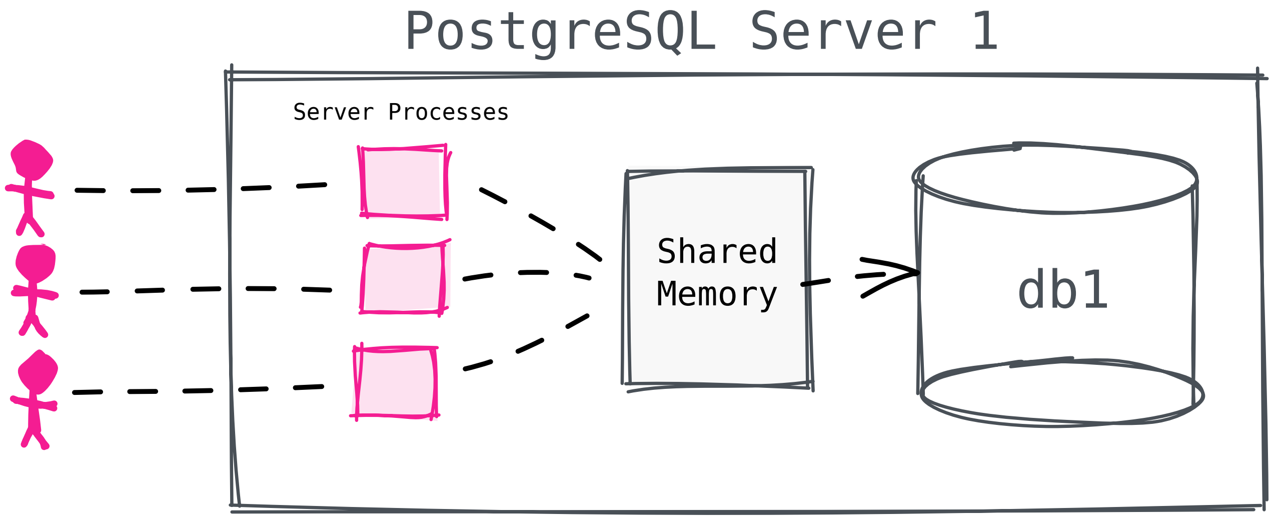
- Allocated for each backend process
- Used for storing temporary tables
- Default 8MB
- `show temp_buffers`



Memory

Maintenance Work Memory

- Allocated for each backend process
- Used for vacuum and create index operations
- Default 64MB
- `show maintenance_work_mem`



Write Ahead Log (WAL)

- A log of all changes to tables and indexes
- Used to restore the database in case of disaster
- Can be used to maintain replication
- Enables point in time recovery

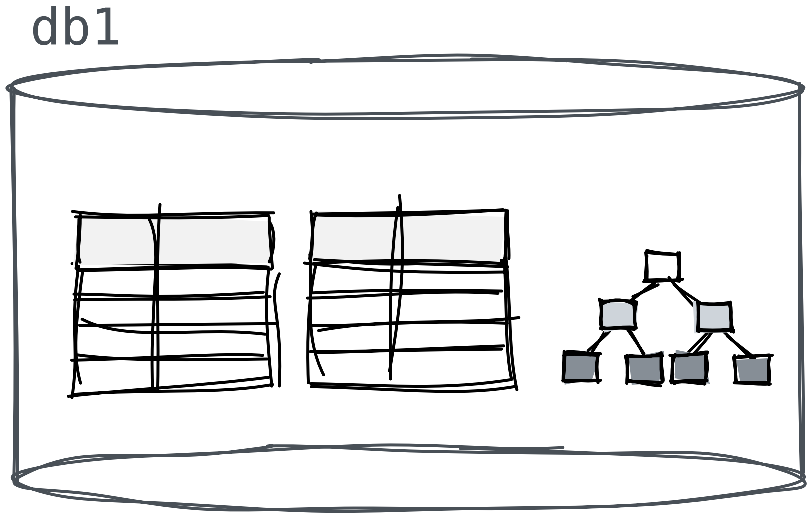
Recap

- Architecture and Terminology
    - Cluster -> database -> schema -> table
- Memory structures
    - Shared memory
    - Local memory
- WAL

Database

Database

- Contains database objects and definitions such as tables, indexes and sequences
- Owned by a user



Database

Creating a database

```
postgres=# CREATE DATABASE db1 OWNER postgres;
CREATE DATABASE

postgres=# \connect db1
You are now connected to database "db1" as user "postgres".
db1=#
```

Database

Creating a database from the terminal

```
$ createdb db1 -O postgres
```

- `-O`: database owner

Database

List databases

```
postgres=# \l
        List of databases
      Name      │     Owner     │ Encoding
────────────────┼───────────────┼──────────
 db1            │ postgres      │ UTF8
 postgres       │ postgres      │ UTF8
 template0      │ postgres      │ UTF8
 template1      │ postgres      │ UTF8
```

Table

- Data in relational database is stored in tables
- Tables have columns
- Can define constraints
- Use indexes to speed access to data stored in tables

Table

Creating a table

```
db1=# CREATE TABLE users (
  id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  active BOOLEAN,
  name TEXT
);

CREATE TABLE
```

Table

Inspecting a table

```
db1=# \d users
                             Table "public.users"
 Column │  Type   │ Nullable │           Default
────────┼─────────┼──────────┼──────────────────────────────
 id     │ integer │ not null │ generated always as identity
 active │ boolean │          │
 name   │ text    │          │

Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
```

Table

Listing all tables

```
db1=# \dt
       List of relations
 Schema │ Name  │ Type  │ Owner
────────┼───────┼───────┼──────────
 public │ users │ table │ postgres
```

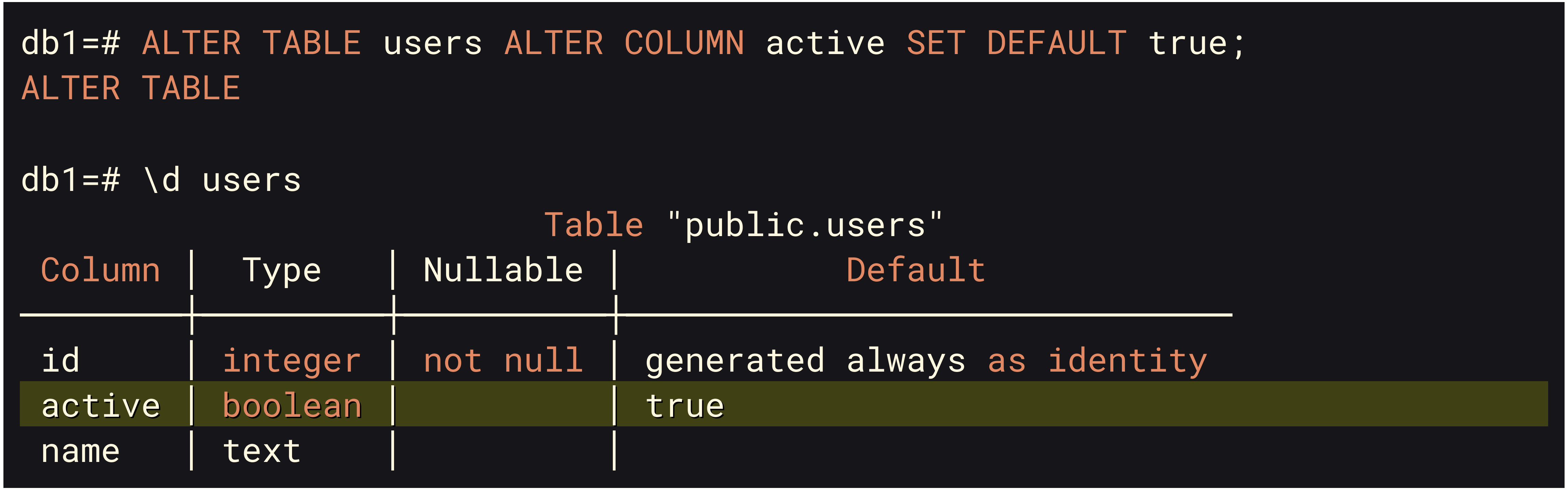
Table

Searching for tables

```
db1=# \dt foo*
Did not find any relation named "foo*"

db1=# \dt u*
       List of relations
 Schema │ Name    │ Type    │ Owner
────────┼─────────┼─────────┼──────────
 public │ users   │ table   │ postgres
```

Table

Alter an existing table



```
db1=# ALTER TABLE users ALTER COLUMN active SET DEFAULT true;
ALTER TABLE

db1=# \d users
                          Table "public.users"
 Column │  Type   │ Nullable │           Default
────────┼─────────┼──────────┼──────────────────────────────
 id     │ integer │ not null │ generated always as identity
 active │ boolean │          │ true
 name   │ text    │          │
```

Table

Default value is used when not explicitly provided

```
db1=# INSERT INTO users (name) VALUES ('Haki Benita');
INSERT 0 1

db1=# SELECT * FROM users;
 id | active | name
----+--------+------------
  1 | t      | Haki Benita
```

Good defaults can prevent errors and confusion!

View

- A named query
- Results are not materialized

View

Creating a view

```
db1=# CREATE VIEW active_users AS
  SELECT id, name
  FROM users
  WHERE active;

CREATE VIEW
```

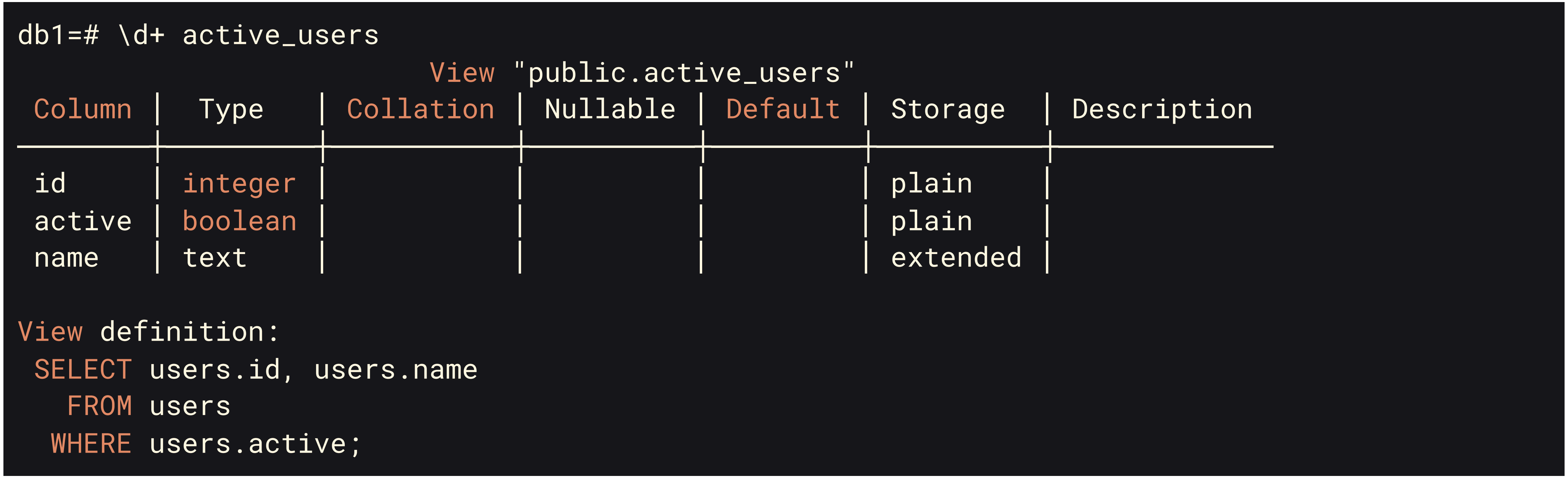
View

Querying a view

```
db1=# INSERT INTO users (name, active) VALUES ('Bob Bar', false);
INSERT 0 1

db1=# SELECT * FROM active_users;
 id │ name
────┼────────────
  1 │ Haki Benita
(1 row)

db1=# SELECT * FROM users;
 id │ active │ name
────┼────────┼────────────
  1 │ t      │ Haki Benita
  2 │ f      │ Bob Bar
(2 rows)
```

View

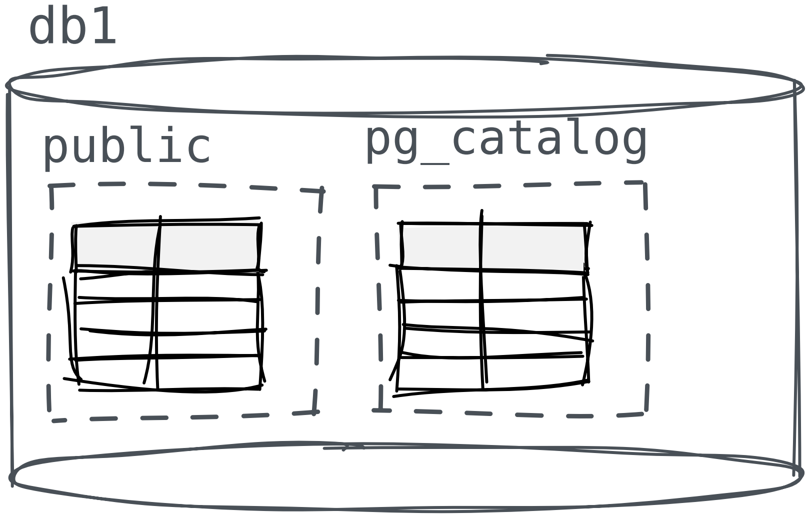
View details



```
db1=# \d+ active_users
                         View "public.active_users"
 Column │  Type   │ Collation │ Nullable │ Default │ Storage  │ Description
────────┼─────────┼───────────┼──────────┼─────────┼──────────┼─────────────
 id     │ integer │           │          │         │ plain    │
 active │ boolean │           │          │         │ plain    │
 name   │ text    │           │          │         │ extended │

View definition:
 SELECT users.id, users.name
   FROM users
  WHERE users.active;
```

Schema

- A namespace within the database
- Database can use multiple schemas
- The default schema is called "public"



Schema

Creating a schema

```
db1=# CREATE SCHEMA restricted;
CREATE SCHEMA

db1=# \dn
   List of schemas
   Name     │  Owner
────────────┼──────────
 restricted │ postgres
 public     │ postgres
```

Schema

Creating tables in a schema

```
db1=# CREATE TABLE restricted.credentials (
  id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  user_id INT,
  password TEXT NOT NULL
);
CREATE TABLE

db1=# SELECT * FROM restricted.credentials;
 id │ user_id │ password
────┼─────────┼──────────
(0 rows)
```

Schema

Referencing objects by name

Name	Pattern	Example
qualified	*database.schema.table*	`db1.restricted.credentials`
qualified	*schema.table*	`restricted.credentials`
unqualified	*table*	`credentials`

Schema

Schema search path

```
db1=# SHOW search_path;
   search_path
-----------------
 "$user", public
```

- The schema order to look for unqualified object names
- `"$user"`: name of the current user
- `public`: the default schema

Schema

Schema search path

```
db1=# SELECT * FROM credentials;
ERROR:  relation "credentials" does not exist

db1=# SET search_path TO restricted, "$user", public;
SET

db1=# SELECT * FROM credentials;
 id │ user_id │ password
────┼─────────┼──────────
(0 rows)
```

Schema

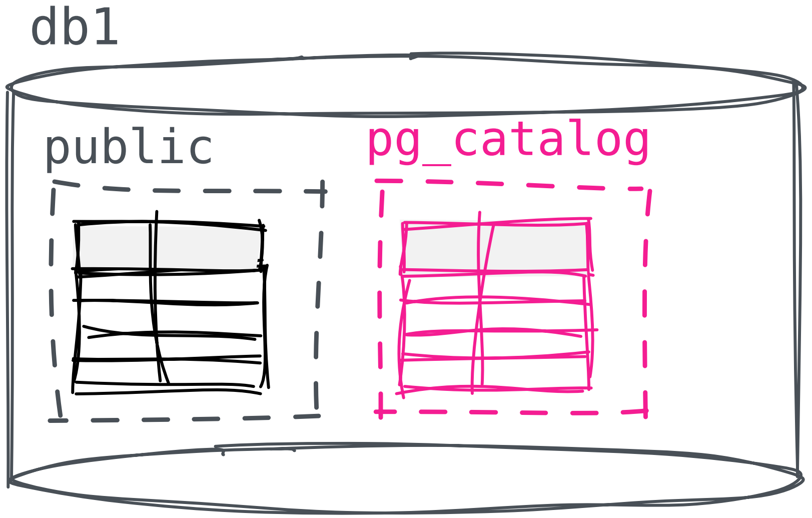
Create objects with similar names in different schemas:

- Can act as a synonym
- Can be used for multi-tenancy, online data migrations

📖 "Synonyms" in PostgreSQL

Information Schema

- A special schema in every database "pg_catalog"
- Contains information about database objects



Information Schema

Information about tables

```
db1=# SELECT * FROM pg_tables WHERE tablename = 'users';
─[ RECORD 1 ]─────────
schemaname  │ public
tablename   │ users
tableowner  │ postgres
tablespace  │ ¤
hasindexes  │ t
hasrules    │ f
hastriggers │ t
rowsecurity │ f
```

Information Schema

Information about indexes

```
db1=# SELECT * FROM pg_indexes WHERE tablename = 'users';
-[ RECORD 1 ]-----------------------------------------------------------
schemaname | public
tablename  | users
indexname  | users_pkey
tablespace | ¤
indexdef   | CREATE UNIQUE INDEX users_pkey ON public.users USING btree (id)
```

Information Schema

Information about all database objects:

```
db1=# SELECT relname, relkind FROM pg_class WHERE relname LIKE '%users%';
   relname     | relkind
---------------+---------
 users         | r
 users_pkey    | i
 users_id_seq  | S
 active_users  | v
```

`r`=table, `i`=index, `S`=sequence, `v`=view, more...

Information Schema

Other useful tables

- `pg_stat_activity`: information about current activity
- `pg_stat_all_table`: table access statistics
- `pg_stat_all_indexes`: index access statistics
- `pg_stats`: statistics about table columns

Recap

- Create database, schema, tables and views
- Referencing objects using qualified and unqualified names
- Schema search path
- Inspect and list database objects using psql
- Using the information schema (catalog)

Exercise

COPY

- Export data to text, CSV or binary format
- Import data from text, CSV or binary format into tables
- Produce reports from query results

Exercise

Getting Started with PostgreSQL: Import and export data using COPY

Launch Katacoda scenario »

Import From CSV

```
postgres=# \COPY users FROM /tmp/users.csv WITH CSV HEADER
COPY 3

postgres=# \COPY users (active,name,id) FROM /tmp/users2.csv WITH CSV HEADER
COPY 2

postgres=# SELECT * FROM users;
 id | active |     name
----+--------+---------------
  1 | t      | Haki Benita
  2 | f      | Bob Bar
  3 | t      | Ben Smith
 10 | t      | Andres Lane
 11 | f      | Jonathan Lane
```

Export Table to CSV

```
postgres=# \COPY users TO /tmp/all_users.csv WITH CSV HEADER
COPY 5

postgres=# \! cat /tmp/all_users.csv
id,active,name
1,t,Haki Benita
2,f,Bob Bar
3,t,Ben Smith
10,t,Andres Lane
11,f,Jonathan Lane
```

Export Query Results to CSV

```
postgres=# \COPY (SELECT id, name FROM users WHERE active) TO /tmp/active_users.csv WITH CSV HEADER
COPY 3

postgres=# \! cat /tmp/active_users.csv
id,name
1,Haki Benita
3,Ben Smith
10,Andres Lane
```

COPY vs. \COPY

The COPY command has two variations:

- `COPY`: executed on the server
- `\COPY`: psql command with similar api, executed on the client

What you usually want is `\COPY`

COPY Trick

You want to produce a report from a big query

```sql
SELECT u.id, u.name, count(*) AS credentials
FROM users u
  LEFT JOIN restricted.credentials c ON u.id = c.user_id
WHERE u.active
GROUP BY 1, 2
ORDER BY 3 DESC;
```

COPY Trick

`\COPY` command does not support multiple lines

```
db1=# \copy (SELECT u.id, u.name, count(*) AS credentials <enter>
\copy: parse error at end of line
```

COPY Trick

`COPY` can accept multiple lines

```
db1=# COPY (SELECT u.id, u.name, count(*) AS credentials
FROM users u LEFT JOIN restricted.credentials c ON u.id = c.user_id
WHERE u.active
GROUP BY 1, 2
ORDER BY 3 DESC) TO STDOUT WITH CSV HEADER;
id,name,credentials
1,Haki Benita,1
...
```

But you want to produce the report to a file, like `\copy` does...

COPY Trick

`psql` has an option to send query output to file

```
db1=# \?
  ...
  \g [(OPTIONS)] [FILE]
    execute query (and send results to file or |pipe);
  ...
```

COPY Trick

Use `\g` to write `COPY` output to a local file!

```
db1=# COPY (SELECT u.id, u.name, count(*) AS credentials
FROM users u LEFT JOIN restricted.credentials c ON u.id = c.user_id
WHERE u.active
GROUP BY 1, 2
ORDER BY 3 DESC)
TO STDOUT WITH CSV HEADER \g report.csv

COPY 4
```

📖 Use \copy With Multi-line SQL

Data Integrity

Database as the source of truth

Column Types

The first line of defence

```
db1=# INSERT INTO users (name, active) VALUES ('foo', 'not a boolean');
ERROR:  invalid input syntax for type boolean: "not a boolean"
```

Cannot insert invalid values!

Column Types

Common types

- varchar(N), text
- smallint, integer, bigint
- decimal, real, double precision
- date, timestamp, timestamptz
- bytea
- boolean

Column Types

Special types

- uuid
- jsonb
- range
- array
- more...

Column Types

Set restrictions that make sense

```
db1=# ALTER TABLE users ALTER name TYPE VARCHAR(20);
ALTER TABLE

db1=# INSERT INTO users (name) VALUES ('probably invalid name');
ERROR: value too long for type character varying(20)
```

📖 Postgres: Boundless text and Back Again

Constraints

- Maintain data integrity
- Keeps the data clean
- Complements types

Not Null Constraint

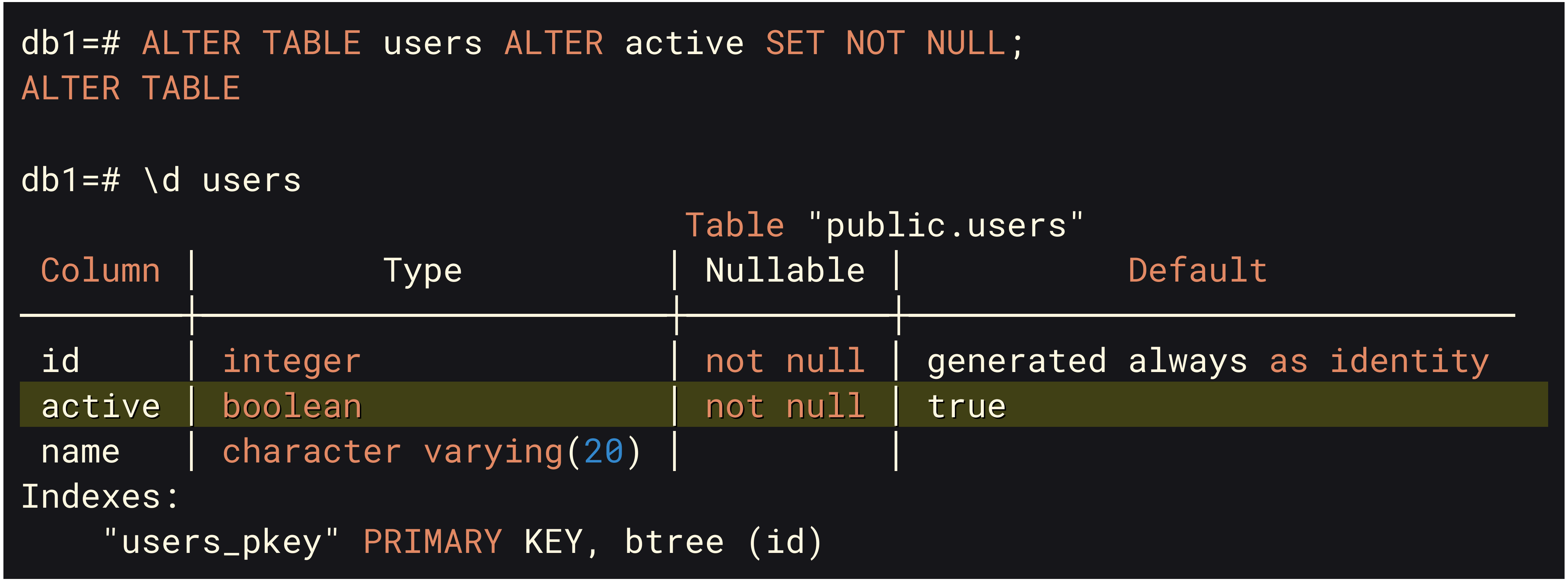
Make fields required

Not Null Constraint

`NULL` is a special values that indicates "missing value"

Not Null Constraint

Add a "not null" constraint on an existing table



```
db1=# ALTER TABLE users ALTER active SET NOT NULL;
ALTER TABLE

db1=# \d users
                                 Table "public.users"
 Column |         Type          | Nullable |           Default
--------+-----------------------+----------+------------------------------
 id     | integer               | not null | generated always as identity
 active | boolean               | not null | true
 name   | character varying(20) |          |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
```

Not Null Constraint

Field is now required

```
db1=# INSERT INTO users (name, active) VALUES ('foo', NULL);
ERROR:  null value in column "active" of relation "users" violates not-null constraint
DETAIL:  Failing row contains (3, null, foo).
```

`active` must be set

Primary Key

Primary unique identifier

Primary Key

Set of fields that uniquely identify each row in the table

- Must be unique
- Must contain a value
- A table can only have one primary key
- A table does not have to have a primary key
- Can have multiple fields ("composite key")

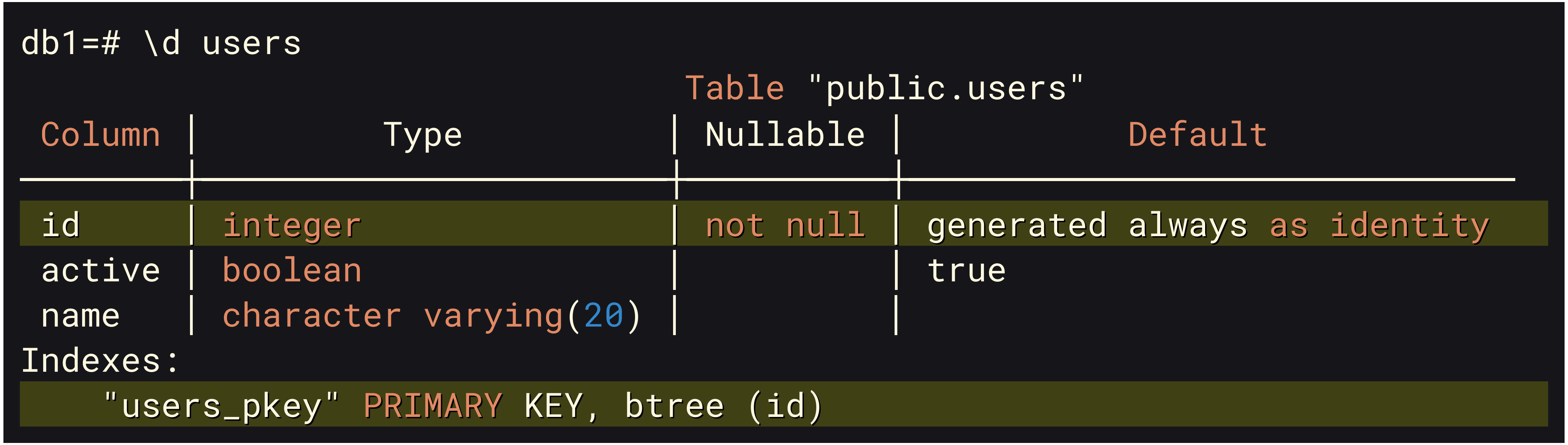
Primary Key

We already defined a primary key

```
db1=# CREATE TABLE users (
  id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  active BOOLEAN,
  name TEXT
);
```

Primary Key

We already defined a primary key



```
db1=# \d users
                                 Table "public.users"
 Column │         Type          │ Nullable │          Default
────────┼───────────────────────┼──────────┼──────────────────────────────
 id     │ integer               │ not null │ generated always as identity
 active │ boolean               │          │ true
 name   │ character varying(20) │          │
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
```

Primary Key

- Creates a unique index to enforce uniqueness
- Marks the fields as not null

Auto Incrementing Primary Key

```
db1=# CREATE TABLE users (
  id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  active BOOLEAN,
  name TEXT
);
```

- Creates a sequence
- Automatically populate PK with next values
- Used to be `serial` (soft-deprecated starting PostgreSQL 10)

Unique Constraint

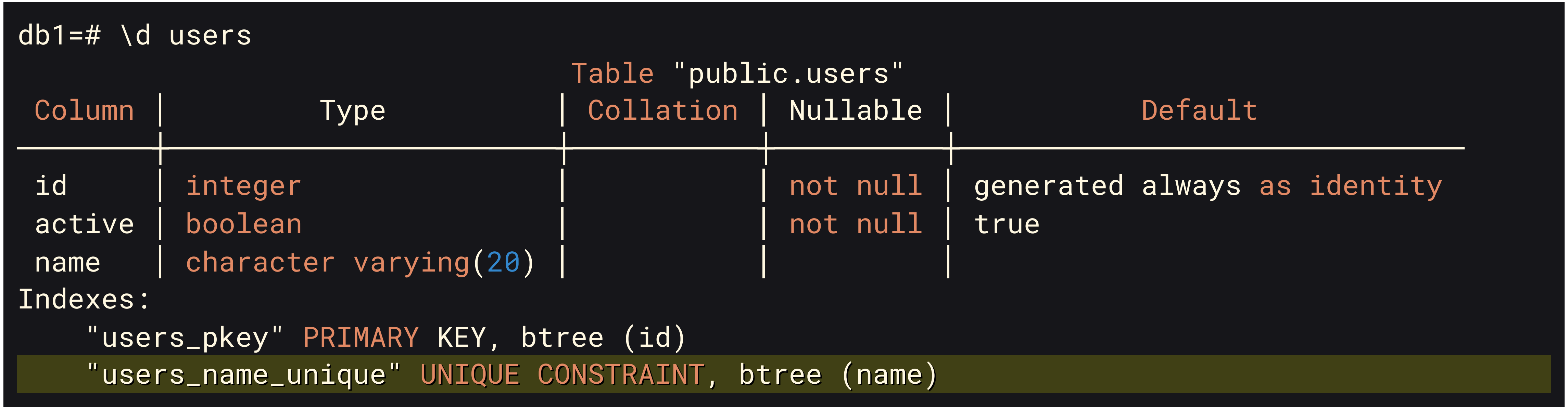
Ensure one or more fields are unique

Unique Constraint

Name must be unique

```
db1=# ALTER TABLE users ADD CONSTRAINT users_name_unique UNIQUE(name);
ALTER TABLE
```

Unique Constraint



```
db1=# \d users
                             Table "public.users"
 Column |         Type          | Collation | Nullable |           Default
--------+-----------------------+-----------+----------+------------------------------
 id     | integer               |           | not null | generated always as identity
 active | boolean               |           | not null | true
 name   | character varying(20) |           |          |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
    "users_name_unique" UNIQUE CONSTRAINT, btree (name)
```

Creates a unique index to enforce the constraint

Unique Constraint

```
db1=# INSERT INTO users (name) VALUES ('Haki Benita');
ERROR:  duplicate key value violates unique constraint "users_name_unique"
DETAIL:  Key (name)=(Haki Benita) already exists.
```

Cannot insert duplicate values!

Unique Constraint

Name is not really unique, so we can drop the constraint:

```
db1=# ALTER TABLE users DROP CONSTRAINT users_name_unique;
ALTER TABLE
```

Will also drop the index

Check Constraint

Implement special validation logic

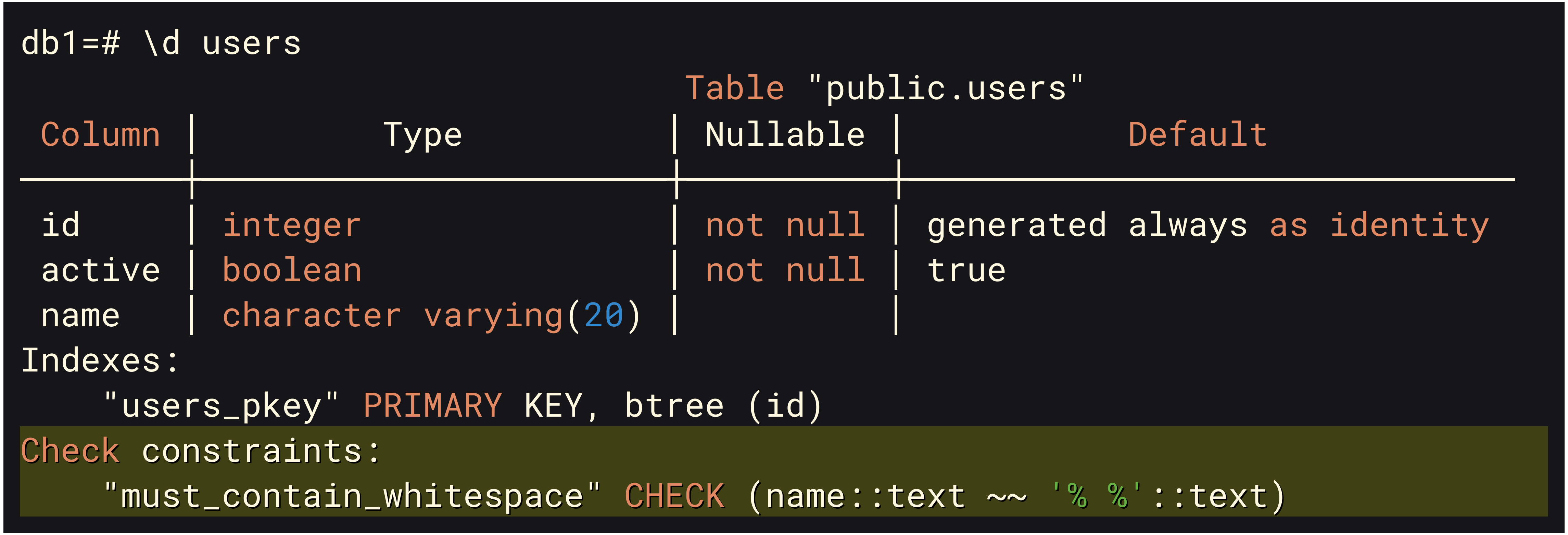
Check Constraint

Name must contain at least one whitespace

```
db1=# ALTER TABLE users ADD CONSTRAINT must_contain_whitespace CHECK (name LIKE '% %');
ALTER TABLE
```

Check Constraint

Name must contain at least one whitespace



```
db1=# \d users
                                 Table "public.users"
 Column │          Type          │ Nullable │          Default
────────┼────────────────────────┼──────────┼───────────────────────────────
 id     │ integer                │ not null │ generated always as identity
 active │ boolean                │ not null │ true
 name   │ character varying(20)  │          │
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
Check constraints:
    "must_contain_whitespace" CHECK (name::text ~~ '% %'::text)
```

Check Constraint

Name must contain at least one whitespace

```
db1=# INSERT INTO users (name) VALUES ('George');
ERROR:  new row for relation "users" violates check constraint "must_contain_whitespace"
DETAIL:  Failing row contains (4, t, George).
```

- Cannot set name without at least one whitespace
- Use meaningful names to make it easier to spot the problem

📖 Add Constraints Without Validating Immediately

Check Constraint

💡 TIP: Use a check constraint with unrestricted text type if the length may change in the future

```
CREATE TABLE student (
  id int,
  name text,
  CONSTRAINT name_length_check CHECK (length(name) < 20)
);

db1=# INSERT INTO student (id, name) VALUES (1, 'Haki Benita');
INSERT 0 1

db1=# INSERT INTO student (id, name) VALUES (2, 'Bernd Ottovordemgentschenfelde');
ERROR:  new row for relation "student" violates check constraint "name_length_check"
DETAIL:  Failing row contains (2, Bernd Ottovordemgentschenfelde).
```

Check Constraint

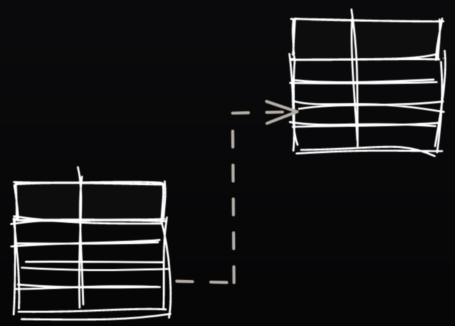
💡 TIP: It is easier to change a check constraint than a column's data type

```
db1=# ALTER TABLE student DROP CONSTRAINT name_length_check;
ALTER TABLE

db1=# ALTER TABLE student ADD CONSTRAINT name_length_check CHECK(length(name) < 100);
ALTER TABLE

db1=# INSERT INTO student (id, name) VALUES (2, 'Bernd Ottovordemgentschenfelde');
INSERT 0 1
```

📖 Gitlab database guide: Strings and the Text data type

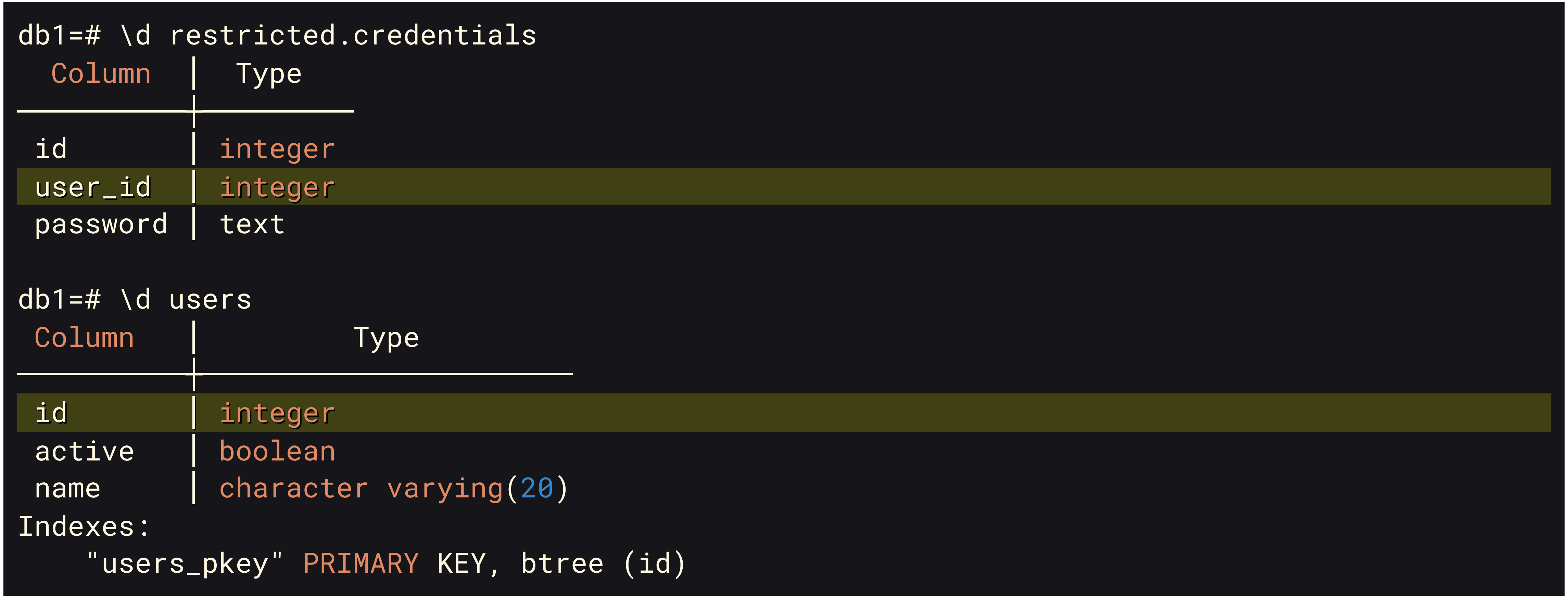
Foreign Key

Maintain relation between tables



Foreign Key

Add a foreign key between users and their credentials



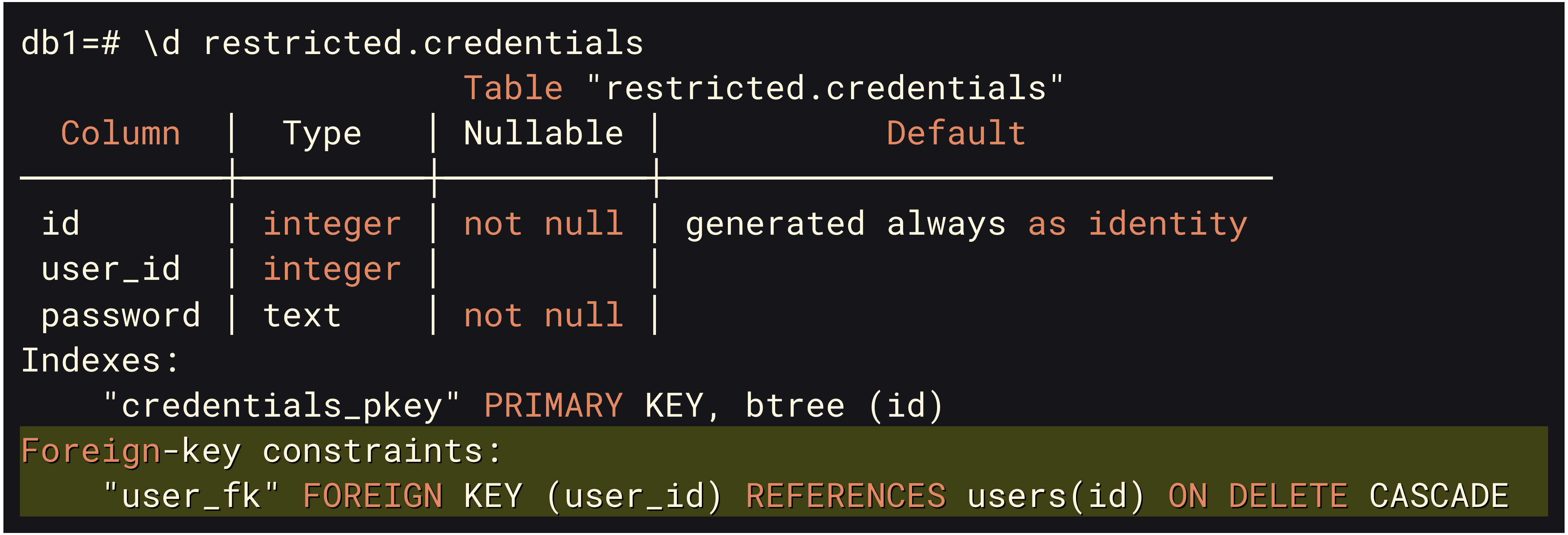
```
db1=# \d restricted.credentials
  Column  │  Type
──────────┼─────────
 id       │ integer
 user_id  │ integer
 password │ text

db1=# \d users
 Column   │        Type
──────────┼──────────────────────
 id       │ integer
 active   │ boolean
 name     │ character varying(20)
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
```

Foreign Key

Add a foreign key between users and their credentials

```
db1=# ALTER TABLE restricted.credentials ADD CONSTRAINT user_fk
    FOREIGN KEY (user_id) REFERENCES users(id)
    ON DELETE CASCADE;

ALTER TABLE
```

Foreign Key



```
db1=# \d restricted.credentials
                      Table "restricted.credentials"
  Column  |  Type   | Nullable |           Default
----------+---------+----------+------------------------------
 id       | integer | not null | generated always as identity
 user_id  | integer |          |
 password | text    | not null |
Indexes:
    "credentials_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "user_fk" FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
```

Foreign Key

```
db1=# ALTER TABLE restricted.credentials ADD CONSTRAINT user_fk
    FOREIGN KEY (user_id) REFERENCES users(id)
    ON DELETE CASCADE;

ALTER TABLE
```

Column `user_id` must have a corresponding value in the `id` column in table `users`

Foreign Key

Cannot add credentials for non-existing users

```
db1=# INSERT INTO restricted.credentials (user_id, password) VALUES (99, 'secret');
ERROR:  insert or update on table "credentials" violates foreign key constraint "user_fk"
DETAIL:  Key (user_id)=(99) is not present in table "users".
```

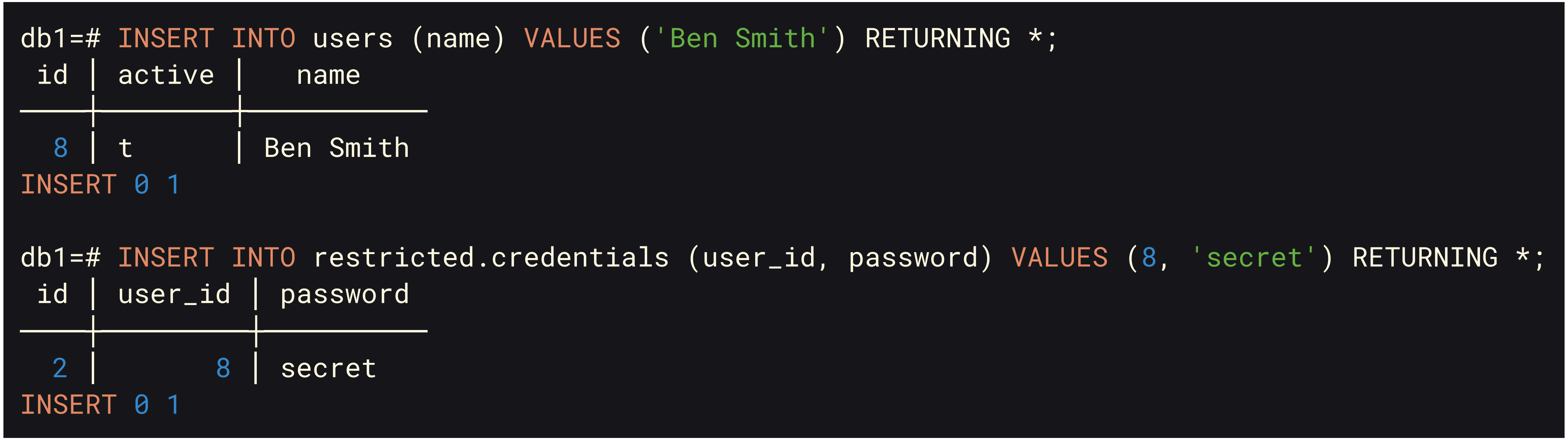
Foreign Key

```
db1=# ALTER TABLE restricted.credentials ADD CONSTRAINT user_fk
    FOREIGN KEY (user_id) REFERENCES users(id)
    ON DELETE CASCADE;

ALTER TABLE
```

- `CASCADE`: When a user is deleted from the `users` table, delete the associated credentials in the `credentials` table
- `RESTRICT`: prevent deleting users with credentials

Foreign Key

Add a new user with credentials



```
db1=# INSERT INTO users (name) VALUES ('Ben Smith') RETURNING *;
 id │ active │   name
────┼────────┼───────────
  8 │ t      │ Ben Smith
INSERT 0 1

db1=# INSERT INTO restricted.credentials (user_id, password) VALUES (8, 'secret') RETURNING *;
 id │ user_id │ password
────┼─────────┼──────────
  2 │       8 │ secret
INSERT 0 1
```

Foreign Key

Delete user

```
db1=# DELETE FROM users WHERE id = 8;
DELETE 1

db1=# SELECT * FROM restricted.credentials WHERE user_id = 8;
 id │ user_id │ password
────┼─────────┼──────────
(0 rows)
```

Credentials were also deleted

Recap

- Make field required -> Not null constraint
- Custom validation logic -> Check constraint
- Ensuring uniqueness -> Unique constraint
- Maintain relation between tables -> Foreign key

Conclusion

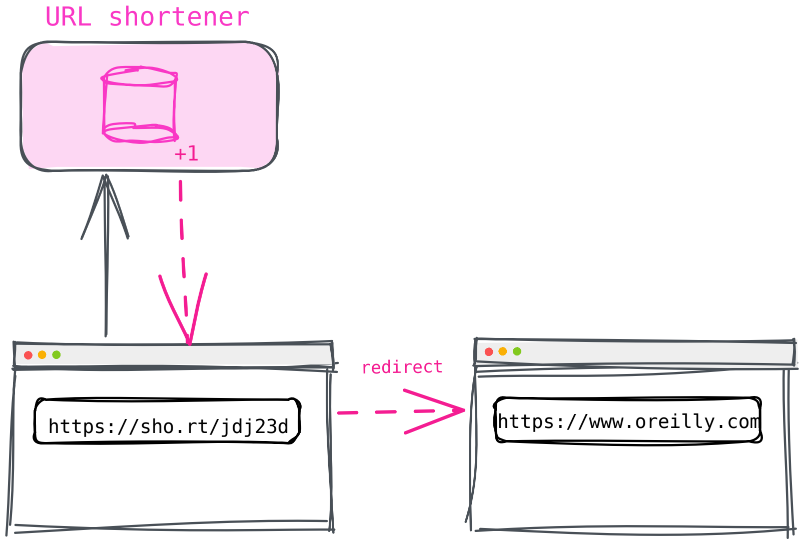
- Database is the source of truth
- Database is the last line of defence
- Data integrity is crucial

Constraints are good… use constraints!

Exercise

URL Shortener

- Create short URLs that redirect to longer URLs
- Keeps track of the number of clicks
- Common in SMS, Tweets
- Useful to keep track of campaigns



Exercise

Getting Started with PostgreSQL: Creating and managing tables

Launch Katacoda scenario »

Users and Privileges

User

Creating a user

```
db1=# CREATE USER app PASSWORD 'secret';
CREATE ROLE

db1=# \connect db1 app
You are now connected to database "db1" as user "app".
```

User

Connecting from terminal

```
$ psql -d db1 -U app -W
Password:

db1=>
```

- `-d` name of database
- `-U` name of user
- `-W` prompt for password

Privileges

Types of privileges

- SELECT / UPDATE / DELETE / TRUNCATE on TABLE
- CONNECT on DATABASE
- CREATE on DATABASE
- more...

Privileges

Enforce privileges

```
db1=> \conninfo
You are connected to database "db1" as user "app".

db1=# SELECT * FROM users;
ERROR: permission denied for table users
```

Privileges

Grant select privileges

```
db1=> \connect db1 postgres
You are now connected to database "db1" as user "postgres".

db1=# GRANT SELECT ON users TO app;
GRANT
```

Privileges

Grant select privileges

```
db1=# \connect db1 app
You are now connected to database "db1" as user "app".

db1=> SELECT * FROM users;
 id │ active │ name
────┼────────┼──────
  1 │ t      │ Haki
  2 │ f      │ Bob
```

Privileges

Grant insert, update and delete privileges

```
db1=> \connect db1 postgres
You are now connected to database "db1" as user "postgres".

db1=# GRANT UPDATE, INSERT, DELETE ON users TO app;
GRANT
```

Privileges

Use insert, update and delete privileges

```
db1=# \connect db1 app
You are now connected to database "db1" as user "app".

db1=> INSERT INTO users (active, name) VALUES (true, 'Jim');
INSERT 0 1

db1=> UPDATE users SET active = false WHERE name = 'Jim';
UPDATE 1

db1=> DELETE FROM users WHERE name = 'Jim';
DELETE 1
```

Privileges

Revoke privileges

```
db1=> \connect db1 postgres
You are now connected to database "db1" as user "postgres".

db1=# REVOKE DELETE ON users FROM app;
REVOKE
```

Notice the user still has update, insert and select privileges

Privileges

Revoke privileges

```
db1=# \connect db1 app
You are now connected to database "db1" as user "app".

db1=> DELETE FROM users;
ERROR: permission denied for table users
```

Privileges

Check user privileges on table

```
SELECT grantee, privilege_type
FROM information_schema.role_table_grants
WHERE table_name = 'users'
AND grantee = 'app';

 grantee │ privilege_type
─────────┼───────────────
 app     │ INSERT
 app     │ SELECT
 app     │ UPDATE
```

Privileges

Grant privileges on schema

```
db1=> \connect db1 postgres
You are now connected to database "db1" as user "postgres".

db1=> GRANT USAGE ON SCHEMA restricted TO app;
GRANT
```

Access to custom schemas require explicit `USAGE` privilege

Privileges

Allow access to specific fields

```
db1=# GRANT SELECT (user_id) ON restricted.credentials TO app;
GRANT
```

Useful for tables with sensitive information such as PII or passwords.

📖 Grant Permissions on Specific Columns

Privileges

Allow access to specific fields

```
db1=> SELECT * FROM restricted.credentials;
ERROR: permission denied for table credentials

db1=> SELECT user_id, password FROM restricted.credentials;
ERROR: permission denied for table credentials

db1=> SELECT user_id FROM restricted.credentials;
 user_id
---------
(0 rows)
```

Roles

- Similar to user
- Don't have login privileges by default
- Should be thought of as a group of privileges a user can be a member of

Roles

Create role

```
db1=# \connect db1 postgres
You are now connected to database "db1" as user "postgres".

db1=# CREATE ROLE analyst;
CREATE ROLE
```

Roles

Grant privileges to role

```
db1=# GRANT SELECT ON ALL TABLES IN SCHEMA public TO analyst;
GRANT

db1=# GRANT UPDATE ON users TO analyst;
GRANT
```

Syntax is similar to granting privileges to users

Roles

Grant membership in a role:

```
db1=# CREATE USER bob;
CREATE ROLE

db1=# GRANT analyst TO bob;
GRANT
```

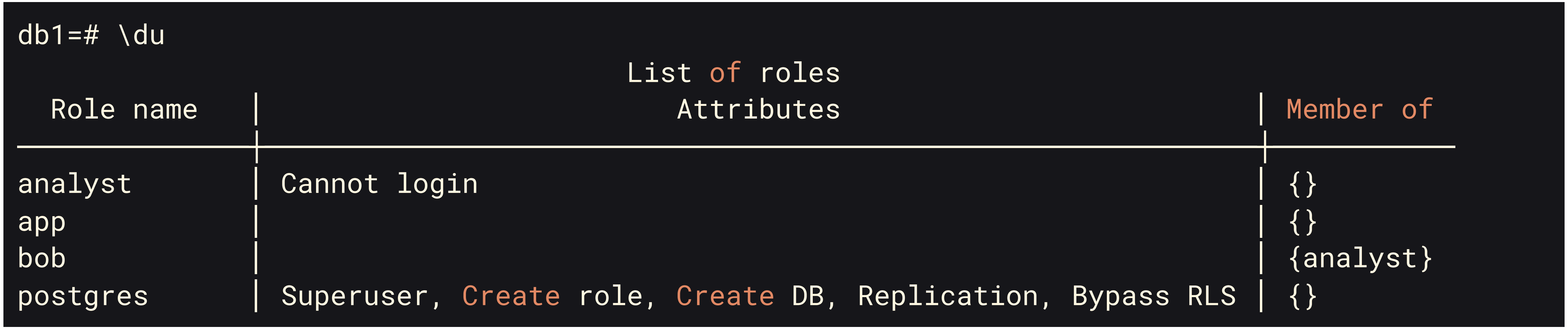
Roles

User `bob` now has all the privileges the role `analyst` has

```
db1=# \connect db1 bob
You are now connected to database "db1" as user "bob".

db1=> SELECT * FROM users;
 id │ active │ name
────┼────────┼──────
  1 │ t      │ Haki
  2 │ f      │ Bob
```

Roles

List roles



```
db1=# \du
                                     List of roles
  Role name   │                         Attributes                         │ Member of
──────────────┼────────────────────────────────────────────────────────────┼────────────
analyst       │ Cannot login                                               │ {}
app           │                                                            │ {}
bob           │                                                            │ {analyst}
postgres      │ Superuser, Create role, Create DB, Replication, Bypass RLS │ {}
```

- Notice `analyst` role cannot login
- Notice `bob` is member of `analyst`
- Notice `postgres` is a superuser

Recap

- Create users and roles
- Grant and revoke privileges and roles
- Grant privileges on specific columns

Query Optimizer

Where the magic happens...

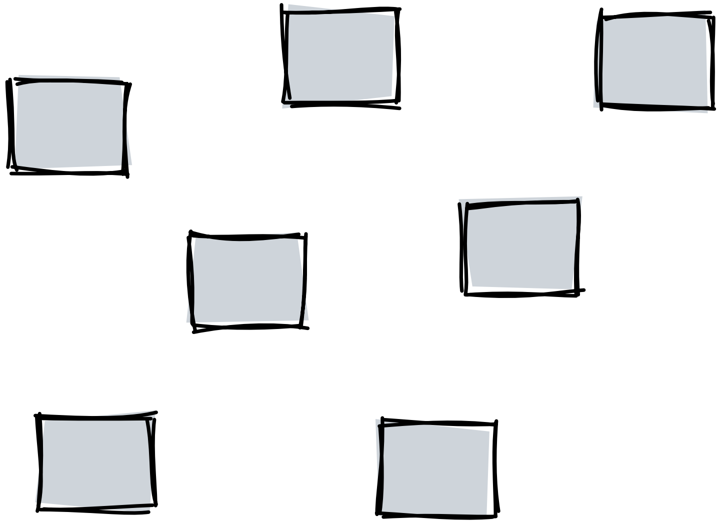
The Path of a Query

- **Parser**: Check for syntax errors
- **Rewrite**: Adjustments to the query (inline views etc.)
- **Planner / Optimizer**: Produce execution plan
- **Executer**: Execute the query according to the execution plan

Query Optimizer

1. **Generate all possible execution plans**

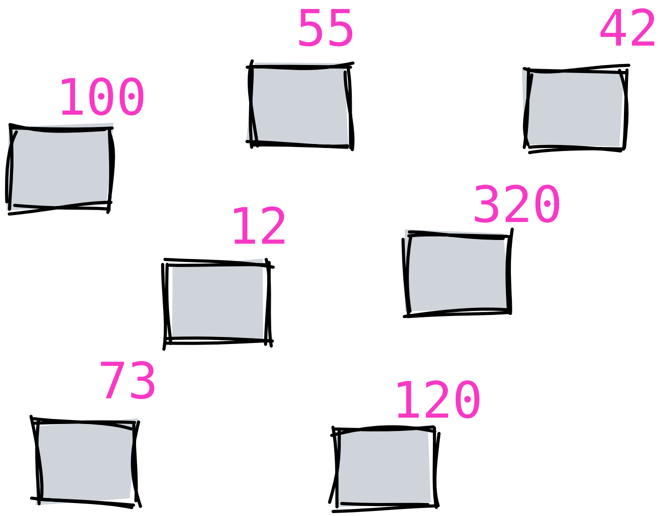
- This can take some time, depending on the query
- Cascade of doom: Postgres update that led to 70% failure



Query Optimizer

2. **Estimate the cost for each plan**

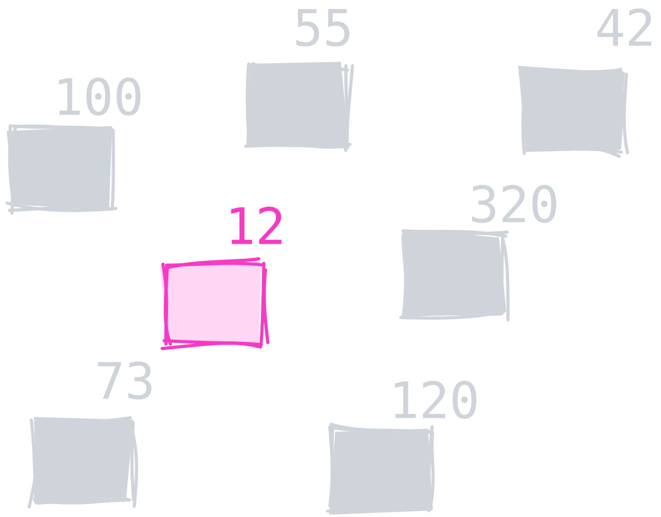
- Cost is measured in arbitrary units
- Using stats from analyzing tables and indexes
- Cost can be used to compare execution plans
- Mostly affected by IO



Query Optimizer

3. **Choose the plan with the lowest cost**

- The lower the cost, the faster the execution is (expected) to be



Execution Plan

Produce execution plan using the `EXPLAIN` command

```
db1=# EXPLAIN SELECT * FROM shorturl WHERE key = '123456';
                                    QUERY PLAN
----------------------------------------------------------------------------------
 Index Scan using shorturl_key_key on shorturl  (cost=0.29..8.30 rows=1 width=48)
   Index Cond: ((key)::text = '123456'::text)
```

- Only produces an execution plan
- Does not execute the query

Execution Plan

Produce execution plan and execute query

```
db1=# EXPLAIN (ANALYZE, TIMING) SELECT * FROM shorturl WHERE key = '123456';
                       QUERY PLAN
------------------------------------------------------------------------------------------
 Index Scan using shorturl_key_key on shorturl  (cost=0.29..8.30 rows=1 width=48)
                                                (actual time=0.032..0.033 rows=0 loops=1)
   Index Cond: ((key)::text = '123456'::text)
 Planning Time: 0.121 ms
 Execution Time: 0.064 ms
```

- Execute and time query
- Display executed plan
- Display estimated vs. actual estimates

Execution Plan

Reading an execution plan

```
db1=# EXPLAIN SELECT * FROM shorturl WHERE key = '123456';
                                    QUERY PLAN
----------------------------------------------------------------------------------
 Index Scan using shorturl_key_key on shorturl  (cost=0.29..8.30 rows=1 width=48)
   Index Cond: ((key)::text = '123456'::text)
```

- The database used the index on `key` to find the row
- The optimizer estimates 1 row in the result

Row Estimates

How the optimizer estimates how many rows in the result

Row Estimates

Query all short URLs

```
db1=# EXPLAIN SELECT * FROM shorturl;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on shorturl  (cost=0.00..199.00 rows=10000 width=48)
```

- Database plans to scan the entire table
- Optimizer estimates 10,000 rows in the result

Row Estimates

Using statistics on tables and indexes:

```
db1=# SELECT reltuples FROM pg_class WHERE relname = 'shorturl';
 reltuples
-----------
     10000
```

Table is *estimated* to have 10,000 tuples (rows)

Row Estimates

Find short URLs with no hits:

```
db1=# EXPLAIN SELECT * FROM shorturl WHERE hits = 0;
                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on shorturl  (cost=0.00..224.00 rows=98 width=48)
   Filter: (hits = 0)
```

- Database plans to scan the entire table
- Optimizer estimates 98 rows in the result

Row Estimates

Using statistics on table columns:

```
db1=# SELECT tablename, attname, most_common_vals, most_common_freqs
FROM pg_stats
WHERE tablename = 'shorturl' AND attname = 'hits';

-[ RECORD 1 ]----------------------------------
tablename         | shorturl
attname           | hits
most_common_vals  | {0, 7306, 9658, ...
most_common_freqs | {0.0098, 0.0008, 0.0007, ...
```

`number of rows` (10,000) * `% hits eq 0` (0.0098) = 98 rows

Statistics

- The database keeps statistics on tables, indexes and columns
- Statistics can be collected explicitly using `ANALYZE`
- The database can collect statistics in the background

Statistics

Explicitly collect statistics on a table:

```
db1=# ANALYZE shorturl;
ANALYZE
```

Statistics

Collect statistics automatically in the background:

```
db1=# show autovacuum;
 autovacuum
------------
 on
```

- "autovacuum" is a routine maintenance task performed by the database
- The database can analyze tables and indexes as they are vacuumed

Statistics

Check when analyzed:

```
db1=# SELECT last_analyze, last_autoanalyze
FROM pg_stat_all_tables
WHERE relname = 'shorturl';

─[ RECORD 1 ]────┬──────────────────────────────
last_analyze     │ 2021-08-05 11:09:45.228915+03
last_autoanalyze │ 2021-08-05 09:58:40.550875+03
```

Recap

- Optimizer uses statistics to produce execution plans
- Keeping statistics up to date has a significant effect on query performance
- Statistics can be collected explicitly using `ANALYZE`, or in the background by enabling autovacuum

Indexes

The need for speed

Indexes

- Speed us access to data
- Enforce constraints (Unique, Foreign Key)
- PostgreSQL offers many different types of indexes

B-Tree Index

The king of all indexes!

Exercise

Exercise

Getting Started with PostgreSQL: B-Tree Index Features

- Partial B-Tree index
- Inclusive B-Tree index
- Function based B-Tree index

Launch Katacoda scenario »

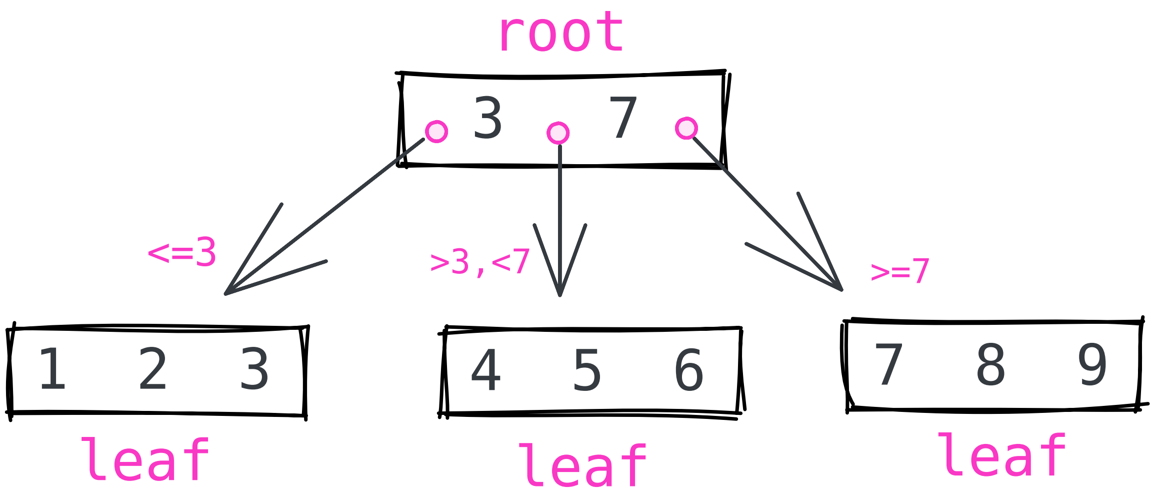
B-Tree Index

How does a B-Tree index work?

1. You have these values:

   1 2 3 4 5 6 7 8 9

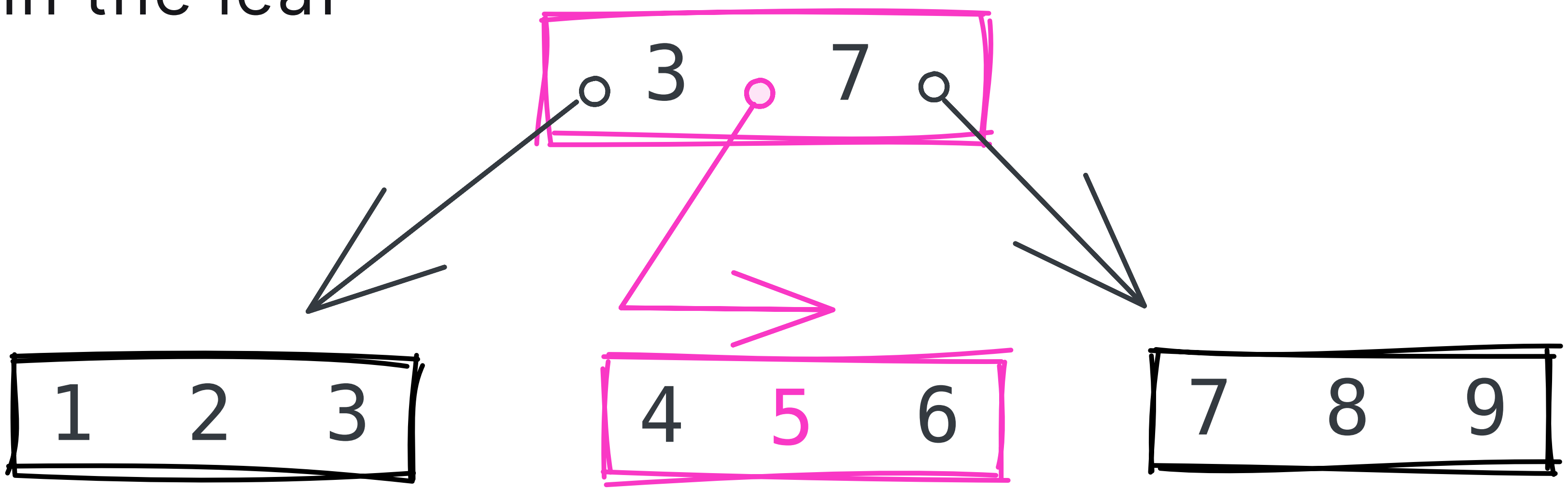
2. Create a tree



B-Tree Index

Search for the value `5`:

1. Scan the index root
2. Find the leaf block
3. Search the value in the leaf



B-Tree Index

Create a B-Tree index

```
db1=# CREATE INDEX shorturl_hits_ix ON shorturl USING btree(hits);
CREATE INDEX
```

Can omit `USING btree()`, this is the default

Partial Index

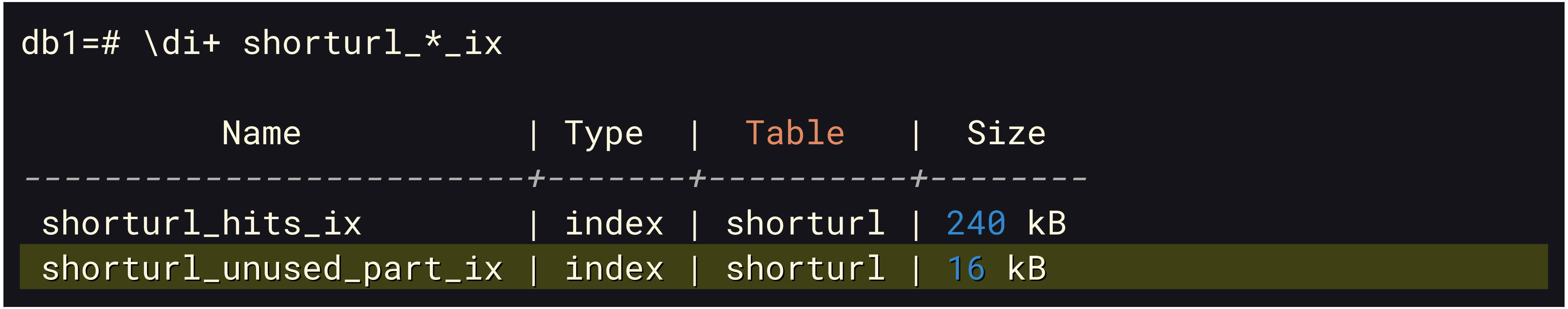
Index only a portion of the table

```
db1=# CREATE INDEX shorturl_unused_part_ix ON shorturl (id) WHERE hits = 0;
CREATE INDEX
```

Index only short URLs with zero hits

Partial Index

Smaller index



```
db1=# \di+ shorturl_*_ix

          Name           | Type  |  Table   |  Size
-------------------------+-------+----------+--------
 shorturl_hits_ix        | index | shorturl | 240 kB
 shorturl_unused_part_ix | index | shorturl | 16 kB
```

Partial Index

- Produce smaller indexes
- Limited to queries using the indexed rows
- Nullable columns are great candidates
- Use when possible

📖 The Unexpected Find That Freed 20GB of Unused Index Space

Inclusive Index

Store additional data in the index leafs:

```
db1=# CREATE UNIQUE INDEX shorturl_key_including_url_ix ON shorturl(key) INCLUDE (url);
CREATE INDEX
```

Index `key` and include the value of `url` in the index leaf

Inclusive Index

Fulfill queries using just the index:

```
db1=# EXPLAIN (ANALYZE, TIMING) SELECT url FROM shorturl WHERE key = 'key123';
                         QUERY PLAN
--------------------------------------------------------------------
Index Only Scan using shorturl_key_including_url_ix on shorturl
  Index Cond: (key = 'key123'::text)
  Heap Fetches: 0
Planning Time: 0.534 ms
Execution Time: 0.120 ms
```

Notice "Index Only Scan"

Inclusive Index

- Fulfill queries without accessing the table
- Index can get very big
- Great for large tables with frequent queries that only use a limited number of columns
- Use with caution
- Non-unique composite indexes can be good candidates for inclusive indexes

Function Based Index

Index an expression

```
db1=# CREATE INDEX shorturl_domain_ix ON shorturl (substring(url FROM '.*://([^/]+)'));
CREATE INDEX
```

Index domain part of URL using regular expression (regexp)

https://hakibenita.com/sql-for-data-analysis

Function Based Index

Can be used by queries using the same expression

```
-- No index
db1=# SELECT * FROM shorturl WHERE substring(url FROM '.*://([^/]+)') = 'hakibenita.com';
Execution Time: 48.918 ms

-- With index
db1=# SELECT * FROM shorturl WHERE substring(url FROM '.*://([^/]+)') = 'hakibenita.com';
Execution Time: 0.914
```

Much faster!

Function Based Index

- Index will only be considered only if the expressions is exactly the same
- Useful for existing applications (you can't change)
- Consider calculated columns as an alternative

Recap

- B-Tree is the default index type and what you normally need
- Used to enforce unique constraints
- Many features:
  - Partial
  - Function based
  - Inclusive
  - more...

Performance

What's it all about

Performance

- CPU
- Memory
- Disk Space
- Cost

Not just speed!

Table Size

Functions to get table size

Function	Description
`pg_relation_size`	Table size
`pg_table_size`	Including TOAST
`pg_total_relation_size`	Including TOAST & Indexes

TOAST is extended storage for large values

📖 The Surprising Impact of Medium-Size Texts on PostgreSQL Performance

Table Size

```
db1=# SELECT pg_table_size('shorturl');
 pg_table_size
───────────────
        851968

db1=# SELECT pg_size_pretty(pg_table_size('shorturl'));
 pg_size_pretty
────────────────
 832 kB

db1=# \dt+ shorturl
                           List of relations
 Schema |   Name   | Type  | Owner | Persistence |  Size
────────┼──────────┼───────┼───────┼─────────────┼────────
 public | shorturl | table | haki  | permanent   | 832 kB
```

Index Size

Check index size

```
db1=# \di+ shorturl*
                          List of relations
 Schema │        Name       │ Type  │ Owner │  Table   │ Persistence │  Size
────────┼───────────────────┼───────┼───────┼──────────┼─────────────┼────────
 public │ shorturl_key_key  │ index │ haki  │ shorturl │ permanent   │ 312 kB
 public │ shorturl_pkey     │ index │ haki  │ shorturl │ permanent   │ 240 kB
```

Naming convention makes this easy...

Hash Index

The Ugly Duckling of index types

Reverse Lookup

Find keys referencing a URL

```
SELECT *
FROM shorturl
WHERE url = 'https://hakibenita.com/postgresql-hash-index';
```

- URL is not unique, but it is *almost unique*
- URL can be a large text

Reverse Lookup

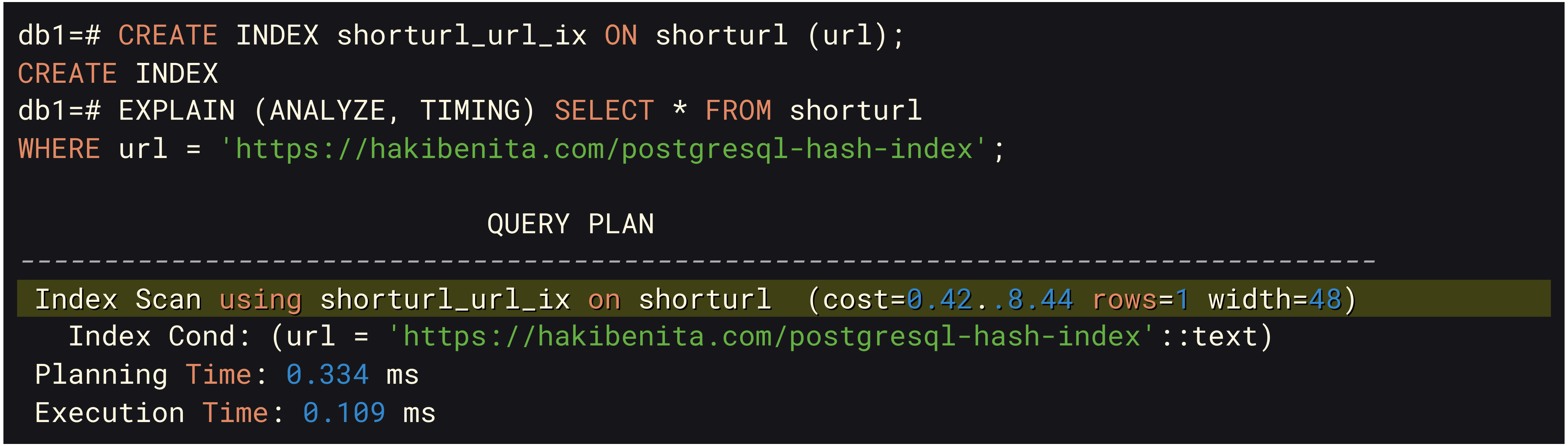
No index

```
db1=# EXPLAIN (ANALYZE, TIMING) SELECT * FROM shorturl
WHERE url = 'https://hakibenita.com/postgresql-hash-index';

                            QUERY PLAN
---------------------------------------------------------------------------------
 Seq Scan on shorturl  (cost=0.00..2239.00 rows=1 width=48)
   Filter: (url = 'https://hakibenita.com/postgresql-hash-index'::text)
   Rows Removed by Filter: 99998
 Planning Time: 0.139 ms
 Execution Time: 15.506 ms
```

Full table scan

Reverse Lookup

B-Tree index



```
db1=# CREATE INDEX shorturl_url_ix ON shorturl (url);
CREATE INDEX
db1=# EXPLAIN (ANALYZE, TIMING) SELECT * FROM shorturl
WHERE url = 'https://hakibenita.com/postgresql-hash-index';

                            QUERY PLAN
---------------------------------------------------------------------------------
 Index Scan using shorturl_url_ix on shorturl  (cost=0.42..8.44 rows=1 width=48)
   Index Cond: (url = 'https://hakibenita.com/postgresql-hash-index'::text)
 Planning Time: 0.334 ms
 Execution Time: 0.109 ms
```

Index scan using B-Tree index

Reverse Lookup

Recap

Access method	Reverse lookup timing
Full table scan	15.506 ms
B-Tree index scan	0.109 ms

Hash Index

How does a Hash index work?

Value
A
B
C
D

Hash Index

Apply a hash function on the values (`hashchar`)

Value	hashchar(value)
`A`	`-201530951`
`B`	`626080936`
`C`	`2018813598`
`D`	`2016322020`

Hash functions for other types

`hashtext` `hashchar` `hash_array` `jsonb_hash` `timestamp_hash`

Hash Index

Divide to N buckets using `mod(N)`:

Value	`hashchar(value)`	Bucket `mod(2)`
`A`	`-201530951`	1
`B`	`626080936`	0
`C`	`2018813598`	0
`D`	`2016322020`	0

Hash Index

Build the index:

Bucket	Row Pointers
0	2, 3, 4
1	1

Hash Index

Search for the value `B`:

1. Find bucket

```
hashchar('B') -> 626080936
```

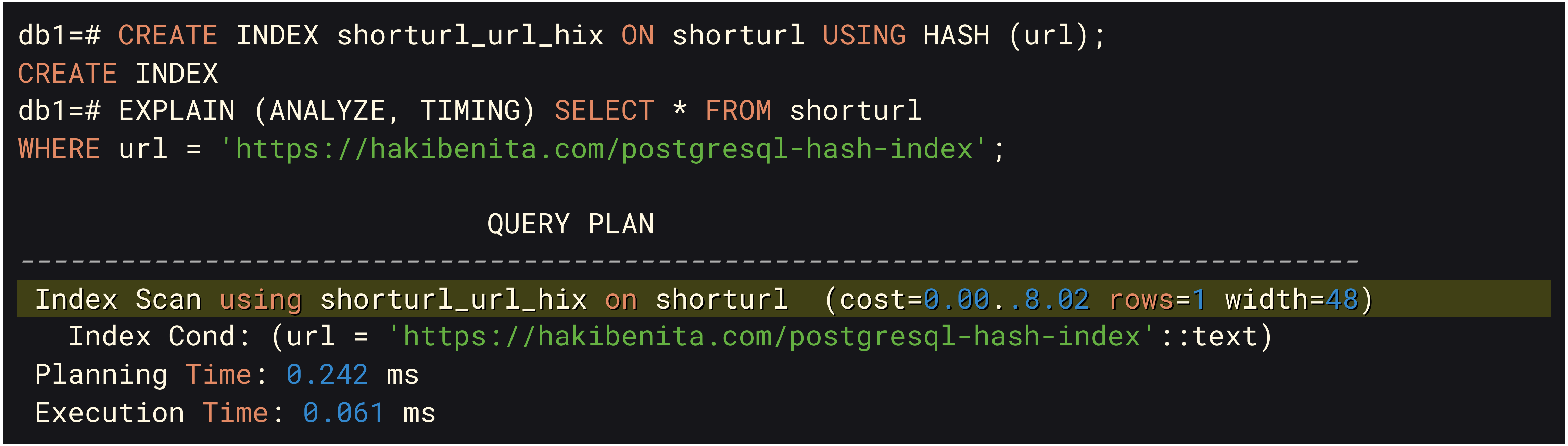
```
mod(626080936, 2) -> 0
```

2. Scan rows in bucket 0 and search value:

Bucket	Row Pointers
0	2, 3, 4

Reverse Lookup

Hash index



```
db1=# CREATE INDEX shorturl_url_hix ON shorturl USING HASH (url);
CREATE INDEX
db1=# EXPLAIN (ANALYZE, TIMING) SELECT * FROM shorturl
WHERE url = 'https://hakibenita.com/postgresql-hash-index';

                            QUERY PLAN
--------------------------------------------------------------------------------
 Index Scan using shorturl_url_hix on shorturl  (cost=0.00..8.02 rows=1 width=48)
   Index Cond: (url = 'https://hakibenita.com/postgresql-hash-index'::text)
 Planning Time: 0.242 ms
 Execution Time: 0.061 ms
```

Index scan using Hash index

Reverse Lookup

B-Tree vs. Hash Index speed

Access method	Reverse lookup timing
Full table scan	15.506 ms
B-Tree index scan	0.109 ms
Hash index scan	0.061 ms

Hash index is faster!

Reverse Lookup

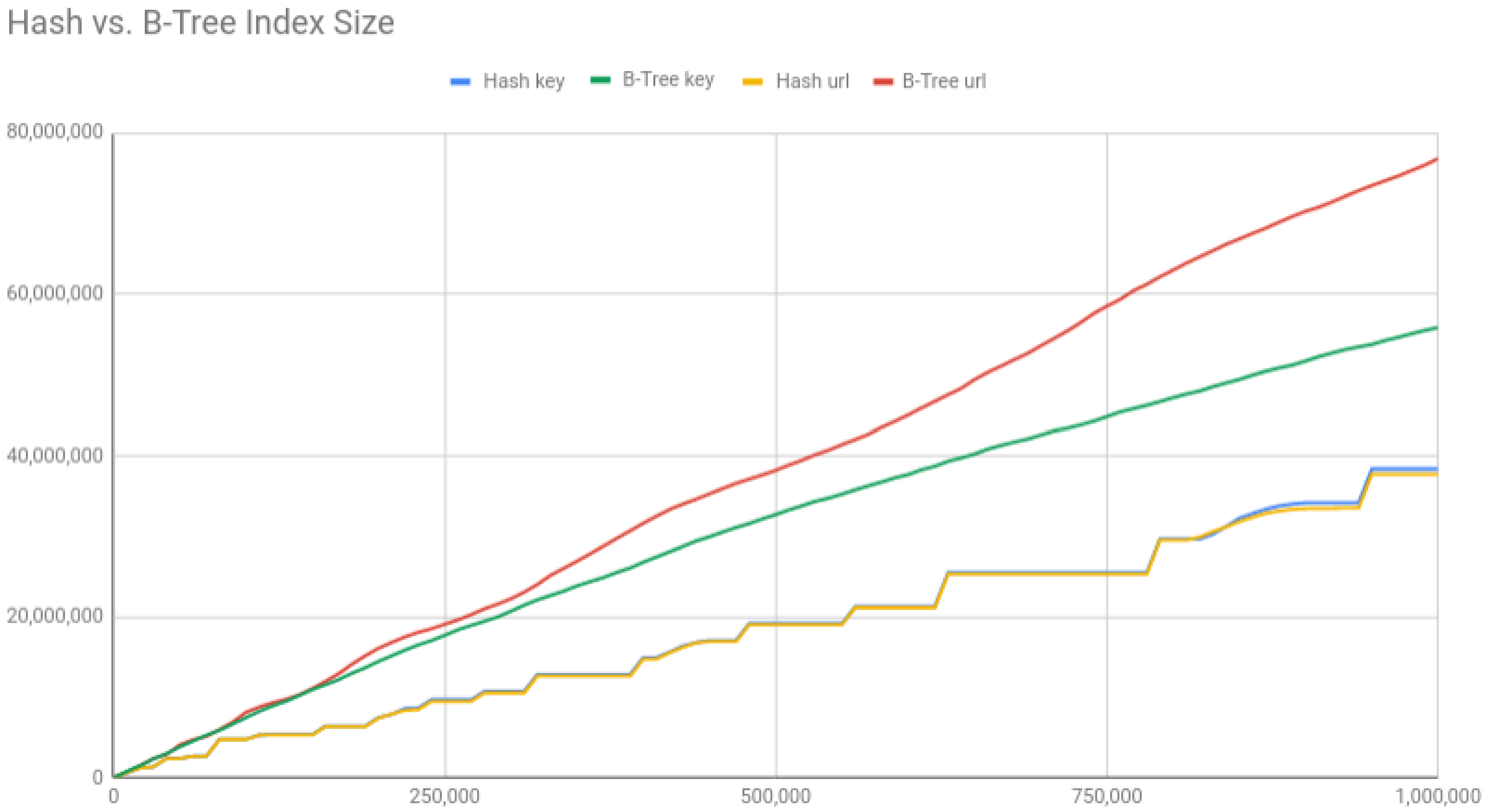
B-Tree vs. Hash Index size

```
db1=# \di+ shorturl_url_*
                               List of relations
Schema |       Name       | Type  |  Owner   |  Table   |  Size
-------+------------------+-------+----------+----------+---------
public | shorturl_url_hix | index | postgres | shorturl | 4112 kB
public | shorturl_url_ix  | index | postgres | shorturl | 4584 kB
```

Hash index is smaller!

Hash Index

Hash vs. B-Tree index size



Hash Index Restrictions

- ❌ Unique
- ❌ Composite
- ❌ Sorting
- ❌ Range search

Hash Index

- Ideal when values are *almost unique*
- Not affected by the size of the values
- Can be smaller and faster than a B-Tree
- Was discouraged prior to PostgreSQL 10, but no more!

Hash Index

Go Further:

- ⚙️ Getting Started with PostgreSQL: Hash Index Interactive Katacode scenario
- 📖 Re-Introducing Hash Indexes in PostgreSQL

Block Range Index

Keep range of values within a number of adjacent pages

Range Search

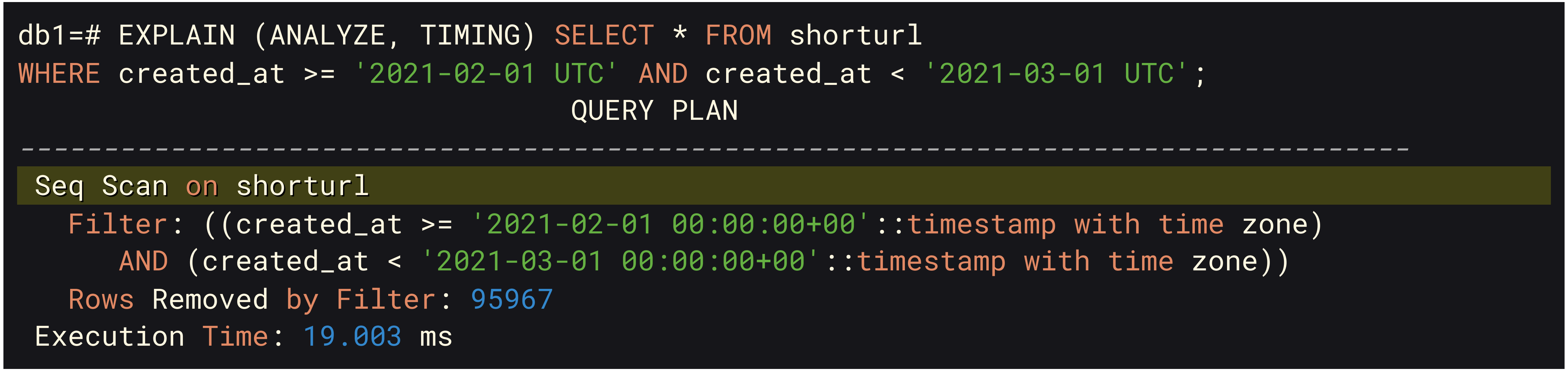
Find short URLs that were created in a date range

```
SELECT *
FROM shorturl
WHERE created_at >= '2021-02-01 UTC'
AND created_at < '2021-03-01 UTC';
```

- Creation date set by the application when a row is added
- Creation date is naturally incrementing

Range Search

No Index

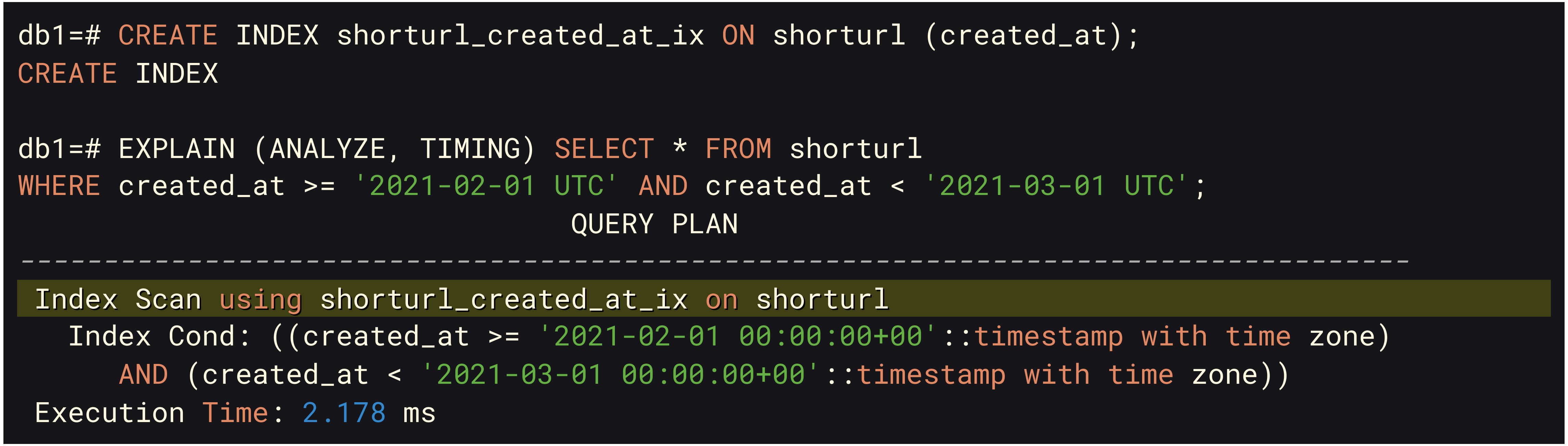


```
db1=# EXPLAIN (ANALYZE, TIMING) SELECT * FROM shorturl
WHERE created_at >= '2021-02-01 UTC' AND created_at < '2021-03-01 UTC';
                                  QUERY PLAN
-------------------------------------------------------------------------------------
 Seq Scan on shorturl
   Filter: ((created_at >= '2021-02-01 00:00:00+00'::timestamp with time zone)
      AND (created_at < '2021-03-01 00:00:00+00'::timestamp with time zone))
   Rows Removed by Filter: 95967
 Execution Time: 19.003 ms
```

Full table scan

Range Search

B-Tree index



```
db1=# CREATE INDEX shorturl_created_at_ix ON shorturl (created_at);
CREATE INDEX

db1=# EXPLAIN (ANALYZE, TIMING) SELECT * FROM shorturl
WHERE created_at >= '2021-02-01 UTC' AND created_at < '2021-03-01 UTC';
                                 QUERY PLAN
-----------------------------------------------------------------------------------
 Index Scan using shorturl_created_at_ix on shorturl
   Index Cond: ((created_at >= '2021-02-01 00:00:00+00'::timestamp with time zone)
      AND (created_at < '2021-03-01 00:00:00+00'::timestamp with time zone))
 Execution Time: 2.178 ms
```

B-Tree Index scan

Range Search

Recap

Access method	Index Size	Timing
Full table scan	-	19.003 ms
B-Tree index scan	2208 kB	2.178 ms

BRIN Index

How does a BRIN index work?

1. You have these values in a column, each is single table page:

   1 2 3 4 5 6 7 8 9

BRIN Index

How does a BRIN index work?

1. You have these values in a column, each is single table page:

   `1` `2` `3` `4` `5` `6` `7` `8` `9`

2. Divide the table into ranges of 3 adjacent pages:

   `[1,2,3]` `[4,5,6]` `[7,8,9]`

BRIN Index

How does a BRIN index work?

1. You have these values in a column, each is single table page:

   `1` `2` `3` `4` `5` `6` `7` `8` `9`

2. Divide the table into ranges of 3 adjacent pages:

   `[1,2,3]` `[4,5,6]` `[7,8,9]`

3. For each range, keep only the minimum and maximum values:

   `[1-3]` `[4-6]` `[7-9]`

BRIN Index

Use the index to search for the value 5:

- `[1–3]` 💀 Definitely not here
- `[4–6]` 💩 Might be here
- `[7–9]` 💀 Definitely not here

We only need to scan blocks 4, 5 and 6!

Range Search

BRIN index

```
db1=# CREATE INDEX shorturl_created_at_ix ON shorturl USING brin(created_at);
CREATE INDEX

db1=# EXPLAIN (ANALYZE, TIMING) SELECT * FROM shorturl
WHERE created_at >= '2021-02-01 UTC' AND created_at < '2021-03-01 UTC';
                                 QUERY PLAN
-----------------------------------------------------------------------------------
 Bitmap Heap Scan on shorturl
   Recheck Cond: ((created_at >= '2021-02-01 00:00:00+00'::timestamp with time zone) AND ...
   Rows Removed by Index Recheck: 8921 Heap Blocks: lossy=128
   ->  Bitmap Index Scan on shorturl_created_at_bix
         Index Cond: ((created_at >= '2021-02-01 00:00:00+00'::timestamp with time zone) AND ...
 Execution Time: 5.207 ms
```

Bitmap Index Scan on BRIN index

Range Search

BRIN index size

```
db1=# \di+ shorturl_created_at_*

          Name           | Type  |  Table   |  Size
-------------------------+-------+----------+---------
 shorturl_created_at_ix  | index | shorturl | 2208 kB
 shorturl_created_at_bix | index | shorturl | 48 kB
```

BRIN is very small!

Range Search

BRIN vs. B-Tree

Access method	Index Size	Timing
Full table scan	-	19.003 ms
B-Tree index scan	2208 kB	**2.178 ms**
BRIN index scan	**48 kB**	5.207 ms

- B-Tree is faster
- BRIN is smaller

BRIN Index

What if the values are **not sorted**?

1. `2` `9` `5` `1` `4` `7` `3` `8` `6`
2. `[2,9,5]` `[1,4,7]` `[3,8,6]`
3. `[2–9]` `[1–7]` `[3–8]`

BRIN Index

Use the index to search for the value 5:

- `[2–9]` 💩 Might be here
- `[1–7]` 💩 Might be here
- `[3–8]` 💩 Might be here

The index is useless!

Correlation

Correlation between logical and physical ordering:

```
db1=# SELECT attname, correlation FROM pg_stats WHERE tablename = 'shorturl';

  attname   | correlation
------------+--------------
 id         |            1
 key        | 0.0018741399
 url        | 0.0033386275
 hits       | 0.0015433382
 created_at |            1
```

1=incrementing -1=decreasing ~0=not correlated

BRIN Index

What if we increase the number of **pages per range** to 5?

1. `1` `2` `3` `4` `5` `6` `7` `8` `9`
2. `[1,2,3,4,5]` `[6,7,8,9]`
3. `[1–5]` `[6,9]`

BRIN Index

Use the index to search for the value 5:

- `[1-5]` 💩 Might be here
- `[6-9]` 💀 Definitely not here

We need to scan blocks 1, 2, 3, 4 and 5

BRIN Index

What if we decrease the number of **pages per range** to 2?

1. `1` `2` `3` `4` `5` `6` `7` `8` `9`
2. `[1,2]` `[3,4]` `[5,6]` `[7,8]` `[9]`
3. `[1-2]` `[3-4]` `[5-6]` `[7-8]` `[9-9]`

BRIN Index

Use the index to search for the value 5:

- `[1-2]` 💀 Definitely not here
- `[3-4]` 💀 Definitely not here
- `[5-6]` 💩 Might be here
- `[7-8]` 💀 Definitely not here
- `[9-9]` 💀 Definitely not here

We only need to scan blocks 5 and 6

BRIN Index: Pages per range

```sql
CREATE INDEX shorturl_created_at_ix ON shorturl
USING brin(created_at) WITH (pages_per_range = 128);
```

- Low `pages_per_range` -> more accurate, bigger size
- High `pages_per_range` -> less accurate, smaller size
- Default is 128, minimum is 2
- Start small!

BRIN Index

- Ideal when data is naturally sorted on disk
    - Find columns with high correlation in `pg_stats.correlation`
    - Columns for auto incrementing column (timestamps etc.)
- Ideal for tables that don't update frequently
- Adjust `pages_per_range` to find ideal range size

BRIN Index

Go Further:

- ⚙️ Getting Started with PostgreSQL: Block range index
  Interactive Katacode scenario
- 📖 Index Columns With High Correlation Using BRIN
- 📖 9 Django Tips for Working with Databases: BRIN indexes
- 📖 Multi minmax and Bloom operators for BRIN index
  New in PostgreSQL 14

Conclusion

You learned

- Architecture and terminology
- Manage tables and indexes
- Create constraints and choose appropriate data types
- Manage users, roles and privileges
- How the database produces execution plans
- How to create different types of indexes

You *also* learned

- How to be productive using psql
- Maintain data integrity using data types and constraints
- Evaluate different aspects of query performance
- Optimize query performance using indexes

See you next time!

[SQL Next Steps: Optimization]

Online Live Training, Aug 2, 2022

Haki Benita

📝 Check out my blog hakibenita.com

🐦 Find me on Twitter @be_haki

📫 Subscribe to my newsletter at hakibenita.com/subscribe

📨 Send me an email me@hakibenita.com