

A Short Introduction to Relational Databases

Part 2

Riccardo Murri <`riccardo.murri@uzh.ch`>

Creating tables (SQL DDL)

Table definition (1/2)

Informally defined:

A table definition is the list of column names and their associated data types.

(For a more formal treatment, see pt. 1 of this talk.)

Table definition (2/2)

In SQL, we write a table definition as the table name, followed by the list of column names (and types) within parentheses:

```
Movie(name VARCHAR(64), released DATE,  
      length INTEGER,    filmType CHAR(5))
```

Data types (1/4)

ANSI SQL includes the following string data types.

Character strings

<code>CHARACTER (n)</code> or <code>CHAR (n)</code>	fixed-width n -character string, padded with spaces
---	---

<code>CHARACTER VARYING (n)</code> or <code>VARCHAR (n)</code>	variable-width string with a maximum size of n characters
---	---

Bit strings

<code>BIT (n)</code>	an array of exactly n bits
----------------------	------------------------------

<code>BIT VARYING (n)</code>	an array of up to n bits
------------------------------	----------------------------

Data types (2/4)

ANSI SQL includes the following numeric data types.

Numbers

INTEGER, SMALLINT, and BIGINT fixed-size integer numbers

FLOAT, REAL and DOUBLE PRECISION floating-point numbers

NUMERIC(prec, scale) or DECIMAL(prec, scale) fixed-precision numbers

The precision is a positive integer that determines the total number of digits. The scale is a non-negative integer, indicating the number of digits after the decimal point/comma.

For example, the number 123.45 has a precision of 5 and a scale of 2.

Data types (3/4)

ANSI SQL includes the following temporal data types.

Time/Date

DATE for date values (e.g. DATE '2011-05-03')

TIME for time values (e.g. TIME '15:51:36')

TIMESTAMP DATE and TIME combined
(e.g. TIMESTAMP '2011-05-03 15:51:36')

The way to enter date/time *constants* is by using a string prefixed with the word TIME/DATE/TIMESTAMP.

The current system date / time of the database server can be called by using functions CURRENT_DATE /
CURRENT_TIME / CURRENT_TIMESTAMP

Data types (4/4)

ANSI SQL includes the following timezone-aware temporal data types.

Time/Date

TIME WITH TIME ZONE or TIMETZ	same as TIME, but including time-zone offset
----------------------------------	---

TIMESTAMP WITH TIME ZONE or TIMESTAMPTZ	same as TIMESTAMP, but including time-zone offset
--	--

SQL provides ways of generating a date/time variable out of a date/time string, as well as for extracting the respective members (seconds, for instance) of such variables. See:

https://en.wikibooks.org/wiki/SQL_Dialects_Reference/Functions_and_expressions/Date_and_time_functions

CREATE TABLE

SQL statement `CREATE TABLE` is used to create a table in the current database:

```
CREATE TABLE Movie(name VARCHAR(64), released DATE,  
length INTEGER, filmType CHAR(5));
```

Right after creation, the table contains no rows.

CREATE TABLE (2/2)

There is no *portable* way of reading back a table definition after it has been created!

For example, to view the schema of table *tablename* you would have to enter (depending on the RDBMS):

<i>RDBMS</i>	<i>command</i>
PostgreSQL	<code>\d tablename</code>
MySQL / MariaDB	<code>DESCRIBE tablename;</code>
SQLite	<code>.schema tablename</code>

CREATE TEMPORARY TABLE

SQL statement `CREATE TEMPORARY TABLE` has exactly the same syntax as `CREATE TABLE`, but temporary tables are automatically deleted at the end of the session.

CREATE TEMPORARY TABLE

```
Movie(name VARCHAR(64), released DATE,  
      length INTEGER,    filmType CHAR(5));
```

DROP TABLE

To *permanently* delete a table, use DROP TABLE:

```
DROP TABLE movie;
```

An additional clause IF EXISTS makes the statement succeed even if the table had already been deleted:

```
DROP TABLE IF EXISTS movie;
```

Inserting and updating data (SQL DML)

INSERT

The INSERT statement can be used to insert a row of values into a table.

```
INSERT INTO Movie(title, released, length, filmType)
VALUES ('Star_Wars', DATE '1977-05-25', 124, 'color');
```

Note:

- ▶ the INSERT fails if the same row (or a row agreeing on the *key* columns) is already present in the table.

Reference: For more details, see: [https://en.wikipedia.org/wiki/Insert_\(SQL\)](https://en.wikipedia.org/wiki/Insert_(SQL))

INSERT INTO ... SELECT

A variant of `INSERT` allows inserting into a table the results of a query on another table:

```
INSERT INTO lustre_u116(grp, size, path)
SELECT grp, size, path FROM lustre
WHERE usr='usr116';
```

Note that:

- ▶ The schema of the table being `INSERT`'ed to must match the (sub)schema of columns in the `SELECT` query.
- ▶ There are no parentheses around the `SELECT` sub-query.
- ▶ No duplicate rows must be returned by the sub-query.

UPDATE

The UPDATE statement allows changing one or more fields in *existing* rows.

```
-- rename user
UPDATE lustre SET usr='rmurri' WHERE usr='usr999';

-- alter all values in a column
UPDATE lustre SET blksize=(4096*blksize);
```

Note:

- ▶ The WHERE clause has the same syntax and semantics as its equivalent in the SELECT statement.
- ▶ If you omit the WHERE clause, **all rows will be updated!**

Reference: For more details, see:

[https://en.wikipedia.org/wiki/Update_\(SQL\)](https://en.wikipedia.org/wiki/Update_(SQL))

MERGE (1/3)

What if you want to
change some fields in a row,
or
add the entire row
if it's not already there?

MERGE (2/3)

SQL:2003 introduced the MERGE statement:

```
MERGE INTO target USING source ON (columns)  
  WHEN MATCHED THEN  
    UPDATE SET column1 = value1 [, column2 = value2 ...]  
  WHEN NOT MATCHED THEN  
    INSERT (column1 [, column2 ...])  
      VALUES (value1 [, value2 ...]);
```

A right outer join is employed over *target* and *source*, then:

- ▶ If the ON field(s) in *source* matches the ON field(s) in *target*, then UPDATE
- ▶ If the ON field(s) in *source* does not match the ON field(s) in *target*, then INSERT
- ▶ If the ON field(s) does not exist in *source* but does exist in *target*, then no action is performed.
- ▶ If the ON field(s) does not exist in either *source* or *target*, then no action is performed.
- ▶ If multiple *source* rows match a given *target* row, an error is mandated by SQL:2003 standards.

MERGE (3/3)

Note: As of today, none of the popular open-source RDBMS'es supports the `MERGE` statement.

MySQL/MariaDB supports `REPLACE INTO`: first attempt an insert, and if that fails, delete the row (if exists) and then insert the new one.

SQLite's `INSERT OR REPLACE INTO` works similarly.

PostgreSQL supports merging via `INSERT INTO ... ON CONFLICT [conflict_target] conflict_action`.

Reference: For more details, see:

[https://en.wikipedia.org/wiki/Merge_\(SQL\)](https://en.wikipedia.org/wiki/Merge_(SQL))

Joins

Joins

From Wikipedia:

A SQL JOIN clause combines columns from one or more tables in a relational database [...] by using values common to each.

ANSI-standard SQL specifies five types of JOIN: INNER, LEFT OUTER, RIGHT OUTER, FULL OUTER and CROSS.

Reference: For more details, see: [https://en.wikipedia.org/wiki/Join_\(SQL\)](https://en.wikipedia.org/wiki/Join_(SQL))

CROSS JOIN

The `CROSS JOIN` operator produces the *Cartesian product* of two relations.

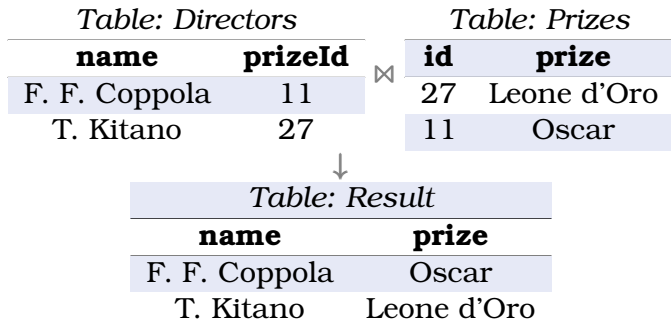
t1		t2		t1	t2
1	×	a	→	1	a
2		b		1	b
				2	a
				2	b

```
SELECT * FROM t1 CROSS JOIN t2;
```

What if two tables have
the same column names?

INNER JOIN

An `INNER JOIN` returns the set of tuples from the cross product of two tables that satisfy a certain predicate:



```
SELECT name, prize
FROM directors INNER JOIN prizes
ON directors.prizeId = prizes.id ;
```


INNER JOIN

An `INNER JOIN` returns the set of tuples from the cross product of two tables that satisfy a certain predicate.

```
SELECT name, prize  
FROM directors INNER JOIN prize  
ON directors.prizeId = prizes.id ;
```

This is the selection predicate.

It can contain any valid boolean expression, not just equality comparisons.

INNER JOIN

An `INNER JOIN` returns the set of tuples from the cross product of two tables that satisfy a certain predicate:

```
SELECT name, prize
FROM directors INNER JOIN prizes
ON directors.prizeId = prizes.id ;
```

The two *relations* to join.

They are referred to as *left* and *right* table.
(Need not be tables: views or subqueries will do as well.)

INNER JOIN

An `INNER JOIN` returns the set of tuples from the cross product of two tables that satisfy a certain predicate:

```
SELECT name, prize  
FROM directors INNER JOIN prizes  
ON directors.prizeId = prizes.id ;
```

As in any `SELECT` statement, you can then project on a subset of the columns, in particular you can *exclude* the joined columns from the output.

Implicit INNER JOIN

These two SELECT statements are equivalent:

```
-- explicit INNER JOIN
SELECT name, prize
FROM directors INNER JOIN prizes
ON directors.prizeId = prizes.id ;

-- implicit
SELECT directors.name, prizes.prize
FROM directors, prizes
WHERE directors.prizeId = prizes.id ;
```

(Omitting the WHERE clause in the implicit syntax results in a cross join.)

Self-joins

Self-joins are joins of a relation with (a copy of) itself.

For example: *find all pairs of film directors who have won the same prize.*

How would you write such a query?

Self-joins

Self-joins are joins of a relation with (a copy of) itself.

For example: *find all pairs of film directors who have won the same prize.*

Use table aliases!

```
SELECT d1.name, d2.name
FROM directors AS d1, directors AS d2
WHERE d1.prizeId = d2.prizeId;
```

Equi-joins and natural joins

A join operation is called an *equi-join* if the selection predicate is a conjunction of equality comparisons.

The `NATURAL JOIN` is an equi-join where the *implied* selection predicate specifies equality on all common attribute names.

(The same natural join query may return different results if table schemas have been altered. Avoid them: “explicit is better than implicit”.)

NULLs, again

Recall that the special SQL value `NULL` makes any arithmetic or boolean expression undefined.

(In particular, `NULL` does not even compare equal to itself.)

Therefore: **INNER JOINS skip any row that contains a NULL value** on either side. (Unless you specifically catered for `NULL`s in the predicate.)

OUTER JOIN

In a OUTER JOIN, the result table retains each row, even if no other matching row exists.

- ▶ In a LEFT OUTER JOIN, every row from the **left** table is retained: the result is padded with NULL if no matching row from the *right* table is found.
- ▶ A RIGHT OUTER JOIN does the same with *left* and **right** reversed.
- ▶ In a FULL OUTER JOIN, rows from both sides are retained.

LEFT OUTER JOIN

Table: Directors

name	prizeId
F. F. Coppola	11
T. Kitano	27
Ed Wood	NULL

Table: Prizes

id	prize
27	Leone d'Oro
11	Oscar



Table: Result

name	prize
F. F. Coppola	Oscar
T. Kitano	Leone d'Oro
Ed Wood	NULL

```
SELECT name, prize
FROM directors LEFT JOIN prizes
ON directors.prizeId = prizes.id ;
```

Why are joins important?

Because Normal forms!

Why normal forms?

Suppose we want to keep a database of courses being taught at the University, and data about their instructors (e.g., room, office hours).

A pretty natural choice seems to have one row per course with this schema:

`Courses (code, instructor, room, office_hours)`

But many instructors teach more than one course!
How do we solve that?

Try repeating rows: each course has a row.

*Unfortunately, this design suffers from insert, update, and deletion **anomalies**.*

Why normal forms?

Suppose we want to keep a database of courses being taught at the University, and data about their instructors (e.g., room, office hours).

A pretty natural choice seems to have one row per course with this schema:

```
Courses (code, instructor, room, office_hours)
```

But many instructors teach more than one course!
How do we solve that?

Try repeating rows: each course has a row.

*Unfortunately, this design suffers from insert, update, and deletion **anomalies**.*

Why normal forms?

Suppose we want to keep a database of courses being taught at the University, and data about their instructors (e.g., room, office hours).

A pretty natural choice seems to have one row per course with this schema:

```
Courses (code, instructor, room, office_hours)
```

But many instructors teach more than one course!
How do we solve that?

Try repeating rows: each course has a row.

*Unfortunately, this design suffers from insert, update, and deletion **anomalies**.*

Why normal forms?

Suppose we want to keep a database of courses being taught at the University, and data about their instructors (e.g., room, office hours).

A pretty natural choice seems to have one row per course with this schema:

```
Courses (code, instructor, room, office_hours)
```

But many instructors teach more than one course!
How do we solve that?

Try repeating rows: each course has a row.

*Unfortunately, this design suffers from insert, update, and deletion **anomalies**.*

Update anomalies

Employees' Skills

Employee ID	Employee Address	Skill
426	87 Sycamore Grove	Typing
426	87 Sycamore Grove	Shorthand
519	94 Chestnut Street	Public Speaking
519	96 Walnut Avenue	Carpentry

An **update anomaly**.
Employee 519 is shown as
having different addresses on different records.

Image & text credit: https://en.wikipedia.org/wiki/Database_normalization

Insertion anomalies

Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201

424	Dr. Newsome	29-Mar-2007	?
-----	-------------	-------------	---

An **insertion anomaly**.
Until the new faculty member, Dr. Newsome,
is assigned to teach at least one course,
his details cannot be recorded.

Image & text credit: https://en.wikipedia.org/wiki/Database_normalization

Deletion anomalies

Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201



DELETE

A deletion anomaly.

All information about Dr. Giddens is lost if he temporarily ceases to be assigned to any courses.

Image & text credit: https://en.wikipedia.org/wiki/Database_normalization

Normalization

Informally, problems arise when a set of data has 1-to-many or many-to-many interdependencies.

Normalization is a set of rules and procedures to properly design DB tables to minimize (or avoid) anomalies.

Keys and functional dependencies play a fundamental role in normalization.

Primary Keys

Recall that a *superkey* for a table T is a set K of columns such that no two rows of T take the same values when projected over K .

(Equivalently: a superkey is a set of attributes within a relation whose values can be used to uniquely identify a tuple.)

A *candidate key* is a minimal such set.

A *primary key* is a chosen candidate key.

Primary Keys in SQL

It is possible to designate one column or a subset of columns as `PRIMARY KEY` in a `CREATE TABLE` statement.

The RDBMS enforces the following constraints for a `PRIMARY KEY`:

- ▶ `UNIQUE`: projection on the set of keys cannot yield duplicate rows.
- ▶ `NOT NULL`: primary keys cannot take on the special value `NULL`.

(It is however possible to designate a column as `UNIQUE` and/or `NOT NULL` without it being a primary key.)

Foreign keys

A *foreign key* in SQL is a constraint: a column is allowed to only take values that are present in a specified column of another table.

A column that is referenced as a foreign key by another column must have been defined with a `UNIQUE` constraint (e.g., a primary key).

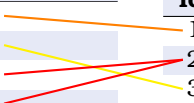
It is possible to designate foreign key constraints in a `CREATE TABLE` statement.

See also: <http://www.1keydata.com/sql/sql-foreign-key.html>

How to represent 1-to-many relationships

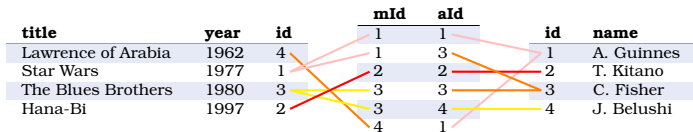
For instance, let us consider the relation “has directed” between film directors and movies: a director typically has supervised many movies, but a movie only credits a single director.

title	year	directorId	id	name
Star Wars	1977	1	1	G. Lucas
Pulp Fiction	1992	3	2	T. Kitano
Kids return	1996	2	3	Q. Tarantino
Hana-Bi	1997	2		



How to represent *many-to-many* relationships

For instance, let us consider the relation “starred in” between actors and movies: actors typically star in many movies, and there is more than one actor starring in a given movie.



Normal forms

These ways of representing $1 \rightarrow N$ and $N \rightarrow M$ relationships generally produce relations in *third normal form* (3NF).

1NF

- ▶ Column values are *atomic*
- ▶ No repeated columns (or groups thereof)
- ▶ Each table has a primary key

2NF

Each column must depend on the *entire* primary key

3NF

Each column must depend *directly* on the primary key.

See also:

<http://www.troubleshooters.com/littstip/1tnorm.html>

Appendix

References

Pagh, R.: “Introduction to Database Systems”,
<http://www.itu.dk/people/pagh/IDB05/>

Codd, E. F.: “A relational model for large shared data banks”, *Comm. ACM* 13:6, pp. 377–387, 1970