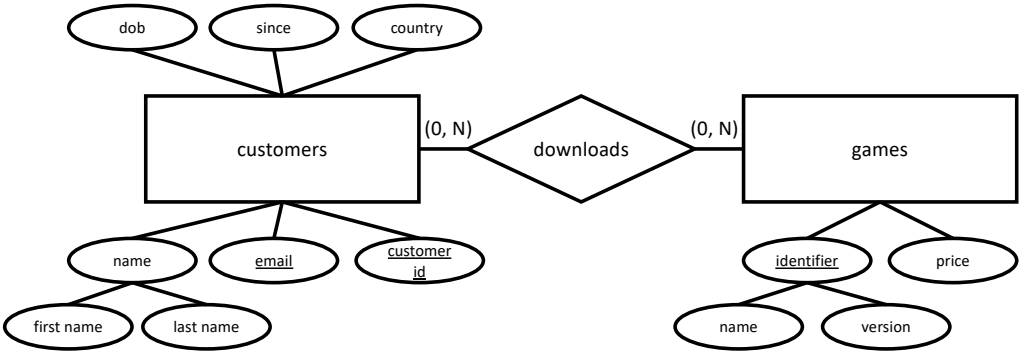
Simple Queries in SQL

Stéphane Bressan





We want to develop an application for managing the data of our online app store. We would like to store several items of information about our customers such as their first name, last name, date of birth, e-mail, date and country of registration to our online sales service and the customer identifier that they have chosen . We also want to manage the list of our products, games, their name, their version and their price. The price is fixed for each version of each game. Finally, our customers buy and download games. So we must remember which version of which game each customer has downloaded. It is not important to keep the download date for this application.


dob
since
country
customers
(0, N)
downloads
(0, N)
games
name
email
customer id
first name
last name
identifier
price
name
version

We can use PostgreSQL interactive terminal `psql`.

```
% psql -h localhost -U postgres
Password for user postgres:
psql (9.6.3, server 9.6.4)
Type "help" for help.
postgres=# CREATE DATABASE cs2102;
CREATE DATABASE
postgres=# \c dedomenology;
psql (9.6.3, server 9.6.4)
You are now connected to database "cs2102" as user "postgres".
postgres=# \i AppStoreSchema.sql
CREATE TABLE
CREATE TABLE
CREATE TABLE
postgres=#  \i AppStoreCustomers.sql
INSERT 0 1
...
postgres=# \i AppStoreGames.sql
INSERT 0 1
...
postgres=# \i AppStoreDownloads.sql
INSERT 0 1
...
postgres=# \q
```

This is the complete schema for our example

```
1  CREATE TABLE IF NOT EXISTS customers (
2   first_name VARCHAR(64) NOT NULL,
3   last_name VARCHAR(64) NOT NULL,
4   email VARCHAR(64) UNIQUE NOT NULL,
5   dob DATE NOT NULL,
6   since DATE NOT NULL,
7   customerid VARCHAR(16) PRIMARY KEY,
8   country VARCHAR(16) NOT NULL);
9
10 CREATE TABLE IF NOT EXISTS games(
11  name VARCHAR(32),
12  version CHAR(3),
13  price NUMERIC NOT NULL,
14  PRIMARY KEY (name, version));
15
16 CREATE TABLE downloads(
17  customerid VARCHAR(16) REFERENCES customers(customerid)
18     ON UPDATE CASCADE ON DELETE CASCADE
19     DEFERRABLE INITIALLY DEFERRED,
20  name VARCHAR(32),
21  version CHAR(3),
22  PRIMARY KEY (customerid, name, version),
23  FOREIGN KEY (name, version) REFERENCES games(name, version)
24     ON UPDATE CASCADE ON DELETE CASCADE
25     DEFERRABLE INITIALLY DEFERRED);
```

A simple SQL query includes a SELECT clause, which indicates the columns to be printed, a FROM clause, which indicates the table(s) to be queried and possibly a WHERE clause, which indicates a possible condition on the records to be printed. We have seen the following query that displays the first and last names of registered customers from Singapore.

```
1 SELECT first_name, last_name
2 FROM customers
3 WHERE country = 'Singapore';
```

first_name	last_name
"Deborah"	"Ruiz"
"Tammy"	"Lee"
"Walter"	"Leong"
...	

The following two queries print all columns and all the rows of the `customers` table. They print the entire `customers` table.

```
SELECT first_name, last_name, email, dob, since,
  customerid, country
FROM customers;
```

The asterisk is a shorthand indicating that all the column names, in order of the `CREATE TABLE` declaration, should be considered in the `SELECT` clause.

```
SELECT *
FROM customers;
```

first_name	last_name	email	dob	since	customerid	country
Deborah	Ruiz	druiz0@drupal.org	1984-08-01	2016-10-17	Deborah84	Singapore
Rebecca	Garza	rgarza2@cornell.edu	1984-06-11"	2016-09-26	RebeccaG84	Malaysia"
Walter	Leong	wleong3@shop-pro.jp	1983-06-26	"2016-06-12"	Walter83	Singapore
...						

We can select a subset of the columns of the entire table.

```
1 SELECT first_name, last_name
2 FROM customers;
```

first_name	last_name
Deborah	Ruiz
Rebecca	Garza
Walter	Leong
...	

Selecting a subset of the columns of a table may result in duplicate rows even if the original table has a primary key.

```
1 SELECT name, version
2 FROM downloads;
```

This can be made clear if we order the result,

```
1 SELECT name, version
2 FROM downloads
3 ORDER BY name, version;
```

name	version
"Aerified"	"1.0"
"Aerified"	"1.0"
"Aerified"	"1.0"
...	

We order the result according to the name and then the version. If two games have the same name, then we use their version to order them.

```
1 SELECT name, version
2 FROM downloads
3 ORDER BY name, version;
```

name	version
...	
"Aerified"	"1.0"
"Aerified"	"1.0"
"Aerified"	"1.1"
...	

If we only order the result according to the name, the games with different versions are in no particular order.

```
1 SELECT name, version
2 FROM downloads
3 ORDER BY name;
```

name	version
"Aerified"	3.0
"Aerified"	1.0
"Aerified"	1.1
...	

We can order the result according to a column that is not printed.

```
1 SELECT name, version
2 FROM games
3 ORDER BY price;
```

name	version
"Duobam"	"1.2"
"Aerified"	"1.2"
"Tres-Zap"	"1.1"
...	

Always use `ASC` and `DESC` to indicate whether you want the result of the query to be sorted in ascending or descending order, respectively, according to each field in the ORDER BY clause. Do not rely on the fact that `ASC` is the default (add `ASC` after the fields in the ORDER BY clauses above).

The keyword DISTINCT eliminates eventual duplicates in the result of the query. It requests that the results contains distinct rows (it does not apply to columns individually).

```
1 SELECT DISTINCT name, version
2 FROM downloads;
```

name	version
"Zathin"	"3.0"
"Pannier"	"2.0"
"Tempsoft"	"2.0"
...	

The query with DISTINCT displays 422 records. The query without DISTINCT displays 4214 records. We shall see later how we can count.

DISTINCT, very often, gives a sorted result but we cannot rely on this behaviour. This is an unguaranteed side-effect of the sorting algorithm used to eliminate duplicates.

```
1 SELECT DISTINCT name
2 FROM games;
```

name
Aerified
Alpha
Alphazap
...

You can combine the different constructs but not always.

```
1 SELECT DISTINCT name
2 FROM games
3 ORDER BY name;
```

name
Aerified
Alpha
Alphazap
...

Not all combinations of all these constructs make sense. Both DISTINCT and ORDER BY involve sorting and conceptually ORDER BY is applied before SELECT DISTINCT.

```
1 SELECT DISTINCT name, version
2 FROM games
3 ORDER BY price;
```

```
1 ERROR:  for SELECT DISTINCT, ORDER BY expressions must appear in select list
2 LINE 3: ORDER BY price;
3                   ^
4 SQL state: 42P10
5 Character: 52
```

The `WHERE` clause is used to filter rows on a Boolean condition. The Boolean condition uses Boolean operators such as `AND`, `OR` and `NOT`, and various comparison operators such as >, <, >=, <=, <>, `IN`, `LIKE` and `BETWEEN AND`.

```
1 SELECT first_name, last_name
2 FROM customers
3 WHERE country IN ('Singapore', 'Indonesia')
4 AND (dob BETWEEN '2000-01-01' AND '2000-12-01' OR since >= '2016-12-01')
5 AND last_name LIKE 'B%';
```

first_name	last_name
"Jonathan"	"Bailey"
"Janice"	"Burns"
"Debra"	"Bishop"
...	

One can use arithmetic and other functions in the `SELECT` and `WHERE` clauses (as well as in the `ORDER BY`, `GROUP BY` and `HAVING` clauses).

```
1 SELECT DISTINCT price * .07 AS gst
2 FROM games
3 ORDER by gst;
```

gst
0.1393
0.2093
0.2793
0.35
0.84

In the example above, we calculate the GST (7%) for our games.

`AS` is used to rename the columns in the result.

The government decides not to collect the GST below 30 cents.

```
1 SELECT name || ' ' || version  AS game, price * 1.07
2 FROM games
3 WHERE price * 0.07 >= 0.3;
```

game	price
"Aerified 1.0"	12.84
"Aerified 2.0"	5.35
"Aerified 2.1"	12.84
"Alpha 1.0"	12.84
...	

```
1 SELECT name || ' ' || version  AS game, price
2 FROM games
3 WHERE price * 0.07 < 0.3;
```

game	price
"Aerified 1.1"	3.99
"Aerified 1.2"	1.99
...	

We could also write the following single query, but it is generally not preferred.

```
1 SELECT name || ' ' || version  AS game,
2     CASE
3     WHEN price * 0.07 >= 0.3 THEN  price * 1.07
4     ELSE price
5     END
6 FROM games;
```

game	price
"Aerified 1.0"	12.84
"Aerified 1.1"	3.99
"Aerified 1.2"	1.99
"Aerified 2.0"	5.35
...	

The syntax of operations and functions can be specific to the DBMS used. For example, Oracle and MySQL use the "`CONCAT ()`" function while PostgreSQL and SQLite use "||" instead of "&" used by Microsoft Access 2010 and "+" by SQL Server.

What is the difference between the following queries?

```sql
SELECT first_name, last_name
FROM customers
WHERE (country = 'Singapore' OR country = 'Indonesia')
AND ((dob >= '2000-01-01' AND dob <= '2000-12-01') OR since >= '2016-12-01')
AND last_name LIKE 'B%';
```

```sql
SELECT first_name, last_name
FROM customers
WHERE (country = 'Singapore' OR country = 'Indonesia')
AND last_name LIKE 'B%'
AND (since >= '2016-12-01' OR NOT (dob < '2000-01-01' OR dob > '2000-12-01'));
```

```sql
SELECT first_name, last_name
FROM customers
WHERE (country = 'Singapore' OR country = 'Indonesia')
AND (dob BETWEEN '2000-01-01' AND '2000-12-01' OR since >= '2016-12-01')
AND last_name LIKE 'B%'
```

De Morgan's Laws

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

```
1 SELECT name
2 FROM games
3 WHERE (version = '1.0' OR version = '1.1');
```

```
1 SELECT name
2 FROM games
3 WHERE version IN ('1.0','1.1');
```

```
1 SELECT name
2 FROM games
3 WHERE NOT (version <> '1.0' AND version <> '1.1');
```

The logic is that of Boolean expressions.

P	Q	P AND Q	P OR Q	NOT P
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

"`SELECT FROM WHERE <condition>`" returns results when the `<condition>` is true.

Every domain has an additional value: the null value. In general the semantics of null values could be ambiguous. It could be "unknown", "does not exists", "unknown or does not exists".
In SQL it is generally (but not always) "unknown".

```
1 CREATE TABLE example (
2 column1 VARCHAR(32),
3 column2 NUMERIC);
```

```
1 INSERT INTO example VALUES ('abc', 1);
2 INSERT INTO example VALUES ('def', null);
3 INSERT INTO example VALUES ('ghi', null);
```

```
1 SELECT *
2 FROM example;
```

column1	column2
"abc"	1
"def"	
"ghi"	

"`something = null`" is unknown (even if "`something`" is "`null`").
"`something <> null`" is unknown.
"`something > null`" is unknown.
"`something < null`" is unknown.
etc.
"`10 + null`" is null.
"`0 * null`" is null!

```
1 SELECT column1, column2
2 FROM example
3 WHERE column2 > 0;
```

column1	column2
"abc"	1

```
1 SELECT column1, column2
2 FROM example
3 WHERE column2 <= 0;
```

column1	column2

```
1 SELECT column1, column2 * 0 AS newcolumn2
2 FROM example;
```

column1	newcolumn2
"abc"	0
"def"	
"ghi"	

With null values, the logic of SQL is a three valued logic with unknown.

P	Q	P AND Q	P OR Q	NOT P
True	True	True	True	False
True	False	False	True	False
True	Unknown	Unknown	True	False
False	True	False	True	True
False	False	False	False	True
False	Unknown	False	Unknown	True
Unknown	True	Unknown	True	Unknown
Unknown	False	False	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown

"`SELECT FROM WHERE <condition>`" returns results when the `<condition>` is true.

SQL has Boolean operators (`ISNULL` or `IS NULL` and `IS NOT NULL`) to check null values and functions to manipulate null values (dependeing on the database management system: `IFNULL()`, `ISNULL()`, `COALESCE()`, and `NVL()`).

"`null ISNULL`" is true.

"`COALESCE()`" returns the first non-null of its argument.

```
1 SELECT column1, column2, COALESCE(column2, 0)
2 FROM example
3 WHERE column2 = NULL;
```

column1	column2	newcolumn2

```
1 SELECT column1, column2, COALESCE(column2, 0)
2 FROM example
3 WHERE column2 IS NULL;
```

column1	column2	newcolumn2
"def"		0
"ghi"		0

```
1 SELECT column1, column2,
2            (CASE WHEN column2 IS NULL
3             THEN 0
4             ELSE column2 END) AS newcolumn2
5 FROM example;
```

column1	column2	newcolumn2
"abc"	1	1
"def"		0
"ghi"		0

"`COUNT(*)`" counts NULL values.
"`COUNT(att)`", "`AVG(att)`", "`MAX(att)`", "`MIN(att)`" eliminate null values.

What happens if we query several tables in the same query?

```
1 SELECT *
2 FROM customers, downloads, games;
```

The result is the table that contains all the columns and all the possible combinations of the rows of the three tables. This is called a Cartesian product (named after the French mathematician René Descartes) or a cross product. Is it useful? Is it inefficient?

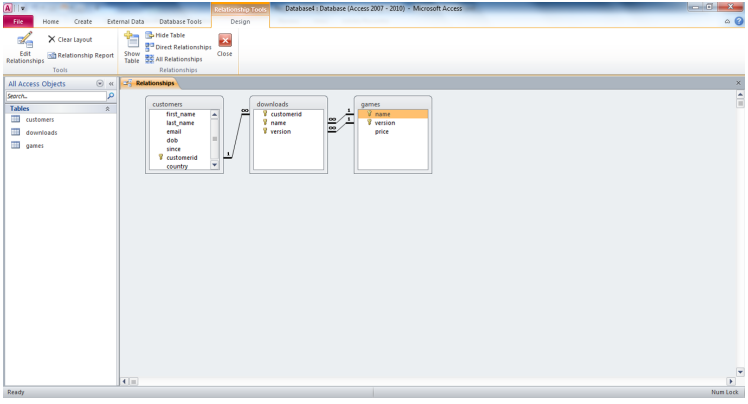
```
1 SELECT *
2 FROM customers CROSS JOIN downloads CROSS JOIN games;
```

The comma in the FROM clause is a convenient shorthand for the key word `CROSS JOIN`. Consequently, nobody uses the key word `CROSS JOIN` and uses the comma instead.

```
1 SELECT *
2 FROM customers, downloads, games;
```

In the example above the table containing the result has $7 + 3 + 3 = 13$ columns (the number of columns of the table `customers` plus the number of columns of the table `games` plus the number of columns of the table `downloads`)
$1000 \times 430 \times 4214 = 1,812,020,000$ rows [a] (the number of rows of the table `customers` times the number of rows of the table `games` times the number of rows of the table `downloads`)

[a] one billion , eight hundred twelve million , twenty thousand.



We should join the tables according to their primary keys and foreign keys to make sense of the data,

The picture is a graphical representation of the logical schema of the database (the tables with their columns and the references from the foreign keys to the primary keys). This picture was produced with Microsoft Access for the code in the file `AppStoreSchema.sql`. Many authors call this graphical representation an entity-relationship diagram. This is wrong. This is a logical diagram. We will see later that the entity-relationship diagram is a conceptual diagram.

We add the condition that foreign key columns are equal to the corresponding primary key columns.

```
1 SELECT *
2 FROM customers c, downloads  d, games g
3 WHERE d.customerid = c.customerid
4 AND d.name = g.name
5 AND d.version = g.version;
```

The result has 13 columns (the number of columns of the table `customers` plus the number of columns of the table `games` plus the number of columns of the table `downloads`) and 4214 rows (the meaningful combinations of a row from each of the three tables corresponding to a customer downloading a game.)

It is recommended (in fact compulsory from now on for this module) to systematically define table variables for each table in the FROM clause and to use these variables and the dot notation for the disambiguation of the column names elsewhere, even when there is no ambiguity. It allows the system to distinguish between columns with the same name. The variables can also be define using the keyword `AS`.

```
1 SELECT *
2 FROM customers c, downloads  d, games g
3 WHERE d.customerid = c.customerid
4 AND d.name = g.name
5 AND d.version = g.version;
```

```
1 SELECT *
2 FROM customers AS c, downloads AS d, games AS g
3 WHERE d.customerid = c.customerid
4 AND d.name = g.name
5 AND d.version = g.version;
```

```
1 SELECT *
2 FROM customers, downloads, games
3 WHERE customerid = customerid
4 AND name = name
5 AND version = version;
```

```
1 ERROR:  column reference "customerid" is ambiguous
2 LINE 3: WHERE customerid = customerid
3                ^
4 SQL state: 42702
5 Character: 49
```

In the example above we omitted the table variables c, g and d. The system cannot understand which column is which.

What does this query find? (in English).

```
1 SELECT c.email, a.version
2 FROM customers c, downloads d, games a
3 WHERE c.customerid = d.customerid AND a.name = d.name AND a.version = d.version
4 AND c.country = 'Indonesia' AND a.name = 'Fixflex';
```

email	version
adaybi@google.co.uk	2.0
awright3v@ebay.com	3.0
chernandeznk@skyrock.com	2.0
...