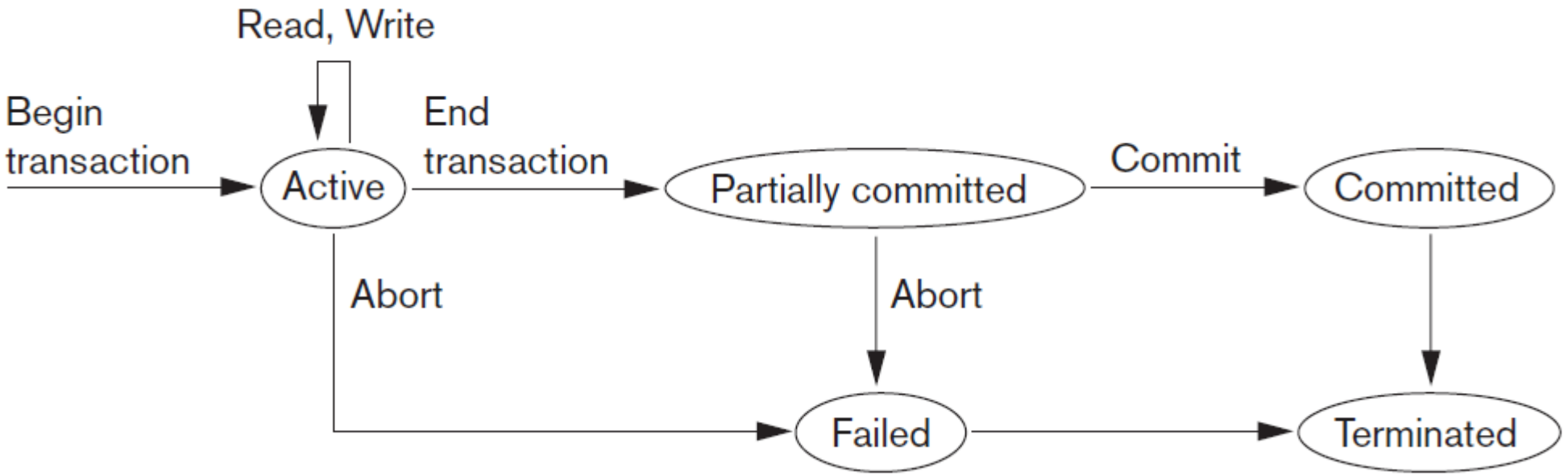
Transactions

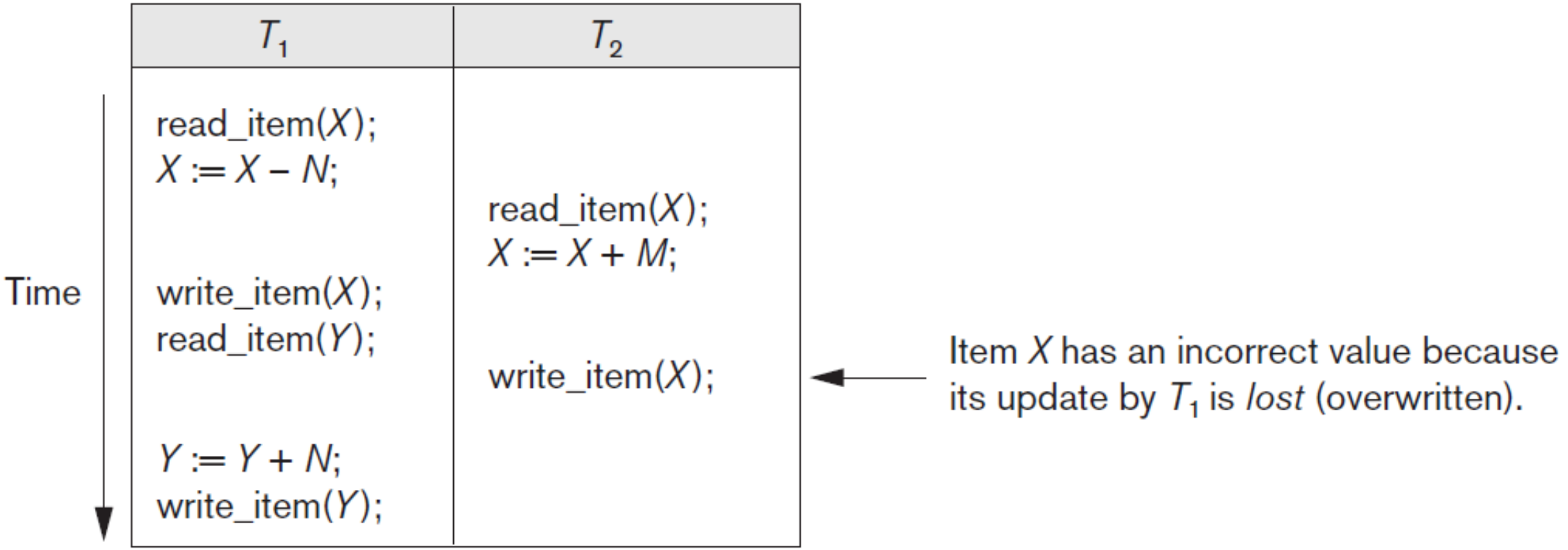
```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

- → What are transactions?
  - → An all or nothing operation
  - → A way to ensure consistency when multiple users are using a database system
  - → Entire transaction is executed, and not committed to the database before a COMMIT command is executed.

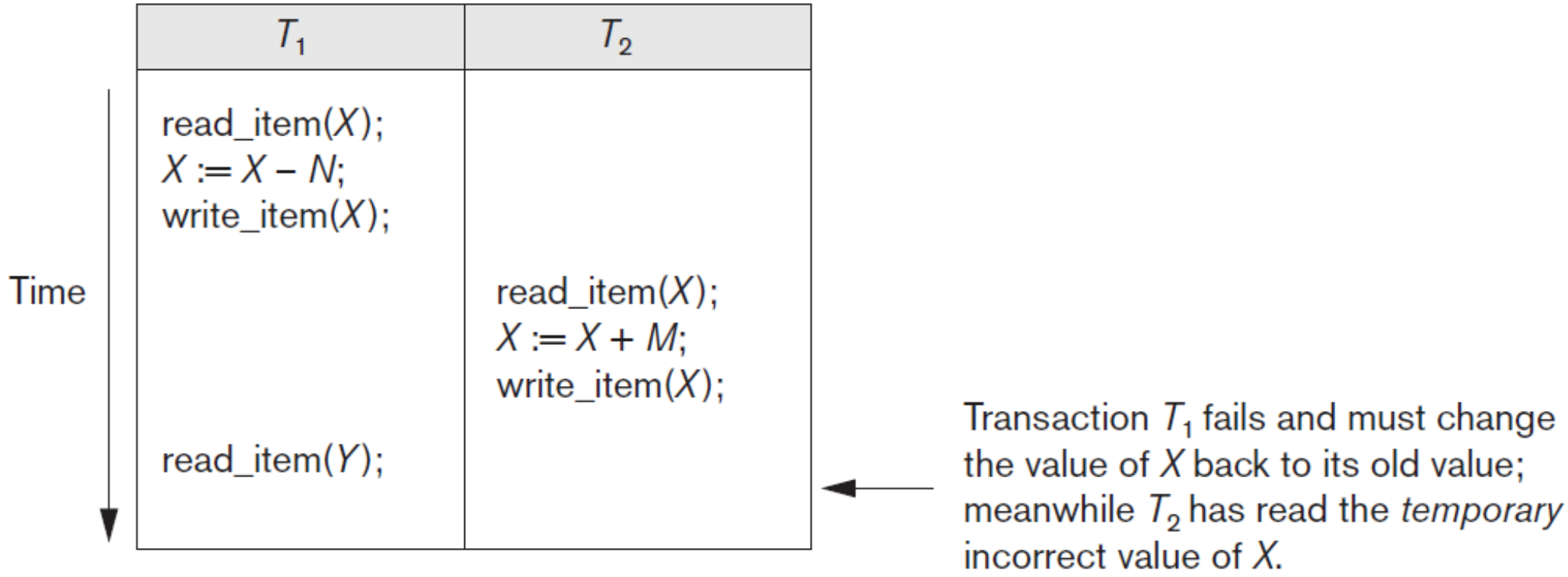
Commit Process



The Lost Update Problem



The Temporary Update (or Dirty Read) Problem



The Incorrect Summary Problem

T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	• • •

read_item(*X*);
X := *X* − *N*;
write_item(*X*);	

	read_item(*X*);
	sum := *sum* + *X*;
	read_item(*Y*);
	sum := *sum* + *Y*;

read_item(*Y*);
Y := *Y* + *N*;
write_item(*Y*);	

← T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Serial transaction

→ Typically the default.

→ Blocks other queries/transactions until one is finished.

(a)

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

(b)

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule B

Nonserial, and Conflict-Serializable Schedules

→ Combines transactions

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

Schedule C

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule D

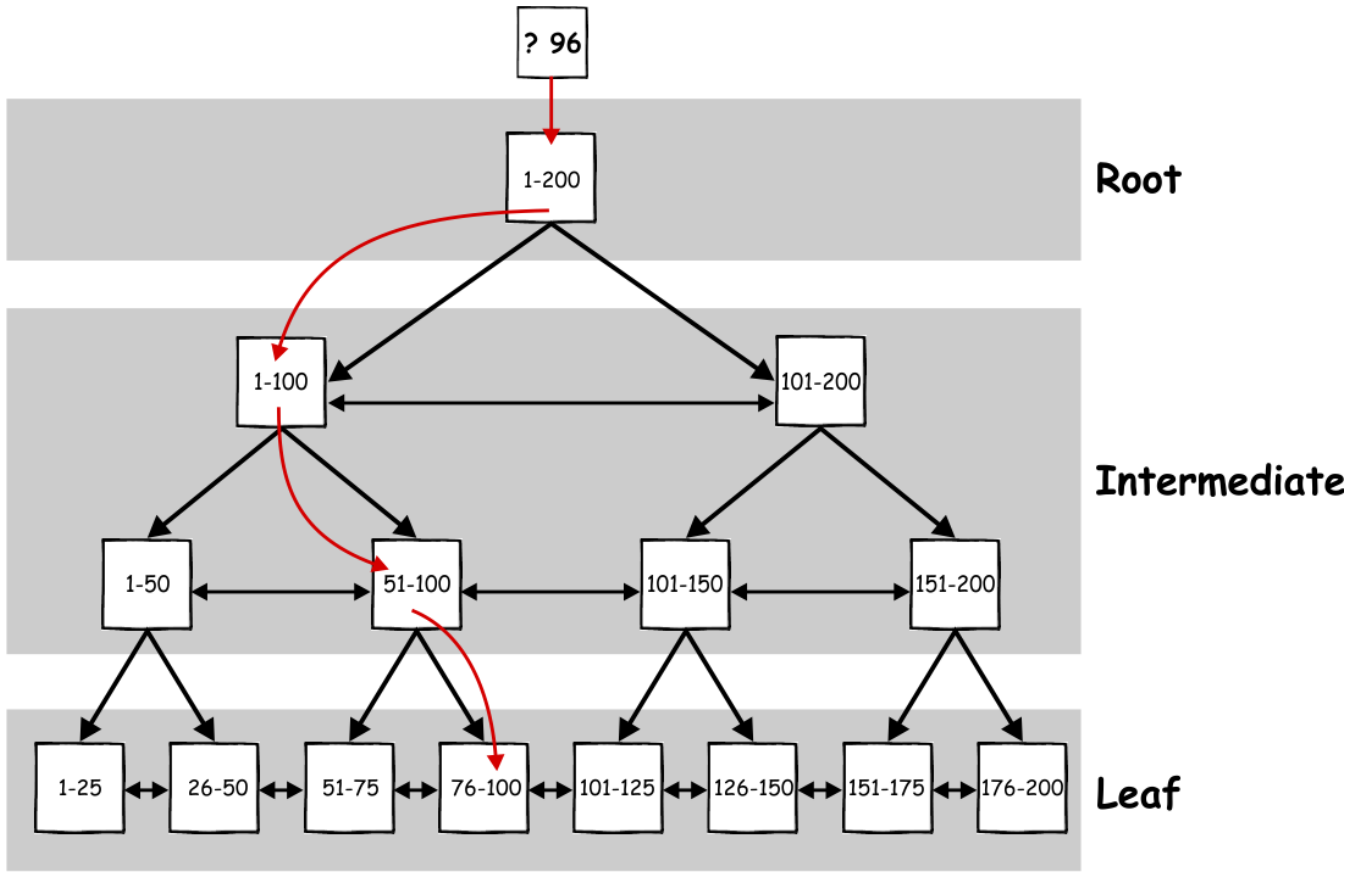
Transaction Commands

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

- → Commands:
  - → BEGIN
  - → SAVEPOINT
  - → ROLLBACK
  - → COMMIT
- → Watch out for not stalling the server with multiple BEGIN's that are never rolled back or committed.

Indexes

Indexes



- → What is an index?
  - → You already use it in programing
    - → Arrays, Maps, hash tables.
- → Used for performance optimization
- → Great when often finding records by something else than your primary key
- → Takes up more space on the disk

Creating Indexes

```
CREATE TABLE books(
    id serial PRIMARY KEY,
    title VARCHAR (250) NOT NULL,
    isbn VARCHAR (20) UNIQUE,
    price float
);

-- insert 2 million books

-- searching for book using ISBN instead of id - return time 2 min
SELECT * FROM books WHERE isbn = '978-12-92097-61-9';

-- slow result with that many entries

-- Creating
CREATE INDEX ON books(isbn);

-- now searching again - return time 5 seconds
SELECT * FROM books WHERE isbn = '978-12-92097-61-9';
```

Stored Procedures

Stored Procedures
(and Functions)

```
//Function PSM1:
0) CREATE FUNCTION Dept_size(IN deptno INTEGER)
1) RETURNS VARCHAR [7]
2) DECLARE No_of_emps INTEGER ;
3) SELECT COUNT(*) INTO No_of_emps
4) FROM EMPLOYEE WHERE Dno = deptno ;
5) IF No_of_emps > 100 THEN RETURN "HUGE"
6) ELSEIF No_of_emps > 25 THEN RETURN "LARGE"
7) ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"
8) ELSE RETURN "SMALL"
9) END IF ;
```

- → Several Languages can be used, depending on what database is used.
  - → Examples include: Java, Python, C# etc.
- → Most support SQL/PSM
  - → Completely portable, high-performance transaction-processing language.
  - → Generally supported directly inside SQL
- → The main functional difference between a function and a stored procedure is that a function returns a result, whereas a stored procedure does not.
- → All examples used after this slide are in PL/pgSQL, as we are using PostgreSQL

IF Structure

```
IF <condition> THEN <statement list>
    ELSEIF <condition> THEN <statement list>
    ...
    ELSEIF <condition> THEN <statement list>
    ELSE <statement list>
END IF ;
```

WHILE Structure

```
WHILE <condition> DO
        <statement list>
END WHILE ;
```

```
REPEAT
        <statement list>
UNTIL <condition>
END REPEAT ;
```

FOR Structure

```
FOR <loop name> AS <cursor name> CURSOR FOR <query> DO
        <statement list>
END FOR ;
```

Stored Procedure Example 1/2

```
CREATE OR REPLACE PROCEDURE update_department_size(department_number INTEGER)
AS $$
DECLARE
    number_of_department_members integer := 0;
BEGIN
    SELECT COUNT(*) INTO number_of_department_members
    FROM department_members WHERE department_id = department_number;

    UPDATE departments SET number_of_members = number_of_department_members
    WHERE id = department_number;
END; $$
LANGUAGE plpgsql;
```

Stored Procedure Example 2/2

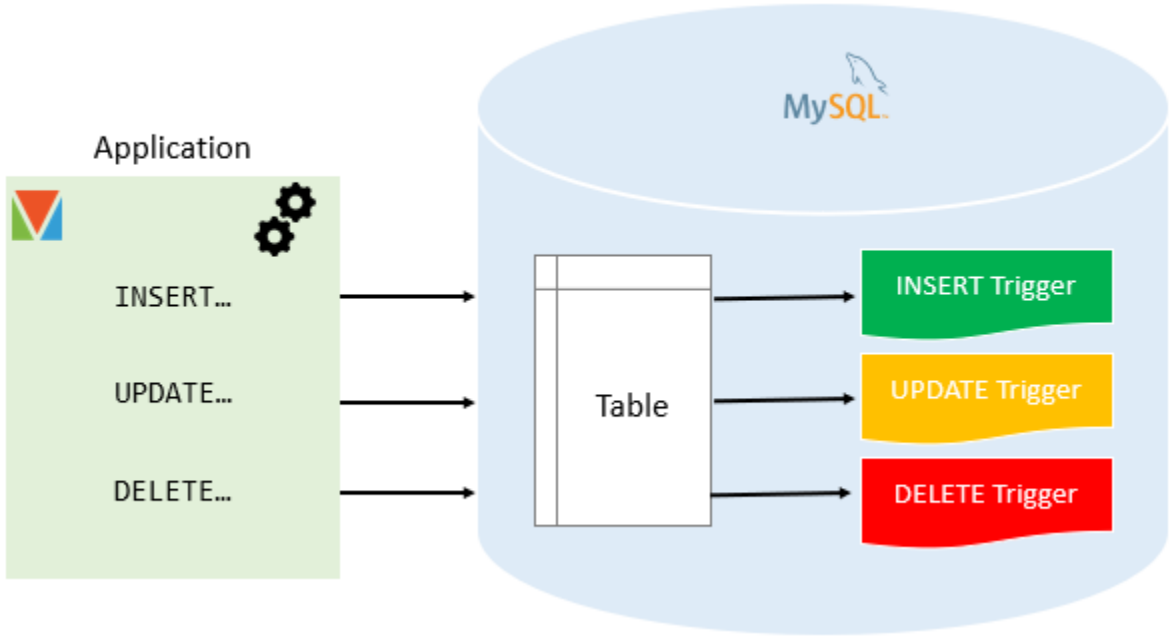
```
CREATE OR REPLACE PROCEDURE update_all_department_sizes()
AS $$
DECLARE
    departments CURSOR FOR SELECT DISTINCT(id) AS id FROM departments;
BEGIN
    FOR department in departments LOOP
        CALL update_department_size( department_number: department.id);
    END LOOP;
END; $$
LANGUAGE plpgsql;
```

Functions & Triggers

Function Example

```
CREATE OR REPLACE FUNCTION update_all_department_sizes_trigger()
    RETURNS TRIGGER
AS $$
BEGIN
    CALL update_all_department_sizes();
    RETURN NEW;
END; $$
LANGUAGE plpgsql;
```

Triggers



→ A way to execute functions or stored procedures if a change is happening in the database

Trigger Example 1

```
CREATE OR REPLACE FUNCTION update_all_department_sizes_trigger()
    RETURNS TRIGGER
AS $$
BEGIN
    CALL update_all_department_sizes();
    RETURN NEW;
END; $$
LANGUAGE plpgsql;
```

```
CREATE TRIGGER update_number_of_members_trigger
    AFTER INSERT OR DELETE ON department_members
    EXECUTE PROCEDURE update_all_department_sizes_trigger();
```

Trigger Examples 2

```
CREATE OR REPLACE FUNCTION log_last_name_changes()
  RETURNS trigger AS
$BODY$
BEGIN
   IF NEW.last_name <> OLD.last_name THEN
       INSERT INTO employee_audits(employee_id,last_name,changed_on)
       VALUES(OLD.id,OLD.last_name,now());
   END IF;

   RETURN NEW;
END;
$BODY$
```

```
CREATE TRIGGER last_name_changes
  BEFORE UPDATE
  ON employees
  FOR EACH ROW
  EXECUTE PROCEDURE log_last_name_changes();
```

Triggered by:

```
INSERT INTO employees (first_name, last_name)
VALUES ('John', 'Doe');

INSERT INTO employees (first_name, last_name)
VALUES ('Lily', 'Bush');
```

Types of triggers

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	—	—

Trigger types example 1/2

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

```
CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE PROCEDURE check_account_update();
```

- → A: Execute the function check_account_update whenever a row of the table accounts is about to be updated
- → B: The same, but only execute the function if column balance is specified as a target in the UPDATE command
- → C: This form only executes the function if column balance has in fact changed value

Trigger types example 2/2

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE PROCEDURE log_account_update();
```

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE PROCEDURE view_insert_row();
```

→ A: Call a function to log updates of accounts, but only if something changed.

→ B: Execute the function view_insert_row for each row to insert rows into the tables underlying a view