# Learning Objectives

### Learners will be able to...

- Explain why indexes are needed in a database
- Create an index
- Describe how transactions work
- Explain how triggers function in a database

info

## Make Sure You Know

Proficiency in writing SQL statements is important for creating indexes, triggers, and working with transactions. Learners should be comfortable with SQL syntax and its various clauses.

# Indexes

Database systems typically use indexes to provide fast access to relational data. An index is a separate physical data structure that allows you to quickly access one or more rows of data.

If we draw an analogy between a database and a book, indexes can be considered the table of contents of the book and the subject index.

But adding indexes everywhere to speed up queries is not the right solution: it helps speed up SELECT queries with a WHERE clause, but slows down data entry with UPDATE and INSERT statements. Indexes can be created or dropped without any impact on the data.

> info
>
> ## Indexes are more than just a quick lookup
>
> Indexes also serve to support some integrity constraints, such as uniqueness and primary keys.

Depending on the type of data being indexed, different approaches are taken for indexing. By default, when creating an index, a B-tree index is used.

Also, the implementation and types of indexes may differ from the used DBMS.

PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST, GIN, BRIN.

### B-Tree

B-trees can handle equality and range queries on data that can be sorted into some ordering. Supported operations:

```
<   <=   =   >=   >
```

### Hash

Hash indexes store a 32-bit hash code derived from the value of the indexed column. Supported operations is only equality =.

### GiST

GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented. Supported operations:

```
<<   &<   &>   >>   <<|   &<|   |&>   |>>   @>   <@   ~=   &&
```

### SP-GiST

SP-GiST indexes, like GiST indexes, offer an infrastructure that supports various kinds of searches.

Supported operations:

```
<<   >>   ~=   <@   <<|   |>>
```

### GIN

GIN indexes are “inverted indexes” which are appropriate for data values that contain multiple component values, such as arrays.

```
<@   @>   =   &&
```

### BRIN

BRIN indexes (a shorthand for Block Range INdexes) store summaries about the values stored in consecutive physical block ranges of a table.

```
<   <=   =   >=   >
```

# CREATE INDEX

## Create index

The add index command looks like this:

```
CREATE INDEX index_name ON table_name (column1, column2, ...);
```

For example, if a frequent operation on the movie table is to select by release year, then the following index can be created to optimize such queries.

```
CREATE INDEX release_year_index ON film (release_year);
```

Indexes can be either single or composite. In the second case, we list several columns separated by commas.

## Indexes and ORDER BY

In addition to simply looking up the rows to be returned by the query, the index can deliver them in a specific sorted order. This allows the ORDER BY query specification to be met without a separate sort step.

By default, B-tree indexes store their entries in ascending order with nulls last. This means that a forward scan of an index on column x produces output satisfying ORDER BY x (or more verbosely, ORDER BY x ASC NULLS LAST). The index can also be scanned backward, producing output satisfying ORDER BY x DESC.

```
CREATE INDEX release_year_index ON film (release_year DESC NULLS
        LAST);
```

## Delete index

To remove an index, use the `DROP INDEX` command. Indexes can be added to and removed from tables at any time.

```
DROP INDEX index_name;
```

To delete a previously created index, we can use the command.

```
DROP INDEX release_year_index;
```

# Transactions

SQL transactions are a group of sequential database operations, which is an atomic unit of work with data. In other words, transactions allow us to control the integrity of the data.

Transactions have the following four standard properties, usually referred to by the acronym ACID.

- **Atomicity** — guarantees that any transaction will be committed only in its entirety (completely). If one of the operations in the sequence fails, then the entire transaction will be cancelled.
- **Consistency** — ensures that any completed transaction commits only valid results.
- **Isolation** — each transaction must be isolated from others, i.e. its result should not depend on the execution of other parallel transactions.
- **Durability** — ensures that the result or effect of a committed transaction persists in case of a system failure.

### Transaction management

The following commands are used to manage transactions:

- **BEGIN** - open transaction.
- **COMMIT** - save changes.
- **ROLLBACK** - undo changes.
- **SAVEPOINT** - creates savepoints in transaction groups.
- **SET TRANSACTION** - puts a name in a transaction.

### Transaction example

For example, we go to rent a DVD. With respect to the database, these are two insert operations into `rental` and `payment` tables. If, as a result of a failure, only the first operation succeeds, and the second one fails, inconsistent data will be stored in the database. Executing operations in a transaction will avoid this.

```
BEGIN;

  INSERT INTO rental(rental_id, rental_date, inventory_id,
        customer_id, staff_id)
  VALUES(16061, NOW(), 10, 3, 1);

  INSERT INTO payment (customer_id, staff_id, rental_id, amount,
        payment_date)
  VALUES(3, 1, 16061, 10, NOW());

COMMIT;
```

# Triggers

Triggers serve as powerful tools in database management systems, initiating specific actions in response to predefined events. Acting as catalysts for change, triggers play a crucial role in orchestrating a sequence of events within a database.

Triggers are used to tell the DBMS to perform an action when a certain event occurs. It turns out a kind of catalyst for change, a trigger that starts a chain of events.

The trigger must be associated with the specified table, view (pseudo table), or external table. It runs its part of the code only when performing operations on this entity - **INSERT**, **UPDATE**, **DELETE** or **TRUNCATE**. Depending on the requirements, we can fire the trigger before, after or instead of the event/operation.

Triggers are divided into two types depending on the level at which they act.

If the trigger is marked with the **FOR EACH ROW** option, then the function is called for each row that changes as a result of the event. For example, if you do an UPDATE on 100 rows, the UPDATE trigger function will be called 100 times, once for each updated row.

The **FOR EACH STATEMENT** option will only call the function once per statement, regardless of the number of rows being changed.

Writing triggers may vary depending on the DBMS used.

An example of a trigger that will update field `last_update` when a record is changed.

```
CREATE OR REPLACE FUNCTION last_update_rental_function() RETURNS
        trigger
    AS $$
BEGIN
    NEW.last_update = CURRENT_TIMESTAMP;
    RETURN NEW;
END $$
    LANGUAGE plpgsql;


CREATE OR REPLACE TRIGGER last_update_rental_trigger
    BEFORE UPDATE ON rental
    FOR EACH ROW
    EXECUTE PROCEDURE last_update_rental_function();
```