

SQL Views & Constraints

Dr. Katharina Schneider & Dr. Tony Garnock-Jones
BCS1510

EPD150 MSM Conference Hall
23 April 2025

Review of Part I

1. Course introduction ✓
2. SQL: Relational model, **DDL**, **DQL**, **DML** ✓
 - Queries, subqueries, operators, set operators, ...
3. Advanced subqueries, aggregations (GROUP BY, HAVING)... ✓
4. Data modification and manipulation ✓
 - INSERT, DELETE, UPDATE, defaults, conditional operations...
5. Built-in functions ✓
6. **Today:** Views, Constraints, Triggers, more SQL functions

Learning Objectives

1. Views:

- virtual and materialized; as "subroutines" to simplify complex queries; for performance

2. Constraints:

- enforce data integrity; key constraints, foreign keys, attribute-based checks

3. Triggers: react to changes

4. A few more SQL functions: COALESCE, GREATEST, LEAST

Views

Views

A **view** is a *synthetic relation* defined in terms of

- other tables ("base tables") and
- other views

Views

A view can be

- **virtual**: essentially an alias for a query; the rows in a virtual view are not physically stored in the database
- **materialized**: actually physically constructed, stored, and kept up-to-date*

* Materialized views are only available in DBMSes like Oracle 🗄️ and PostgreSQL 🐘; not in lightweight DBMSes like MySQL or SQLite

Views

```
CREATE VIEW viewname AS  
  SELECT ... FROM ...;
```

```
DROP VIEW viewname;
```

Or, in Oracle etc., CREATE MATERIALIZED VIEW.

(**Warning:** Go look up REFRESH MATERIALIZED VIEW!
Now look up "Incremental View Maintenance" (IVM).
What makes IVM difficult?)

Views

```
CREATE VIEW model_sales AS
  SELECT model,
         SUM(quantity) AS num_sold,
         SUM(paid) AS revenue
  FROM sales
  GROUP BY model;
```


Views

```
SELECT *  
FROM model_sales;
```

model	num_sold	revenue
2010	1	2300.0
3001	1	99.0
2002	3	2847.0
3002	1	239.0
1001	1	1902.6
1007	5	2499.0
3007	2	360.0

- Query a view as if it were a table.
- Limited ability to UPDATE or INSERT into views!

MySQL manual §27.5.3: "complex rules"; "one-to-one relationship" between view and table rows; other conditions apply

Views

```
SELECT model, num_sold  
FROM model_sales;
```

model	num_sold
2010	1
3001	1
2002	3
3002	1
1001	1
1007	5
3007	2

Views

```
SELECT model,  
       revenue / num_sold  
       AS unit_price  
FROM model_sales;
```

model	unit_price
2010	2300.0
3001	99.0
2002	949.0
3002	239.0
1001	1902.6
1007	499.8
3007	180.0

Views: simplify complex queries

AVG(price)

1195.6666666666667

```
CREATE VIEW pc_a_pricelist AS
  SELECT model, price FROM pcs WHERE model IN (
    SELECT model
    FROM products
    WHERE maker = 'A');
```

```
SELECT AVG(price) FROM pc_a_pricelist;
```

```
UPDATE pc_a_pricelist SET price = 1234 WHERE model = 1002;
```

Views: simplify complex queries

```
CREATE VIEW pc_prices      AS SELECT model, price FROM pcs;  
CREATE VIEW laptop_prices AS SELECT model, price FROM laptops;  
CREATE VIEW printer_prices AS SELECT model, price FROM printers;  
  
CREATE VIEW prices AS  
  (SELECT * FROM pc_prices) UNION  
  (SELECT * FROM laptop_prices) UNION  
  (SELECT * FROM printer_prices);
```

```
SELECT * FROM prices;
```

Constraints

Constraints

A **constraint** adds an *invariant* to your database. The DBMS enforces the invariant. Constraints ensure **data integrity**.

- **Keys:** PRIMARY KEY, UNIQUE KEY
- **Foreign keys:** FOREIGN KEY — ensure *referential integrity*
- **Value-** and **Tuple-based** constraints: NOT NULL, CHECK
 - Constrain values of a particular attribute, or relationship among multiple attributes in one table
- **Assertions:** any boolean expression, across tables ? ? ?

Revision: single-attribute keys

Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.

```
CREATE TABLE products (  
    maker CHAR(1),  
    model CHAR(4) PRIMARY KEY,  
    type VARCHAR(8) );
```

PRIMARY KEY implies NOT NULL for each key column.
UNIQUE doesn't care about NULLs.

Revision: single-attribute keys

Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.

```
CREATE TABLE customers (  
  customer_id CHAR(10) PRIMARY KEY,  
  firstname VARCHAR(32),  
  lastname VARCHAR(32),  
  city VARCHAR(32),  
  address VARCHAR(128),  
  email VARCHAR(128) UNIQUE);
```

Revision: multi-attribute keys

Let's assume that for each model X purchased by customer Y there is *at most* one entry per day in table sales:

```
CREATE TABLE sales (  
  customer_id CHAR(10),  
  model CHAR(4),  
  quantity INTEGER,  
  `day` DATE,  
  paid REAL,  
  type_of_payment VARCHAR(32),  
  PRIMARY KEY (customer_id, model, `day`));
```

Manipulating constraints

We can often manipulate constraints separately from the table — but you have to either *name* your constraints or know their auto-generated name.

```
CREATE TABLE x (i INT, j INT);  
ALTER TABLE x ADD CONSTRAINT ij_unique UNIQUE (i, j);  
INSERT INTO x VALUES (1, 2);  
INSERT INTO x VALUES (1, 2); -- bang!
```

Manipulating constraints

We can often manipulate constraints separately from the table — but you have to either *name* your constraints or know their auto-generated name.

```
ALTER TABLE x DROP CONSTRAINT ij_unique;
INSERT INTO x VALUES (1, 2); -- ok
SELECT * from x;             -- two identical rows!
INSERT INTO x VALUES (1, 2); -- still ok
SELECT * from x;             -- three identical rows!
```

Manipulating constraints

We can often manipulate constraints separately from the table — but you have to either *name* your constraints or know their auto-generated name.

```
ALTER TABLE x ADD CONSTRAINT ij_unique UNIQUE (i, j);  
ERROR 1062 (23000): Duplicate entry '1-2' for  
key 'x.ij_unique'
```

The DBMS *enforces the invariants you give it*. It won't allow you to add an invariant that doesn't hold.

Named vs unnamed key constraints

```
CREATE TABLE x (i INT, j INT, UNIQUE (i, j));
```

```
SHOW CREATE TABLE x;
```

```
CREATE TABLE `x` (  
  `i` int DEFAULT NULL,  
  `j` int DEFAULT NULL,  
  UNIQUE KEY `i` (`i`,`j`)  
) ENGINE=InnoDB  
  DEFAULT CHARSET=utf8mb4  
  COLLATE=utf8mb4_0900_ai_ci;
```

This is for MySQL. In SQLite, you can say `.schema tablename`, `.indexes tablename` instead.

Constraints are often implemented by the DBMS via an *index* for efficiently maintaining the invariant.

Named vs unnamed key constraints

```
CREATE TABLE x (i INT, j INT, UNIQUE foo (i, j));
```

```
SHOW CREATE TABLE x;
```

```
CREATE TABLE `x` (  
  `i` int DEFAULT NULL,  
  `j` int DEFAULT NULL,  
  UNIQUE KEY `foo` (`i`, `j`)  
) ENGINE=InnoDB  
  DEFAULT CHARSET=utf8mb4  
  COLLATE=utf8mb4_0900_ai_ci;
```

This is for MySQL. In SQLite, you can say `.schema tablename`, `.indexes tablename` instead.

Constraints are often implemented by the DBMS via an *index* for efficiently maintaining the invariant.

Foreign Keys

Invariant: values appearing in attributes of one relation must appear together in certain attributes of another.

Example: Consider Sales(customer_id, model, quantity, day, paid, type_of_payment).

- We might expect that a customer_id attribute also appears in Customers, and a model attribute in Products.
- We can *enforce* these relationships using FOREIGN KEYS.

Expressing Foreign Keys

Use keyword REFERENCES:

```
CREATE TABLE sales (  
  customer_id CHAR(10) REFERENCES customers (customer_id),  
  model CHAR(4) REFERENCES products (model),  
  ...);
```

```
CREATE TABLE sales (  
  customer_id CHAR(10),  
  ...,  
  FOREIGN KEY (customer_id) REFERENCES customers (customer_id),  
  FOREIGN KEY (model) REFERENCES products (model));
```

Expressing Foreign Keys

- Multi-column keys:

```
FOREIGN KEY (a1, a2, ...) REFERENCES t (a1, a2, ...)
```

- Referenced columns must be declared (as a group) PRIMARY KEY or UNIQUE.

- NULL is acceptable to a FOREIGN KEY constraint! 🤪

Consider declaring foreign keys NOT NULL:

```
CREATE TABLE sales (  
  customer_id CHAR(10) NOT NULL REFERENCES customers (customer_id),  
  ...);
```

(However, we will see a reason why allowing NULL might make sense shortly)

Important Note

Define foreign keys **before** inserting data in your tables!

Again: The DBMS *enforces the invariants you give it*. It won't allow you to add an invariant that doesn't hold.

```
ALTER TABLE sales ADD CONSTRAINT cust_id_fk  
    FOREIGN KEY (customer_id)  
    REFERENCES customers (customer_id);
```

```
ALTER TABLE sales DROP CONSTRAINT cust_id_fk;
```

Important SQLite-specific note

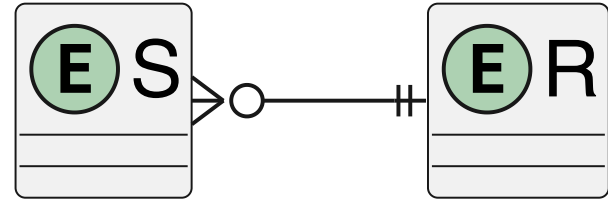
- Because of SQLite's history and intended usage, it *does not enforce* foreign key relationships by default.
- Furthermore, enforcement of FK constraints is done on a *per-connection basis*.

Turn on enforcement for the active connection (only!), each time you open a connection, using:

```
PRAGMA foreign_keys = ON;
```

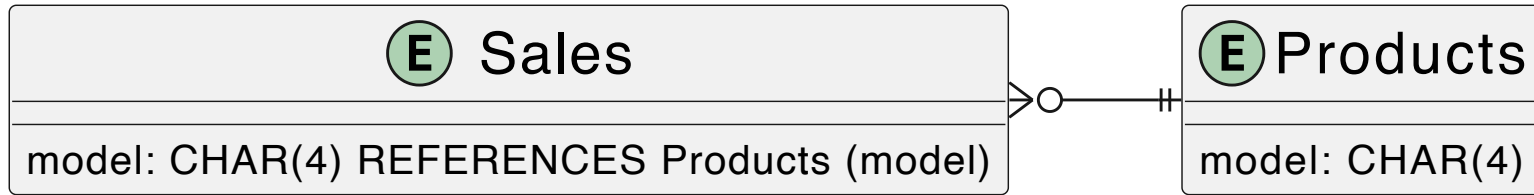
Maintaining FOREIGN KEY constraints

If there is a foreign-key constraint from relation S to relation R, two violations are possible:



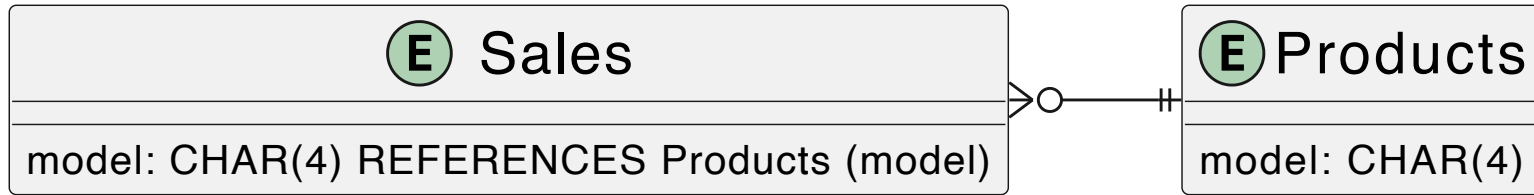
- An INSERT or UPDATE to S mentions values not found in R
- A DELETE or UPDATE to R causes some tuples of S to “dangle”

Maintaining FOREIGN KEY constraints



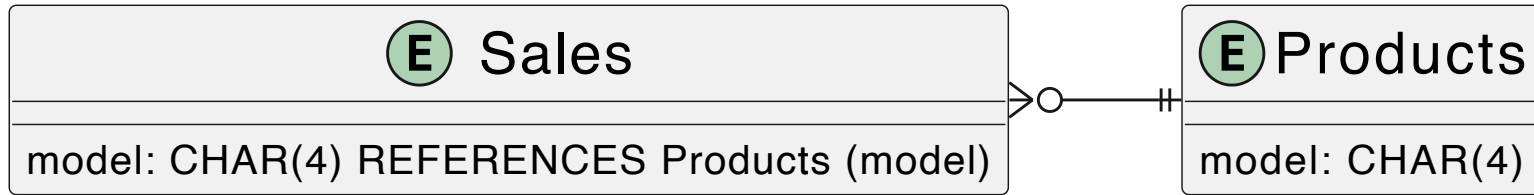
- An INSERT or UPDATE to Sales that mentions a nonexistent model...

Maintaining FOREIGN KEY constraints



- An INSERT or UPDATE to Sales that mentions a nonexistent model... must be rejected.

Maintaining FOREIGN KEY constraints



- A DELETE or UPDATE to Products that *removes* a model that is *in use* by some tuple(s) of Sales can be handled in multiple ways.
- Add **ON UPDATE ...**, **ON DELETE ...** clauses to the FOREIGN KEY declaration to choose one.

ON (UPDATE/DELETE): NO ACTION

```
ALTER TABLE sales ADD CONSTRAINT cust_id_fk  
  FOREIGN KEY (customer_id) REFERENCES customers (customer_id)  
  ON UPDATE NO ACTION  
  ON DELETE NO ACTION;
```

Rejects the problematic UPDATE or DELETE operation.

```
mysql> DELETE FROM customers;  
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key  
constraint fails (`pcshop`.`sales`, CONSTRAINT `cust_id_fk` FOREIGN KEY  
(`customer_id`) REFERENCES `customers` (`customer_id`))
```

DBMS-specific: RESTRICT is *almost* a synonym for NO ACTION. See your DBMS's docs!

ON (UPDATE/DELETE): Unspecified

```
ALTER TABLE sales ADD CONSTRAINT cust_id_fk  
  FOREIGN KEY (customer_id) REFERENCES customers (customer_id);  
  -- say nothing about ON UPDATE or ON DELETE
```

If you don't specify, you get NO ACTION — a safe default!*

* So long as you remembered to `PRAGMA foreign_keys = ON` for each and every connection, if you're using SQLite...

ON (UPDATE/DELETE): CASCADE

```
ALTER TABLE sales ADD CONSTRAINT cust_id_fk  
  FOREIGN KEY (customer_id) REFERENCES customers (customer_id)  
  ON UPDATE CASCADE  
  ON DELETE CASCADE;
```

Updates or deletes the dependent rows (of sales) when the target rows (of customers) are changed or removed.

```
mysql> DELETE FROM customers;  
Query OK, 5 rows affected (0.00 sec)  
  
mysql> SELECT * FROM sales;  
Empty set (0.00 sec)
```

ON (UPDATE/DELETE): SET NULL

Changes the referring columns in dependent rows (of sales) to NULL when the target rows (of customers) are changed or removed.

```
ALTER TABLE sales ADD CONSTRAINT cust_id_fk  
  FOREIGN KEY (customer_id) REFERENCES customers (customer_id)  
  ON UPDATE SET NULL  
  ON DELETE SET NULL;
```

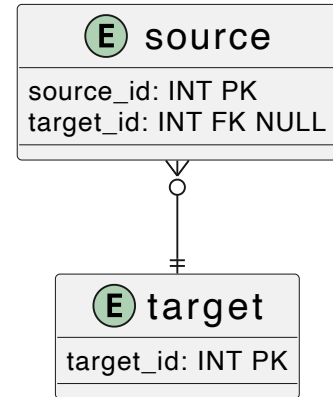
```
ERROR 1830 (HY000): Column 'customer_id' cannot be NOT NULL:  
needed in a foreign key constraint 'cust_id_fk' SET NULL
```

ON (UPDATE/DELETE): SET NULL

```
CREATE TABLE target (target_id INT PRIMARY KEY);
CREATE TABLE source (
  source_id INT PRIMARY KEY,
  target_id INT NULL REFERENCES target (target_id)
  ON UPDATE SET NULL ON DELETE SET NULL);

INSERT INTO target VALUES (123);
INSERT INTO source VALUES (808, 123);

DELETE FROM target;
SELECT * FROM source;
```



```
+-----+-----+
| source_id | target_id |
+-----+-----+
|      808 |      NULL |
+-----+-----+
1 row in set (0.00 sec)
```

Some DBMSs offer SET
DEFAULT as an additional
strategy.

Maintaining FOREIGN KEY constraints

What happens if we DROP TABLE something referred to by a FOREIGN KEY in another table?

Maintaining FOREIGN KEY constraints

What happens if we DROP TABLE something referred to by a FOREIGN KEY in another table?

```
mysql> DROP TABLE target;  
ERROR 3730 (HY000): Cannot drop table 'target' referenced  
by a foreign key constraint 'source_ibfk_1' on table  
'source'.
```

(It *could* have worked like a DELETE FROM target plus a DROP CONSTRAINT in source, but that isn't the semantics that was chosen...)

Choosing an UPDATE/DELETE Policy

- When we declare a foreign key, we may choose policies independently for deletions and updates.
- They do **not** have to be the same! It can be useful to configure them differently.
- If unspecified, a default (NO ACTION, i.e. reject) is used.

Read your DBMS's documentation *carefully*. Each DBMS differs! [MySQL docs](#), [SQLite docs](#)

Break

Value- and Tuple-based constraints

Constraints on value(s) of a column *or group of columns*.

- NULL, NOT NULL — permit or forbid NULLs in a column

```
CREATE TABLE professor (  
    ...,  
    coffee_cups INT NOT NULL, -- must be supplied; may not be NULL  
    current_task INT NULL REFERENCES tasks (task_id),  
    ...);
```

- CHECK(*condition*) — enforce arbitrary computed invariant for a column *or table*

CHECK for single columns

The condition may only use the name of the constrained column. Subqueries are not allowed in most DBMSs.

```
CREATE TABLE students (student_id INT PRIMARY KEY);  
CREATE TABLE courses (course_id CHAR(8) PRIMARY KEY);  
CREATE TABLE grades (  
    student_id INT NOT NULL REFERENCES students (student_id),  
    course_id CHAR(8) NOT NULL REFERENCES courses (course_id),  
    grade INT NOT NULL);
```

```
INSERT INTO grades VALUES (123456, 'BCS1510', 8000); -- !!!
```

CHECK for single columns

The condition may only use the name of the constrained column. Subqueries are not allowed in most DBMSs.

```
CREATE TABLE students (student_id INT PRIMARY KEY);  
CREATE TABLE courses (course_id CHAR(8) PRIMARY KEY);  
CREATE TABLE grades (  
    student_id INT NOT NULL REFERENCES students (student_id),  
    course_id CHAR(8) NOT NULL REFERENCES courses (course_id),  
    grade INT NOT NULL CHECK (grade >= 1 AND grade <= 10));
```

```
INSERT INTO grades VALUES (123456, 'BCS1510', 8000);  
ERROR 3819 (HY000): Check constraint 'grades_chk_1' is violated.
```

CHECK for multiple columns

The condition may use any columns in the table.

```
CREATE TABLE users (user_id INT PRIMARY KEY);
CREATE TABLE friends ( -- symmetric relationship
    friend1 INT NOT NULL REFERENCES users (user_id),
    friend2 INT NOT NULL REFERENCES users (user_id));
```

```
INSERT INTO users VALUES (123456);
INSERT INTO users VALUES (987654);
INSERT INTO friends VALUES (123456, 987654);
INSERT INTO friends VALUES (123456, 123456); -- ???
INSERT INTO friends VALUES (987654, 123456); -- !!!
```

CHECK for multiple columns

The condition may use any columns in the table.

```
CREATE TABLE users (user_id INT PRIMARY KEY);  
CREATE TABLE friends ( -- symmetric relationship  
    friend1 INT NOT NULL REFERENCES users (user_id),  
    friend2 INT NOT NULL REFERENCES users (user_id),  
    CHECK (friend1 < friend2));
```

```
INSERT INTO friends VALUES (987654, 123456);  
ERROR 3819 (HY000): Check constraint 'friends_chk_1' is violated.
```

Timing of CHECKs

CHECK expressions are evaluated only when the column or table concerned is INSERTed to or UPDATED.

```
CREATE TABLE sales (  
  customer_id CHAR(10) NOT NULL REFERENCES customers (customer_id),  
  model CHAR(4) NOT NULL CHECK (model IN (SELECT model FROM products)),  
  `day` DATE NOT NULL CHECK (`day` > '2025-01-01'),  
  quantity INT NOT NULL,  
  paid DOUBLE NOT NULL,  
  PRIMARY KEY (customer_id, model, `day`));
```

Delete a model from products → sales loses referential integrity. (Most DBMSs will not allow subqueries in CHECK!)

SQL-92 CREATE ASSERTION ? ? ?

The SQL-92 standard includes

```
CREATE ASSERTION assertion_name CHECK (expr);
```

The idea is to have a *schema-wide* CHECK, able to enforce arbitrary consistency conditions across multiple tables.

However, **hardly any DBMSs implement it!** So if we *need* something like this, we have to reach for the big guns:

Triggers

Triggers

A **trigger** is a kind of *stored procedure*, called when something *changes* in the database.

```
CREATE TRIGGER trigger_name
  trigger_condition  -- [BEFORE | AFTER] [UPDATE | INSERT | DELETE]
  ON table_name
  FOR EACH ROW expr;  -- this can be BEGIN ... END
```

The trigger condition can be BEFORE UPDATE, AFTER INSERT, BEFORE DELETE, etc. etc.

Triggers

You can use triggers to enforce invariants that constraints, CHECKs and ASSERTIONS cannot express.

However, they are *general purpose*: you can put any kind of code in a trigger body!

This freedom, and their **imperative nature**, makes them difficult to use correctly.

They are powerful and dangerous.

Triggers instead of subqueries in CHECK

```
... CHECK (model IN (SELECT model FROM products)) ...
```

× Subqueries not supported in most DBMSs

... so what can we do? Setting aside that this kind of "check" is a bad idea

Triggers instead of subqueries in CHECK

```
-- 🤔 This is needed because the semicolon terminates input
-- of the statement mid-way through the IF! So we set it to
-- something else temporarily to allow multiple semicolons...
DELIMITER //
```

```
CREATE TRIGGER enforce_sales_model BEFORE INSERT ON sales
  FOR EACH ROW BEGIN
    IF NEW.model NOT IN (SELECT model FROM products) THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'invalid sales.model';
    END IF;
  END;
```

```
DELIMITER ;
-- ^ Don't forget to set it back!
```

Triggers instead of subqueries in CHECK

- ✓ Great, that handles INSERTs!
- 🤔 Now, go duplicate the code for BEFORE UPDATE
 - Or use CREATE PROCEDURE and call it from both the INSERT and UPDATE triggers
- MySQL uses SIGNAL SQLSTATE '45000' to raise an exception; SQLite has RAISE; there is no standard here 😞

Aside: in an UPDATE trigger you get to use both OLD and NEW; in INSERT triggers, only NEW; and in DELETE triggers, only OLD.

What other ~~bad~~ ideas can we implement with triggers?

```
CREATE TABLE sales_history (all_sales BIGINT NOT NULL,  
                             latest_sale DATE NOT NULL);  
INSERT INTO sales_history VALUES (0, NOW());  
  
DELIMITER //  
  
CREATE TRIGGER update_sales_history AFTER INSERT on sales  
FOR EACH ROW BEGIN  
    UPDATE sales_history SET all_sales = all_sales + 1,  
                           latest_sale = NOW();  
  
END;  
  
DELIMITER ;
```

But why not just...

```
SELECT COUNT(*), MAX(`day`) FROM sales;
```

???

But why not just...

```
SELECT COUNT(*), MAX(`day`) FROM sales;
```

???

What if someone executes `DELETE FROM sales` ?

What other ~~bad~~ ideas can we implement with triggers?

```
DELIMITER //
```

```
CREATE TRIGGER add_model_if_absent BEFORE INSERT on sales
  FOR EACH ROW BEGIN
    IF NEW.model NOT IN (SELECT model FROM products) THEN
      INSERT INTO products VALUES (NULL, NEW.model, NULL);
    END IF;
  END;
```

```
DELIMITER ;
```

```
INSERT INTO sales VALUES ('1231231231', '1999', 1, '2025-04-02', 100, 'cash');
SELECT * FROM products;
```

But why not just...

make the programmer do the right thing and insert necessary rows beforehand

???


But why not just...

make the programmer do the right thing and insert necessary rows beforehand

???



Triggers: Gotchas

- Cascaded foreign key actions in MySQL do not activate triggers. They do in some other DBMSs.
- Multiple triggers can exist for the same event on the same table. You have to order them carefully sometimes. Read your DBMS documentation!
- Some DBMSs (Oracle , SQL Server) support trigger events for DDL as well as DML. Some DBMSs support “statement-level triggers” as well as the FOR EACH ROW triggers we've seen.
- What if there's an exception in some trigger? Usually you get something like a rollback, but each DBMS is subtly different.

More important SQL functions

COALESCE

Takes **any number of arguments**, returns the **first non-NULL**.

Useful where zero or more related rows could be present:

```
CREATE TABLE posts (post_id INT PRIMARY KEY);
CREATE TABLE ratings (
  post_id INT NOT NULL REFERENCES posts (post_id),
  rating INT CHECK (rating >= 1 AND rating <= 5));
INSERT INTO posts VALUES (1234);
INSERT INTO posts VALUES (5678);
INSERT INTO ratings VALUES (1234, 4);
INSERT INTO ratings VALUES (1234, 5);
```

COALESCE

```
SELECT p.post_id,  
       (SELECT AVG(r.rating)  
        FROM ratings r  
        WHERE r.post_id = p.post_id)  
       AS avg_rating  
FROM posts p;
```

post_id	avg_rating
1234	4.5000
5678	NULL

COALESCE

```
SELECT p.post_id,  
       (SELECT COALESCE(AVG(r.rating), 0)  
        FROM ratings r  
        WHERE r.post_id = p.post_id)  
       AS avg_rating  
FROM posts p;
```

post_id	avg_rating
1234	4.5000
5678	0.0000

GREATEST and LEAST

```
mysql> SELECT MAX(1, 103, 44);  
ERROR 1064 (42000): You have an error in your SQL syntax;  
check the manual that corresponds to your MySQL server version  
for the right syntax to use near ', 103, 44)' at line 1
```

```
mysql> SELECT GREATEST(1, 103, 44);  
+-----+  
| GREATEST(1, 103, 44) |  
+-----+  
|                    103 |  
+-----+  
1 row in set (0.00 sec)
```

GREATEST and LEAST

```
mysql> SELECT MIN(1, 103, 44);  
ERROR 1064 (42000): You have an error in your SQL syntax;  
check the manual that corresponds to your MySQL server version  
for the right syntax to use near ', 103, 44)' at line 1
```

```
mysql> SELECT LEAST(1, 103, 44);  
+-----+  
| LEAST(1, 103, 44) |  
+-----+  
|                  1 |  
+-----+  
1 row in set (0.00 sec)
```

See you in lab!