>>

SQL: Updating the Data

- Data Modification in SQL
- Insertion
- Bulk Insertion of Data
- Deletion
- Semantics of Deletion
- Updates

^ >>

❖ Data Modification in SQL

We have seen statements to modify table meta-data (in DB catalog):

- **CREATE TABLE** ... add new, initially empty, table to DB
- **DROP TABLE** ... remove table data (all tuples) and meta-data
- **ALTER TABLE** ... change meta-data of table (e.g add constraints)

SQL also provides statements for modifying data in tables:

- **INSERT** ... add a new tuple(s) into a table
- **DELETE** ... remove tuples from a table (via condition)
- **UPDATE** ... modify values in existing tuples (via condition)

Constraint checking is applied automatically on any change.

i.e. not null will block nulls regardless of if they're supplied on tuple insertion or update.

Operation fails (no change to DB) if any constraint check fails

i.e. there is rollback. A operation either succeeds fully, or it fails with no side-effects. This is obviously important behaviour for ensuring integrity of data.

<< ^ >>

❖ Insertion

Add new tuples via the **INSERT** operation:

```
INSERT INTO RelationName
VALUES (val1, val2, val3, ...)
```

not specifying the columns (you have to specify all values for all columns in order if you do this though)

```
INSERT INTO RelationName(Attr1, Attr2, ...)
VALUES (valForAttr1, valForAttr2, ...)
```

explicitly specifying which columns these values are for

```
INSERT INTO RelationName
VALUES Tuple1, Tuple2, Tuple3, ...
```

The first two add a single new tuple into *RelationName*.

The last form adds multiple tuples into *RelationName*.

<< ^ >>

❖ Insertion (cont)

INSERT INTO *R* **VALUES** (v_1, v_2, ...)

- values must be supplied for all attributes of *R*
- in same order as appear in **CREATE TABLE** statement
- special value **DEFAULT** forces default value or **NULL**

i believe this default here is saying: if this column has a default use that, else use null (obviously if there is a not null constraint the insertion will fail).

INSERT INTO *R*(A_1, A_2, ...) **VALUES** (v_1, v_2, ...)

- can specify any subset of attributes of *R*
- values must match attribute specification order
- unspecified attributes are assigned default or null

this is the common sense approach

Note that the default talked about above is not the same as the default constraint on the column definition of a table. The default talked about here is a keyword that you supply when inserting into a table e.g. INSERT INTO invitation(to, from, status) values ('Nick', 'Jen', DEFAULT)
and the default keyword here will mean the status field will get a default value or null (depending on whether the column definition has a default value or not e.g. status::invited)

<< ^ >>

❖ Insertion (cont)

Example: Add the fact that Justin likes 'Old'.

```
INSERT INTO Likes VALUES ('Justin','Old');
-- or --
INSERT INTO Likes(drinker,beer) VALUES('Justin','Old');
-- or --
INSERT INTO Likes(beer,drinker) VALUES('Old','Justin');
```

note that when you specify attributes you can specify them in whichever order you like (although it probably makes most sense to insert them in the same order as the schema specifies)

Example: Add a new beer with unknown style.

```
INSERT INTO Beers(name,brewer)
       VALUES('Mysterio','Hop Nation');
-- which inserts the tuple ...
('Mysterio', 'Hop Nation', null)
```

Note that if there as a not null constraint on the style then this insertion wouldn't work (unless there was also a default value on the column)

❖ Insertion (cont)

Example: insertion with default values

```
ALTER TABLE Likes
   ALTER COLUMN beer SET DEFAULT 'New';
ALTER TABLE Likes
   ALTER COLUMN drinker SET DEFAULT 'Joe';

INSERT INTO Likes(drinker) VALUES('Fred');
INSERT INTO Likes(beer) VALUES('Sparkling Ale');

-- inserts the two new tuples ...
('Fred', 'New')
('Joe', 'Sparkling Ale')
```

new is the default used in the first insertion where no beer is supplied
joe is the default used in the second insertion where no drinker is supplied

<< ^ >>

❖ Insertion (cont)

Example: insertion with insufficient values.

E.g. specify that drinkers' phone numbers cannot be **NULL**.

```
ALTER TABLE Drinkers
   ALTER COLUMN phone SET NOT NULL;
```

Then try to insert a drinker whose phone number we don't know:

```
INSERT INTO Drinkers(name,addr) VALUES ('Zoe','Manly');

ERROR:  null value in column "phone" violates
                                not-null constraint
DETAIL:  Failing row contains (Zoe, Manly, null).
```

<< ^ >>

❖ Bulk Insertion of Data

Tuples may be inserted individually:

```
insert into Stuff(x,y,s) values (2,4,'green');
insert into Stuff(x,y,s) values (4,8,null);
insert into Stuff(x,y,s) values (8,null,'red');
...
```

but this is tedious if 1000's of tuples are involved.

It is also inefficient

- all relevant constraints are checked on insertion of each tuple

So, most DBMSs provide non-SQL methods for bulk insertion

❖ Bulk Insertion of Data (cont)

Bulk insertion methods typically ...

- use a compact representation for each tuple
- "load" all tuples without constraint checking
- do all constraint checks at the end
- if any tuples fail checks, none are inserted

Example: PostgreSQL's **copy** statement:

```
COPY Stuff(x,y,s) FROM stdin;
2       4       green
4       8       \N
8       \N      red
\.
```

Can also copy from a named file (but must be readable by PostrgeSQL server)

<< ^ >>

❖ Deletion

Removing tuples is accomplished via **DELETE** statement:

```
DELETE FROM Relation
WHERE  Condition
```

Removes all tuples from *Relation* that satisfy *Condition*.

Example: Justin no longer likes Sparkling Ale.

```
DELETE FROM Likes
WHERE drinker = 'Justin'
      AND beer = 'Sparkling Ale';
```

Special case: Make relation *R* empty.

```
DELETE FROM R;
```

or

```
DELETE FROM R WHERE true;
```

these are delete all statements

<< ^ >>

❖ Deletion (cont)

Example: remove all expensive beers from sale.

```
DELETE FROM Sells WHERE price >= 5.00;
```

Example: remove all beers with unknown style

```
DELETE FROM Beers WHERE style IS NULL;
```

This fails* if such Beers are referenced from other tables

E.g. such Beers are liked by someone or sold in some bar

* no beers are removed, even if some are not referenced

i.e. if 1 fails, the whole operation fails, this seems to be a pattern (either everything succeeds, or everything is rolled back and there are no side-effects).

<< ^ >>

❖ Semantics of Deletion

Method A for **DELETE FROM** *R* **WHERE** *Cond*:

```
FOR EACH tuple T in R DO
    IF T satisfies Cond THEN
        remove T from relation R
    END
END
```

this changes the database and hence may change the amount of other tuples that are deleted

Method B for **DELETE FROM** *R* **WHERE** *Cond*:

```
FOR EACH tuple T in R DO
    IF T satisfies Cond THEN
        make a note of this T
    END
END
FOR EACH noted tuple T DO
    remove T from relation R
END
```

because we only make a note, the actual database contents remain unchanged and so we are guaranteed that we will always delete the same amount.

<< ^ >>

❖ Semantics of Deletion (cont)

Does it matter which method the DBMS uses?

For most cases, the same tuples would be deleted

But if *Cond* involes a query on the table *R*

- the result of *Cond* might change as the deletion progresses
- so Method A might delete less tuples than Method B

E.g.

```
DELETE FROM Beers
WHERE (SELECT count(*) FROM Beers) > 10;
```

Method A deletes beers until there are only 10 left
Method B deletes all beers if there were more than 10 to start with

A evaluates the condition, deletes if true, then evaluates again, and continues until false
B evaluates the condition, marks for deletion, then evaluates again, and so on, then deletes at the very end. Hence it either deletes all or none.

<< ^ >>

❖ Updates

The **UPDATE** statement allows you to ...

- modify values of specified attributes in specified tuples of a relation

```
UPDATE R
SET    List of assignments
WHERE  Condition
```

Each tuple in relation *R* that satisfies *Condition* is affected

Assignments may:

- assign constant values to attributes,
  e.g. **`SET price = 2.00`**
- use existing values in the tuple to compute new values,
  e.g. **`SET price = price * 0.5`**

❖ Updates (cont)

Example: Adam changes his phone number.

```
UPDATE Drinkers
SET    phone = '9385-2222'
WHERE  name = 'Adam';
```

Example: John moves to Coogee.

```
UPDATE Drinkers
SET    addr = 'Coogee',
       phone = '9665-4321'
WHERE  name = 'John';
```

<< ^

❖ Updates (cont)

Examples that modify many tuples ...

Example: Make $6 the maximum price for beer.

```
UPDATE Sells
SET    price = 6.00
WHERE  price > 6.00;
```

any tuple

Example: Increase beer prices by 10%.

```
UPDATE Sells
SET    price = price * 1.10;
```

Updates all tuples (as if **`WHERE true`**)

i.e. leaving out the where statement, or including where true; means all tuples will be updated.

Produced: 28 Sep 2020