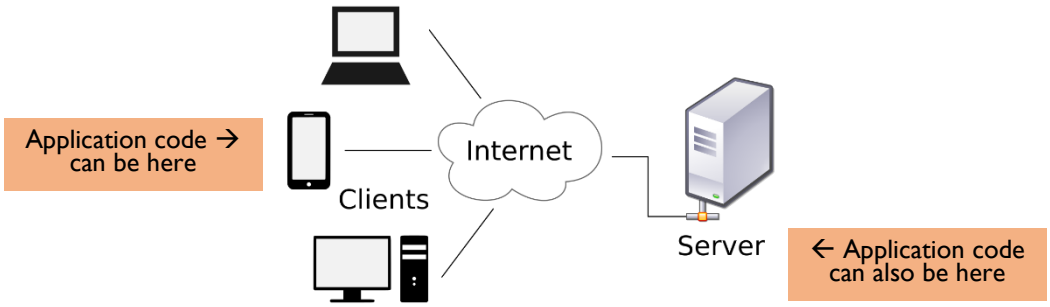
Stored procedures and triggers

- Following notes not a comprehensive representation of underlying concepts, syntax differs from system to system

Stored procedures/functions

Most DBMS have a client-server architecture:



- A design choice *where* to run application code (e.g. php/javascript):

Client side	**Server** (i.e. the database)
Easier on server	Allows stronger computational power
	To consolidate integrity constraints

- *PostgreSQL* (and other DBMS) can store, share and execute code on the server
- *PL/pgSQL*:
  - *PostgreSQL*'s language designed to seamlessly embed SQL code and interact with database SQL code
  - Partially complies with the SQL standard
  - Similar to *Oracle PL/SQL, SQLServer Transact-SQL* and *DB2 SQL Procedural Language* and other languages implementing variants of the Persistent Stored Modules portion of the SQL standard (i.e. different for different DBMS, cannot port code between DBMS directly)
- **Performance:**
  - Stored procedures are compiled
  - Code is cached and shared by all users
  - Code is executed on server's side (usually a powerful machine), generally incurring fewer data transfers across the network
- **Productivity:**
  - Stored procedures are shared and reused under access control
  - Application logic they encode is implemented and maintained in a single place
- **Security:**
  - Data manipulation can be restricted to calling stored procedures under strict access control

CREATE FUNCTION	Creates a function	
	CREATE OR REPLACE FUNCTION gst(val NUMERIC) RETURNS NUMERIC AS 'BEGIN RETURN val * 1.07; END;' LANGUAGE PLPGSQL; SELECT gst(1); SELECT g.name, gst(g.price) FROM game g WHERE gst(g.price) < 5;	- Function definition is between **single quotes** - *LANGUAGE* is *PLPGSQL*, and can also be *C, Perl, Python, tcl*
Example function **(1)**	CREATE OR REPLACE FUNCTION hello() RETURNS CHAR(5) AS $$ BEGIN RETURN 'Hello World' ; END; $$ LANGUAGE PLPGSQL; -- Returned as a result in a table -- i.e. String in a table of one entry SELECT hello(); -- Function called on **every** row in the table -- 430 rows of 'Hello World' returned SELECT hello() FROM games g;	- Function definition is between $$ or $<*name*>$ - Otherwise, quotes in the function definition need to be escaped - Available types include: SQL domains *NUMERIC, VARCHAR()* and *tablename%ROWTYPE, tablename.columnname%TYPE, RECORD*
Example function **(2)** – Catching error with ***RAISE NOTICE***	DROP FUNCTION hello(); CREATE OR REPLACE FUNCTION hello() RETURNS BOOLEAN AS $$ BEGIN RAISE NOTICE 'hello' ; RETURN TRUE; END; $$ LANGUAGE PLPGSQL; -- Output: rows of 'TRUE' -- Messages: 'NOTICE: hello' SELECT hello() FROM games g;	- *RAISE NOTICE* prints on the database console

Example function **(3) –** Reading from the database	CREATE TABLE gst(gst NUMERIC); INSERT INTO gst VALUES (7); CREATE OR REPLACE FUNCTION gst(val NUMERIC) RETURNS NUMERIC AS $$ DECLARE gst1 NUMERIC; BEGIN SELECT g.gst/100 INTO gst1 FROM gst g; RETURN val * (1+gst1); END; $$ LANGUAGE PLPGSQL; SELECT g.name, gst(g.price) FROM games g;	- Variable *gst1* declared - SQL integrates with PL/pgSQL
CONTROL STRUCTURES	IF condition THEN ... ELSIF condition ... THEN ... ELSE ... END IF;	
	FOR somevariable IN (number ... number) LOOP ... EXIT EXIT WHEN END LOOP:	

Example function **(4) –** Control structures	CREATE OR REPLACE FUNCTION gst(val NUMERIC)
	RETURNS NUMERIC AS $$
	DECLARE gst1 NUMERIC;
	BEGIN
	SELECT g.gst/100 INTO gst1 FROM gst g;
	IF val * (1 + gst) > 5 THEN
	RETURN val * 1 + gst1;
	ELSE
	RETURN 5;
	END; $$
	LANGUAGE PLPGSQL;
	
	SELECT g.name, gst(g.price),
	FROM games g;
		Function returns items at the minimum of $5

CURSORS

- Allows handling of data one-by-one, churn-by-churn instead of *huge* rows of data at once (especially when database is large and querying returns a large number of rows)
- Cursors are the scalable way to process data from a query

SCROLL, NO SCROLL,

indicates whether a cursor can be scrolled backwards or not respectively

- A cursor can move in different directions and modes:

NEXT, LAST, PRIOR, FIRST,
ABSOLUTE, RELATIVE,
FORWARD, BACKWARD

- Cursors **must be closed** **

Example function **(5) –** Control structures

```
CREATE OR REPLACE FUNCTION
                    avg1(appname VARCHAR(32))
RETURNS NUMERIC AS $$
DECLARE mycursor SCROLL CURSOR (vname
                                VARCHAR(32))
FOR SELECT g.price FROM games g
WHERE g.name = vname;
price NUMERIC; avgprice NUMERIC;
count NUMERIC:
BEGIN
OPEN mycursor(vname:=appname);
avgprice:= 0; count:= 0; price:= 0;
LOOP
      FETCH mycursor INTO price;
      EXIT WHEN NOT FOUND;
      avgprice:= avgprice + price;
      count:= count + 1;
END LOOP:
CLOSE mycursor;
IF count < 1 THEN RETURN NULL:
ELSE RETURN round(avgprice/count, 2);
END IF;
END; $$
LANGUAGE PLPQSQL;

SELECT avg1( 'Aerified' );
SELECT name, avg1(name) FROM games g;
SELECT g.name, AVG(g.price) FROM games g
GROUP BY g.name;
```

- Cursor goes through result one-by-one (returns on the client side)
- Looping tables in the database
- Using *avg1* since there is already a *AVG* function in SQL

Triggers

- Triggers program the reaction to events happening to the database
- E.g. Used for integrity constraints, any modification triggers the checking whether constraints are kept (which is not really implemented during creation of the table)
- Modification include *insert, update, delete*
- A trigger is a procedure or function that is executed when a database events occurs on a table: *INSERT, DELETE, UPDATE, CREATE TABLE* etc.
- Used to maintain integrity, propagate updates and repair the database (they are a generalisation of *ON UPDATE/DELETE*)
- Syntax and semantics differ from one DBMS to the next
- ☹ May cause changes that trigger other triggers
- **Performance:**
  - Triggers are compiles
  - Code is cached and shared by all users
  - Code is executed on the server's side, usually a powerful machine
  - Incurs no data transfer across networks
- **Productivity:**
  - Applied to all interactions
  - Application logic they encode is implemented and maintained in a single place
  - Code is not portable from one DBMS to another
  - Interactions among triggers (chain reactions) and between triggers, constraints and transactions are difficult to control
- **Security:**
  - Data manipulation can be restricted and transformations automatically propagated
  - Very easy to make a mistake in the coding

CREATE TRIGGER	`CREATE TRIGGER name`
	`{BEFORE | AFTER | INSTEAD OF} {event [OR event]*}`
	`ON table`
	`-- statement by default`
	`[FOR [EACH] {ROW | STATEMENT}]`
	`-- for row only`
	`[WHEN condition]`
	`EXECUTE PROCEDURE function()`
		- Hard to debug, requires another action to trigger it
		- Option to call trigger once or multiple times for each row

Variables	`-- for row only`
	`NEW, OLD`
	`-- Others`
	`TG_WHEN( 'BEFORE' , 'AFTER' ),`
	`TG_OP( 'INSERT' , 'DELETE' , 'UPDATE' ,`
	`'TRUNCATE' )`	

Example trigger **(1)** – For each statement	`DROP FUNCTION hello() CASCADE;`
	
	`CREATE OR REPLACE FUNCTION hello()`
	`RETURNS TRIGGER AS $$`
	`BEGIN`
	`RAISE NOTICE 'hello' ;`
	`RETURN NULL;`
	`END; $$`
	
	`CREATE TRIGGER hello`
	`BEFORE INSERT OR UPDATE`
	`ON games`
	`FOR EACH STATEMENT`
	`EXECUTE PROCEDURE hello();`
	
	`INSERT INTO games VALUES ( 'A' , '5.1' , 100),`
	`( 'B' , '3.0' , 101),`
	`( 'C' , '3.0' , 102);`
	
	`SELECT * FROM games`
	`WHERE name = 'A' OR name = 'B'`
	`OR name = 'C' ;`
		- In PostgreSQL, a trigger executes a function of type *trigger*
		- In the example, the trigger sends a message to the database console before every insertion and update (for each statement) on the table
		- The message *'hello'* is **displayed once**
		- No row is inserted
		- Rows that are not caught by triggers are executed as normal

Example trigger **(2)** – For each row	DROP FUNCTION hello() CASCADE; CREATE OR REPLACE FUNCTION hello() RETURNS TRIGGER AS $$ BEGIN RAISE NOTICE 'hello' ; RETURN NULL; END; $$ LANGUAGE PLPGSQL; CREATE TRIGGER hello BEFORE INSERT OR UPDATE ON games FOR EACH ROW WHEN (NEW.price > 100) EXECUTE PROCEDURE hello(); INSERT INTO games VALUES ('AA' , '5.1' , 100), ('BB' , '3.0' , 101), ('CC' , '3.0' , 102); SELECT * FROM games WHERE name = 'AA' OR name = 'BB' OR name = 'CC' ;	- In the example, the trigger sends a message to the database console for **each row** of the table *apps* **affected** by an insertion or update if the new price is more than 100 - *RETURN NULL* prevents the insertion or update - When the row *(AA, 5.1, 100)* is inserted, the message *'hello'* is displayed twice and the other rows are not inserted
Example trigger **(3)** – ***RETURN NEW***	DROP FUNCTION hello CASCADE; CREATE OR REPLACE FUNCTION hello() RETURNS TRIGGER AS $$ BEGIN RAISE NOTICE 'hello' ; RETURN NEW; END; $$ LANGUAGE PLPGSQL; CREATE TRIGGER hello BEFORE INSERT OR UPDATE ON games FOR EACH ROW WHEN (NEW.price > 100) EXECUTE PROCEDURE hello(); INSERT INTO apps VALUES ('AAA' , '5.1' , 100), ('BBB' , '3.0' , 101), ('CCC' , '3.0' , 102); SELECT * FROM apps WHERE name = 'AAA' OR name = 'BBB' OR name = 'CCC' ;	- The insertion and update is done when the stored procedure returns *NEW* - In the example, 3 rows inserted, the message *'hello'* is displayed twice

Example trigger (4) – **LOGGING CHANGES**

```
CREATE TABLE glog (
name            VARCHAR (32) NOT NULL,
pricebefore     NUMERIC,
priceafter      NUMERIC NOT NULL,
date            DATE NOT NULL);

CREATE OR REPLACE FUNCTION pricelog()
RETURNS TRIGGER AS $$
DECLARE delta NUMERIC;
DECLARE pb NUMERIC;
DECLARE now DATE;
BEGIN
now := now();
IF TG_OP := 'INSERT' OR TG_OP := 'UPDATE'
THEN pb := NULL;
ELSE pb := OLD.price END IF;
INSERT INTO glog VALUES (NEW.name, NEW.version,
                         pb, NEW.price, now);
RETURN NULL;
END; $$
LANGUAGE PLPGSQL;

CREATE TRIGGER pricelog
AFTER INSERT OR UPDATE
ON games
FOR EACH ROW
EXECUTE PROCEDURE pricelog();
```

```
INSERT INTO games VALUES( 'AAAA' ,' 5.1' , 100),
                        ( 'BBBB' ,' 3.0' , 101),
                        ( 'CCCC' ,' 3.0' , 102);

SELECT * FROM games
WHERE names = 'AAAA' OR name = 'BBBB'
                     OR name = 'CCCC' ;

SELECT * FROM glog;

UPDATE games SET price = 110
WHERE name = 'AAAA' ;

SELECT * FROM games WHERE name = 'AAAA' ;
SELECT * FROM glog;
```

<table>
<tr><td></td><td>

```
CREATE OR REPLACE FUNCTION hello()
RETURNS TRIGGER AS $$
BEGIN RAISE NOTICE  'hello' ;
RETURN NULL;
END; $$
LANGUAGE PLGPSQL;

INSERT INTO games VALUES ( 'AAAA' ,' 5.1' ,120);
SELECT * FROM games WHERE name =  'AAAA' ;
SELECT * FROM glog;

DROP FUNCTION hello() CASCADE;

INSERT INTO games VALUES ( 'AAAA' , '5.1' ,130);
```

</td><td>- ** Triggers interact in alphabetical order
- The hello trigger and the hello() function prevented the change
- The hello trigger and hello() function is removed, but there is an integrity constraint violation</td></tr>
<tr><td rowspan="3">Note on constraints and transactions</td><td colspan="2">- Triggers interact with constraints but things are even more complicated when constraints are deferred
- Note that triggers cannot be deferred
- ** Always defer constraints (check constraint at the end)</td></tr>
<tr><td>

```
-- Example
CREATE TABLE test1 (
father NUMERIC PRIMARY KEY NOT DEFERRABLE,
son    NUMERIC REFERENCES test1(father)
       NOT DEFERRABLE,
);

CREATE TABLE test2 (
father NUMERIC PRIMARY KEY NOT DEFERRABLE,
son    NUMERIC REFERENCES test2(father)
       DEFERRABLE INITIALLY DEFFERED
);
```

</td><td>The foreign key constraint on table test1 is not deferred; the foreign key constraint on table test2 is deferred</td></tr>
<tr><td>

```
BEGIN;
INSERT INTO test1 VALUES (1,2);
INSERT INTO test1 VALUES (2,1);
END;

BEGIN;
INSERT INTO test2 VALUES (1,2);
INSERT INTO test2 VALUES (2,1);
END;

SELECT * FROM test2;
```

</td><td>- The transaction on table test1 is aborted
- The transaction on table test2 succeeds and is committed</td></tr>
</table>