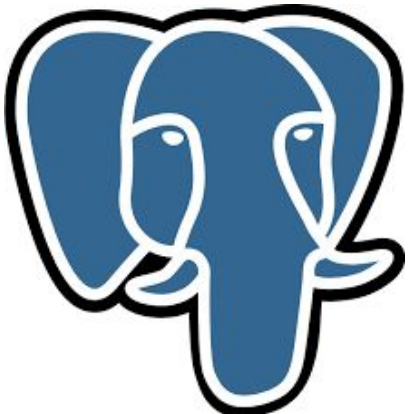
# Introduction to Relational Databases & SQL

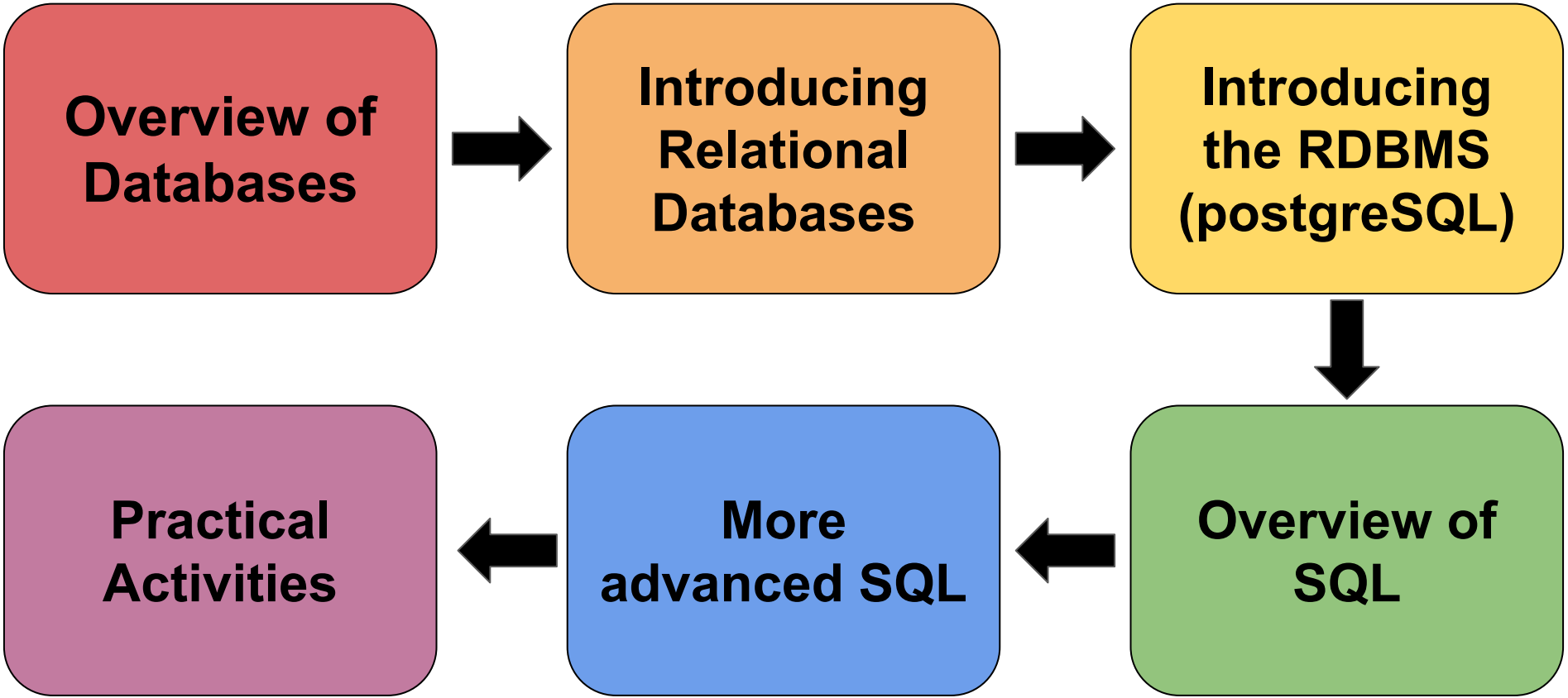
```sql
CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    username varchar(50) UNIQUE NOT NULL,
    password CHAR(40) NOT NULL,
    role user_role NOT NULL DEFAULT 'USER'
    email varchar(50) UNIQUE NOT NULL,
    name varchar(50),
    mobile varchar(50)
);
```



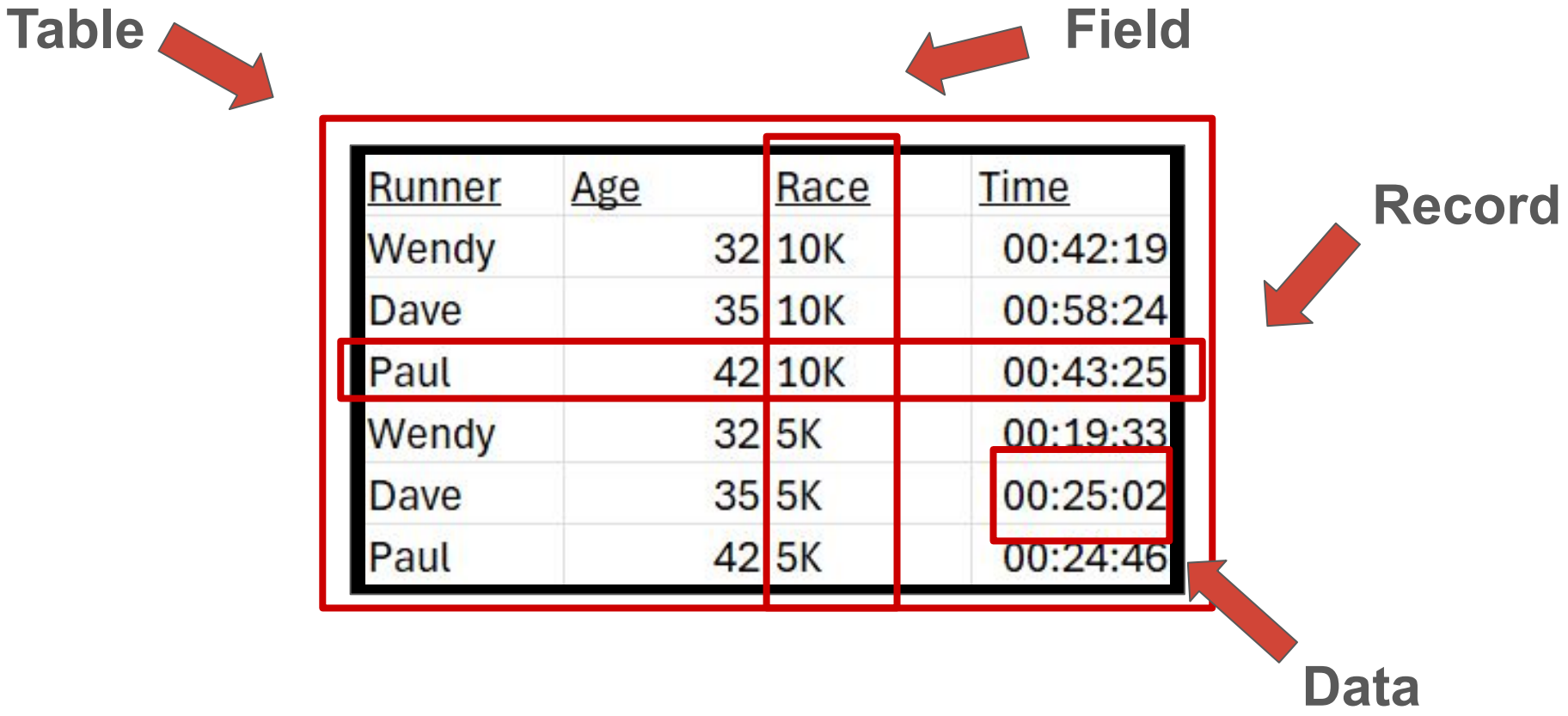




# How will this session work?



# Overview of Databases



# Overview of Databases









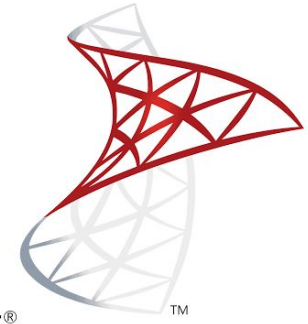


















# Overview of Databases



















**Flat-File Databases**

*Most primitive*

*A database made up of a single table in its own file*

# Overview of Databases




{.json}

PostgreSQL

MySQL®

SQLite

Microsoft® SQL Server®

## Relational Databases

- *Multiple tables*
- *Data can 'reference' other data that is shared*
- *Uses SQL*
- *Most common type of application database*

# Overview of Databases




**'NoSQL' database**

- *Anything non-relational*
- *No tables*
- *Based on either:*
  - *Key-value*
  - *documents*
  - *graphs*

MongoDB

reSQL

MySQL®

SQLite

Microsoft® SQL Server®

redis

# Introducing Flat-file Databases

A database made of a single table stored in its own file

| Runner | Age | Race | Time |
|---|---|---|---|
| Wendy | 32 | 10K | 00:42:19 |
| Dave | 35 | 10K | 00:58:24 |
| Paul | 42 | 10K | 00:43:25 |
| Wendy | 32 | 5K | 00:19:33 |
| Dave | 35 | 5K | 00:25:02 |
| Paul | 42 | 5K | 00:24:46 |

**Benefits**

- Lightweight
- Portable
- Easy to set-up
- Low cost
- Human-readable



{JSON}

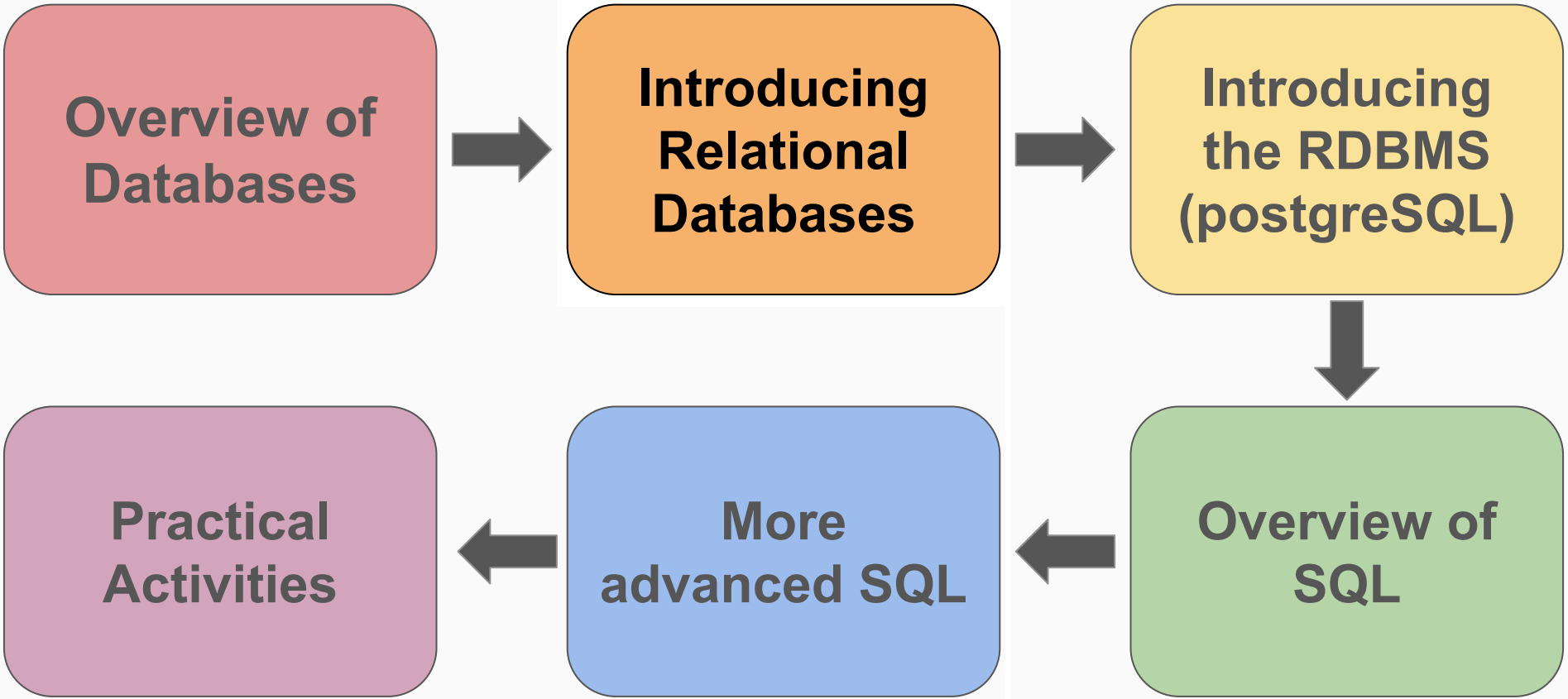
# Introducing Flat-file Databases

| Runner | Age | Race | Time |
|---|---|---|---|
| Wendy | 32 | 10K | 00:42:19 |
| Dave | 35 | 10K | 00:58:24 |
| Paul | 42 | 10K | 00:43:25 |
| Wendy | 32 | 5K | 00:19:33 |
| Dave | 35 | 5K | 00:25:02 |
| Paul | 42 | 5K | 00:24:46 |

## Issues with the Current DB

1. Repeated Data (data consistency)
2. Difficult to query
3. Scalability
4. Poor Security

# How will this session work?



# Introducing Relational Databases

*In relational databases…*

- There are multiple tables
- The relationships between tables are also stored
- It requires a Database Management System (DBMS)
- It uses SQL to manage data and the tables









# Introducing Relational Databases

## Flat-File Databases

| Runner | Age | Race | Time |
|---|---|---|---|
| Wendy | 32 | 10K | 00:42:19 |
| Dave | 35 | 10K | 00:58:24 |
| Paul | 42 | 10K | 00:43:25 |
| Wendy | 32 | 5K | 00:19:33 |
| Dave | 35 | 5K | 00:25:02 |
| Paul | 42 | 5K | 00:24:46 |

**Benefits**

- Lightweight
- Portable
- Easy to set-up
- Low cost
- Human-readable

**Drawbacks**

1. Repeated Data
2. Difficult to query
3. Scalability
4. Poor Security

**What have we gained and lost?**

| RunnerID | Name | Age |
|---|---|---|
| 1 | Wendy | 32 |
| 2 | Dave | 35 |
| 3 | Paul | 42 |

| RaceID | RaceType |
|---|---|
| 1 | 10K |
| 2 | 5K |

| RunnerID | RaceID | Time |
|---|---|---|
| 1 | 1 | 00:42:19 |
| 2 | 1 | 00:58:24 |
| 3 | 1 | 00:43:25 |
| 1 | 2 | 00:19:33 |
| 2 | 2 | 00:25:02 |
| 3 | 2 | 00:24:46 |

# What are Relational Databases good for?

## Strengths

- Data integrity
- Reliability
- Security (role-based access)
- Handling highly structured data
- Querying data

## Weaknesses

- Complexity (setup)
- Handling flexible data
- Handling (incredibly) large datasets
- Handling files
- Performance



# Relationships in Relational Databases

The relationship defines how the different entities (tables) link together.

There are **three different types**:

- One-to-One



- One-to-Many



- Many-to-Many



## Examples

One-To-One: one runner has one name

One-To-Many: one location may have many races

Many-To-Many: many runners may run in many races



# How do Relationships work?

The **Primary key** is an attribute used to identify a record in a table.

They must be **unique** and must have a value (**not null**).

| RunnerID | Name | Age |
|---|---|---|
| 1 | Wendy | 32 |
| 2 | Dave | 35 |
| 3 | Paul | 42 |

| RaceID | RaceType |
|---|---|
| 1 | 10K |
| 2 | 5K |

A **foreign key** is an attribute which is the primary key from another table.

The foreign key links that table to the table of the primary key.

| RunnerID | RaceID | Time |
|---|---|---|
| 1 | 1 | 00:42:19 |
| 2 | 1 | 00:58:24 |
| 3 | 1 | 00:43:25 |
| 1 | 2 | 00:19:33 |
| 2 | 2 | 00:25:02 |
| 3 | 2 | 00:24:46 |

# Representing Different Relationship Types

- ***One-to-One*** is the simplest to implement
- You can simple add it as a field for that table

| RunnerID | Name | Age |
|---|---|---|
| 1 | Wendy | 32 |
| 2 | Dave | 35 |
| 3 | Paul | 42 |

| id | runs | wicket_id |
|---|---|---|
| 12846 | 2 | |
| 12847 | 0 | 815 |
| 12848 | 0 | |
| 12849 | 6 | |
| 12850 | 0 | |
| 12851 | 0 | 816 |
| 12852 | 6 | |
| 12853 | 1 | |
| 12854 | 0 | 817 |
| 12855 | 1 | |
| 12856 | 0 | 818 |
| 12857 | 0 | |

- **One-to-Many** involves having separate tables for each entity
- The ‘One’ contains a **foreign key** based on the **primary key** in the ‘Many’
- Now, many rows in ‘One’ can reference ‘Many’

| RaceID | RaceType | Location |
|---|---|---|
| 1 | 10K | 1 |
| 2 | 5K | 1 |

| LocationID | Location |
|---|---|
| 1 | Lancaster |
| 2 | Manchester |



# Representing Different Relationship Types

**Many-To-Many** is the most complicated relationship to represent…

The main approach to representing this involves the creation of a third table, called a '***junction table***'.

The *junction table* has a foreign key for each entity, and then any data associated with the **Many-to-Many** relationship

| RunnerID | Name | Age |
|---|---|---|
| 1 | Wendy | 32 |
| 2 | Dave | 35 |
| 3 | Paul | 42 |

| RaceID | RaceType |
|---|---|
| 1 | 10K |
| 2 | 5K |

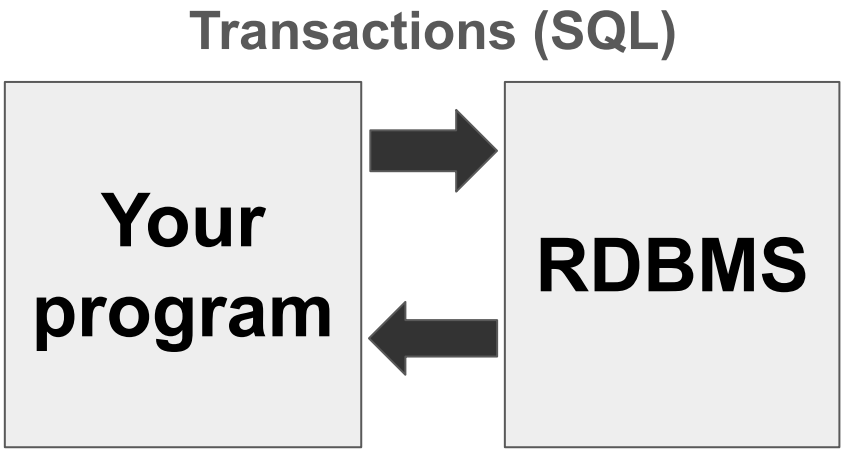
| RunnerID | RaceID | Time |
|---|---|---|
| 1 | 1 | 00:42:19 |
| 2 | 1 | 00:58:24 |
| 3 | 1 | 00:43:25 |
| 1 | 2 | 00:19:33 |
| 2 | 2 | 00:25:02 |
| 3 | 2 | 00:24:46 |



# Introducing RDBMS

***RDBMS stands for Relational Database Management System***

- It is usually takes the form of an application
- They follow Client-Server architecture
- Allows you to create, manage and query databases through SQL
- Allow operation with multiple users
- Enforce a rigid schema, for what can and can't be stored in the database
- **Use transactions (ACID)**






# Introducing Transactions

***A transaction is a series of one or more commands. These commands can be treated as a single update to the database made all at once.***

**ACID Properties**

***Atomicity:*** The transaction must be processed in full or not-at-all

***Consistency:*** The transaction strictly obeys all the rules of the database

***Isolation:*** Transactions execute independently without interfering each other

***Durability:*** Committed changes are permanent

```
BEGIN;

UPDATE accounts
SET balance = balance - 1000
WHERE account_id = 101;

UPDATE accounts
SET balance = balance + 1000
WHERE account_id = 102;

COMMIT;
```

***When you don't want to commit, use* 'ROLLBACK;'**



# Creating Schema

```sql
-- Creating the table for users
CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    username varchar(50) UNIQUE NOT NULL,
    password CHAR(40) NOT NULL,
    role user_role NOT NULL DEFAULT 'USER',
    email varchar(50) UNIQUE NOT NULL,
    name varchar(50),
    mobile varchar(50)
);
```

```sql
-- Creating the enum (essentially a datatype) for roles
CREATE TYPE user_role AS ENUM (
    'ADMIN',
    'USER'
);
```



# Numeric Data Types in Postgres

| Name | Description | Range |
|---|---|---|
| smallint | 2-byte integer | -32,768 to 32,767 |
| integer | 4-byte integer | -2,147,483,648 to 2,147,483,647 |
| bigint | 8-byte integer | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| decimal(p,s) | Exact numeric value with precision and scale | $-10^{38} + 1$ to $10^{38} - 1$ |
| numeric(p,s) | Exact numeric value with precision and scale | $-10^{38} + 1$ to $10^{38} - 1$ |
| real | 4-byte floating-point number | Approx. $-3.4 \times 10^{38}$ to $3.4 \times 10^{38}$ |
| double precision | 8-byte floating-point number | Approx. $-1.8 \times 10^{308}$ to $1.8 \times 10^{308}$ |
| serial | Auto-incrementing integer | Same as integer |
| bigserial | Auto-incrementing large integer | Same as bigint |



# Character Data Types in Postgres

| Name | Description | Range |
| --- | --- | --- |
| char(n) | Fixed-length character type. | 1 to 8,000 |
| varchar(n) | Variable-length character type with a maximum length of n. | 1 to 8,000 |
| text | Variable-length character type with no specific length limit. | No limit |



# Datetime Data Types in Postgres

| Name | Description | Range |
|---|---|---|
| date | Stores calendar dates. | 4713 BC to 5874897 AD |
| time | Stores time of day. | 00:00:00 to 24:00:00 |
| timestamp | Stores date and time without time zone. | 4713 BC to 5874897 AD |
| timestamptz | Stores date and time with time zone. | 4713 BC to 5874897 AD |
| interval | Represents a span of time. | -178,000 to 178,000 years |



# Creating relationships

```
-- Creating the table for game game_ownership
CREATE TABLE IF NOT EXISTS game_ownership (
    game_id INTEGER REFERENCES games(id),
    user_id INTEGER REFERENCES users(id),
    PRIMARY KEY (game_id, user_id)
);
```

Constraint: foreign key

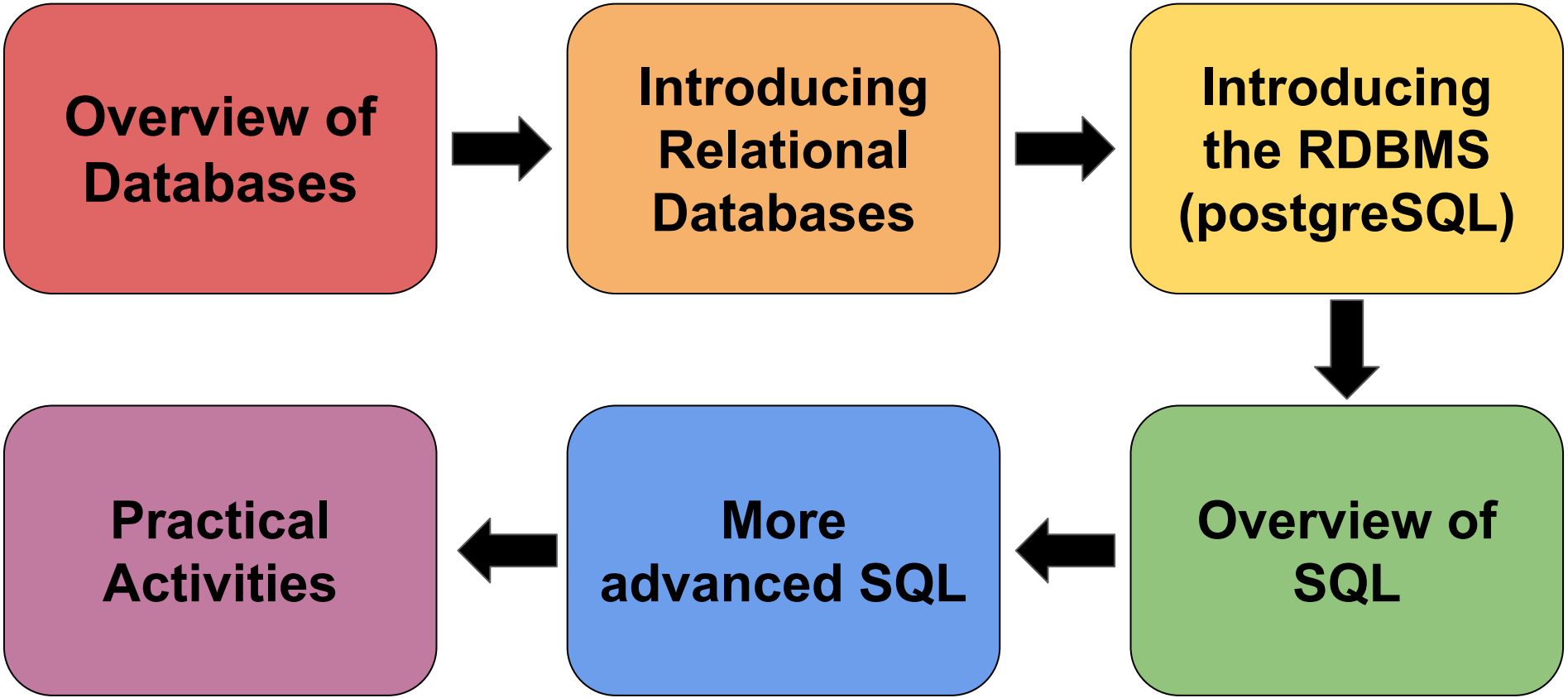
Creating a column

***Referential Integrity***

- The foreign key must be the same type as the referred primary key
- Foreign keys must match an **actual** row
- A row that is referenced cannot be removed without the associated rows also being removed

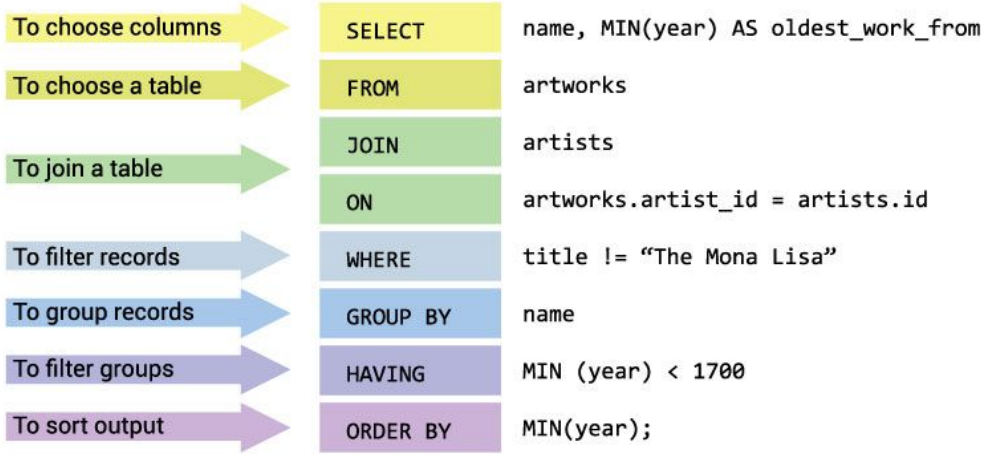


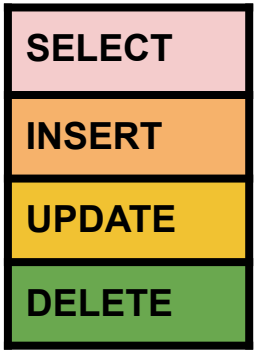
# How will this session work?



# Introduction to SQL

- **Stands for Structured Query Language**
- English-like
- While it is standardised, there are slight differences between different RDBMS
- It is **NOT** case-sensitive
- You can have as much whitespace and newlines as you'd like
- Use **single quotes (‘’)** for literal text – not needed for column or table names
- Use **‘--’** for comments






# SELECT Statements

```
running-demo=# SELECT id, name, date_of_birth FROM runners;
 id | name  | date_of_birth
----+-------+---------------
  1 | Wendy | 1994-10-29
  2 | Dave  | 1991-07-06
  3 | Paul  | 1984-05-12
(3 rows)
```

```
running-demo=# SELECT * FROM runners;
 id | name  | date_of_birth
----+-------+---------------
  1 | Wendy | 1994-10-29
  2 | Dave  | 1991-07-06
  3 | Paul  | 1984-05-12
(3 rows)
```



# SELECT Statements

```
running-demo=# SELECT id, name, AGE(CURRENT_DATE, date_of_birth) FROM runners;
 id | name  |            age
----+-------+---------------------------
  1 | Wendy | 32 years 11 days
  2 | Dave  | 35 years 4 mons 3 days
  3 | Paul  | 42 years 5 mons 28 days
(3 rows)
```

```
running-demo=# SELECT id+1 AS id, CONCAT(name,'hello') AS name FROM runners;
 id |    name
----+------------
  2 | Wendyhello
  3 | Davehello
  4 | Paulhello
(3 rows)
```



# Filtering with WHERE

```
running-demo=# SELECT * FROM runners WHERE name = 'Wendy';
 id | name  | date_of_birth
----+-------+---------------
  1 | Wendy | 1994-10-29
(1 row)
```

```
running-demo=# SELECT * FROM runners WHERE name = 'Wendy' OR name = 'Dave';
 id | name  | date_of_birth
----+-------+---------------
  1 | Wendy | 1994-10-29
  2 | Dave  | 1991-07-06
(2 rows)
```

```
running-demo=# SELECT * FROM runners WHERE name = 'Wendy' AND 1=1;
 id | name  | date_of_birth
----+-------+---------------
  1 | Wendy | 1994-10-29
(1 row)
```



# Operators in POSTGRES

| | |
|---|---|
| `=` | Equal to |
| `<` | Less than |
| `>` | Greater than |
| `<=` | Less than or equal to |
| `>=` | Greater than or equal to |
| `<>` | Not equal to |
| `!=` | Not equal to |
| `LIKE` | Check if a value matches a pattern (case sensitive) |
| `ILIKE` | Check if a value matches a pattern (case insensitive) |
| `AND` | Logical AND |
| `OR` | Logical OR |
| `IN` | Check if a value matches any value within a provided list |
| `BETWEEN` | Check if a value is within a specified range |
| `IS NULL` | Check if a value is NULL |
| `NOT` | Makes a negative result e.g. `NOT LIKE`, `NOT IN`, `NOT BETWEEN` |



# Controlling Results: ORDER

```
running-demo=# SELECT * FROM runners ORDER BY name;
 id | name  | date_of_birth
----+-------+---------------
  2 | Dave  | 1991-07-06
  3 | Paul  | 1984-05-12
  1 | Wendy | 1994-10-29
(3 rows)
```

```
running-demo=# SELECT * FROM runners ORDER BY name DESC;
 id | name  | date_of_birth
----+-------+---------------
  1 | Wendy | 1994-10-29
  3 | Paul  | 1984-05-12
  2 | Dave  | 1991-07-06
(3 rows)
```



# Controlling Results: LIMIT

```
running-demo=# SELECT * FROM runners LIMIT 2;
 id | name  | date_of_birth
----+-------+---------------
  1 | Wendy | 1994-10-29
  2 | Dave  | 1991-07-06
(2 rows)
```

```
running-demo=# SELECT * FROM runners ORDER BY id DESC LIMIT 1;
 id | name | date_of_birth
----+------+---------------
  3 | Paul | 1984-05-12
(1 row)
```

```
running-demo=# SELECT * FROM runners LIMIT 0;
 id | name | date_of_birth
----+------+---------------
(0 rows)
```



# Using Joins

*A reminder that we split this up into three different tables*

| Runner | Age | Race | Time |
|---|---|---|---|
| Wendy | 32 | 10K | 00:42:19 |
| Dave | 35 | 10K | 00:58:24 |
| Paul | 42 | 10K | 00:43:25 |
| Wendy | 32 | 5K | 00:19:33 |
| Dave | 35 | 5K | 00:25:02 |
| Paul | 42 | 5K | 00:24:46 |

**How could we ‘reconstruct’ this table in a single select?**



# Anatomy of a Join

| | | |
|---|---|---|
| To choose columns | SELECT | name, MIN(year) AS oldest_work_from |
| To choose a table | FROM | artworks |
| To join a table | JOIN | artists |
| | ON | artworks.artist_id = artists.id |
| To filter records | WHERE | title != “The Mona Lisa” |
| To group records | GROUP BY | name |
| To filter groups | HAVING | MIN (year) < 1700 |
| To sort output | ORDER BY | MIN(year); |



# Reconstructing the flatfile

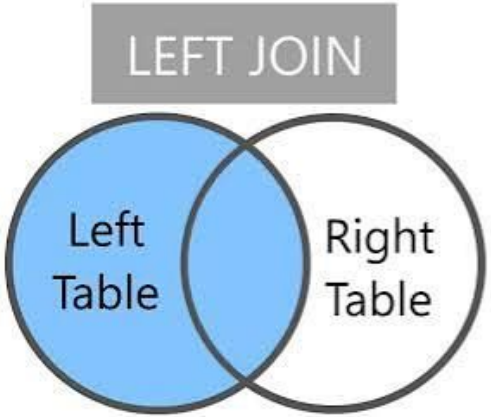
*A reminder that we split this up into three different tables*

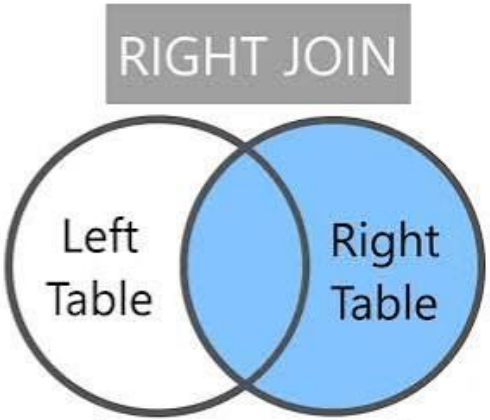
```
SELECT runners.name,
AGE(CURRENT_DATE,
runners.date_of_birth),
races.distance,
race_entries.time
FROM race_entries
JOIN runners
ON race_entries.runner_id = runners.id
JOIN races
ON race_entries.race_id = races.id;
```

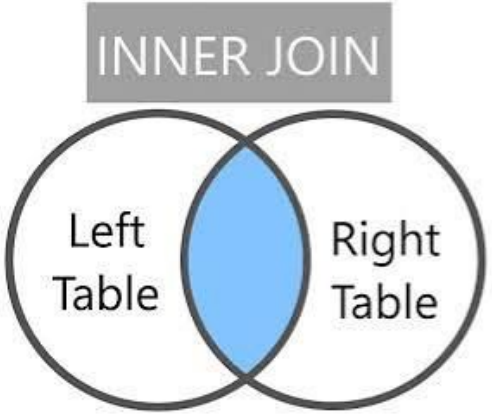
```
 name  |            age             | distance |   time
-------+----------------------------+----------+---------
 Wendy | 32 years 12 days           |    10000 | 00:42:1
 Dave  | 35 years 4 mons 4 days     |    10000 | 00:58:2
 Paul  | 42 years 5 mons 29 days    |    10000 | 00:43:2
 Wendy | 32 years 12 days           |     5000 | 00:19:3
 Dave  | 35 years 4 mons 4 days     |     5000 | 00:25:0
 Paul  | 42 years 5 mons 29 days    |     5000 | 00:24:4
(6 rows)
```

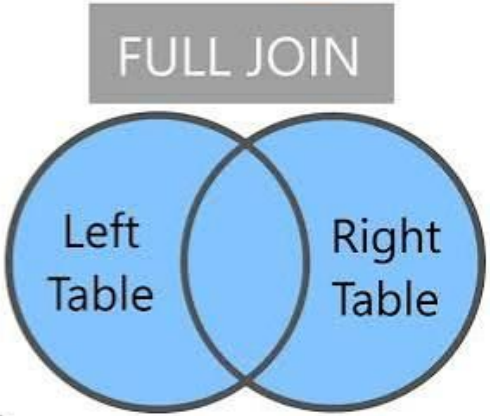


# Join Types




**INNER JOIN** is the default join in PostgreSQL (and most RDBMS).
Therefore, you can just use **JOIN** and get the inner join



# Aggregate Functions

*Aggregate functions compress multiple rows into one (aggregation)*

**COUNT()**
- Counts the number of rows
- Pass in a specific column
- Count(*) / Count(1)

**SUM()**
- Pass in a specific columns
- Sums the value in the column

**GROUP BY** [column]
- The column provided are the values that remain the same



# Aggregate Functions

Table: Customers

| customer_id | first_name | last_name | age | country |
|---|---|---|---|---|
| 1 | John | Doe | 31 | USA |
| 2 | Robert | Luna | 22 | USA |
| 3 | David | Robinson | 22 | UK |
| 4 | John | Reinhardt | 25 | UK |
| 5 | Betty | Doe | 28 | UAE |

```
SELECT country, COUNT(*) AS number
FROM Customers
GROUP BY country;
```

| country | number |
|---|---|
| UAE | 1 |
| UK | 2 |
| USA | 2 |



# HAVING

Table: Customers

| customer_id | first_name | last_name | age | country |
|---|---|---|---|---|
| 1 | John | Doe | 31 | USA |
| 2 | Robert | Luna | 22 | USA |
| 3 | David | Robinson | 22 | UK |
| 4 | John | Reinhardt | 25 | UK |
| 5 | Betty | Doe | 28 | UAE |

```
SELECT country, COUNT(*) AS number
FROM Customers
GROUP BY country;
```

| country | number |
|---|---|
| UAE | 1 |
| UK | 2 |
| USA | 2 |

**HAVING** is used to filter the results of an aggregate, instead of **WHERE**

