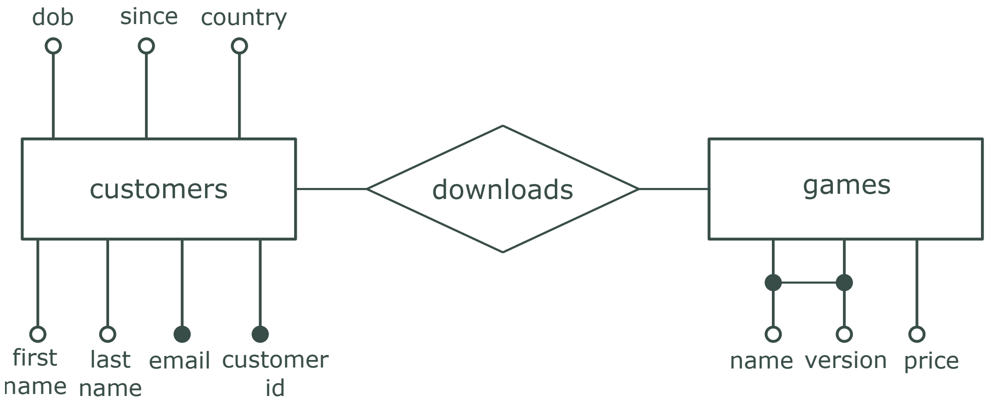
Creating and Populating Tables with Constraints

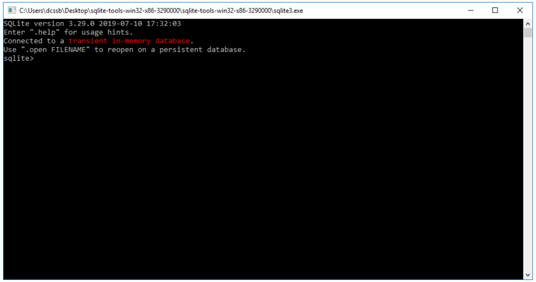
Stéphane Bressan

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
downloads
games
first name
last name
email
customer id
name
version
price

SQLite Installation

Download SQLite3 from `www.sqlite.org`. Go to the download page. Choose the precompiled binaries for your operating system and download a bundle of command-line tools for managing SQLite database files. Extract the zip file. You can now start the SQLite command-line shell program by clicking on `sqlite3.exe`.



C:\Users\dcssb\Desktop\sqlite-tools-win32-x86-3290000\sqlite-tools-win32-x86-3290000\sqlite3.exe

```
SQLite version 3.29.0 2019-07-10 17:32:03
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

SQLite Installation

```
sqlite> .open myfile.db
sqlite> .mode column
sqlite> .header on
sqlite> PRAGMA foreign_keys = ON;
sqlite> .help
...
sqlite> .quit
```

`.open` opens or creates a database (file). `.mode` and `.header` define the settings for the display of results. the `PRAGMA` statement enables foreign key constraints checking.`.quit` saves the database and quits. `.help` lists the available commands.

The language for this SQLite shell is made of commands and SQL statements.

Commands (starting with ".") are part of SQLite proprietary language. SQL statements can be typed directly. SQL statements end with a semicolon ";" (warning: you will forget it.)

You may skip ".open" for this lecture. That step creates a persistent database but also may slow down some update operations. All your work will be lost.

In theory, the relational model proposes to organise data in relations. relations have a name. Relations have attributes. Attributes have a name. Mathematically, relations are subsets of the Cartesian product of the domains of their attributes. Domains extensions of types.

In practice, the relational database management systems organise data in tables. Tables have a name. Tables are multi-sets (not lists) of rows or records. Rows or records have fields corresponding to the columns of the table. Columns or fields have a name. Columns or fields also have an implicit position indicated by the order in the corresponding creation statement. Columns or fields have a domain which is a type to the extension of which is added the possibility of a null value.

We shall discuss in a subsequent lecture the syntax, semantics, and behaviour of null values and of the constructs that are used to manipulate them.

Logical design is the activity consisting in choosing the appropriate schema for the database application. It consists mainly the number of tables, the names of the tables, the number of columns each table should have and the names and domains of the columns.

Creating Tables

```sql
CREATE TABLE downloads(
    first_name VARCHAR(64),
    last_name VARCHAR(64),
    email VARCHAR(64),
    dob DATE,
    since DATE,
    customerid VARCHAR(16),
    country VARCHAR(16),
    name VARCHAR(32),
    version CHAR(3),
    price NUMERIC);
```

The choice of the number of columns and of their domains implicitly imposes structural constraints. For instance, if we use only one table, there can be no customer without a game and no game without a customer, unless we use null values. We can use the structural constraints as integrity constraints to control and maintain the integrity of data because data that does not fit in the table cannot be stored and only data that fit can be stored.

Creating Tables

```
CREATE TABLE customers (
  first_name VARCHAR(64),
  last_name VARCHAR(64),
  email VARCHAR(64),
  dob DATE,
  since DATE,
  customerid VARCHAR(16),
  country VARCHAR(16));
```

```
CREATE TABLE games (
  name VARCHAR(32),
  version CHAR(3),
  price DECIMAL(2,2));
```

```
CREATE TABLE downloads(
  customerid VARCHAR(16),
  name VARCHAR(32),
  version CHAR(3));
```

We opt for a schema comprising the three tables above with the indicated fields and their respective domains.

Integrity Constraints

A consistent state of the database is a state which complies with the with the business rules as defined by the structural constraints and the integrity constraints in the schema.

SQL allows to express business rules, such as "*students who have not passed cs1010 cannot take cs2020*" as integrity constraints.

If an integrity constraint is violated by an operation or a transaction, the operation or the transaction is aborted and rolled back and its changes are undone, otherwise, it is committed and its changes are effective for all users.

In practice, there are different ways to specify the scope of transactions. One of them is to use blocks with keywords such as `BEGIN`, `END` or `COMMIT`, `ABORT` and `ROLLBACK`.

In most systems, unfortunately, integrity constraints are immediate, i.e. they are checked after each insertion, update and deletion. It is preferable to set all integrity constraints to be differed, i.e. they are checked after the end of transactions. Unfortunately again, some systems, like PostgreSQL do only allow certain constraints to be deferred.

SQL supports five kinds of integrity constraints:

- `NOT NULL`,
- `PRIMARY KEY`,
- `UNIQUE`,
- `FOREIGN KEY` and
- `CHECK`.

Before recreating the tables with different definitions, let us delete the existing tables. The same applies for the subsequent examples with the same or other tables.

```sql
DELETE FROM customers;
```

DELETE deletes content of the table but not its definition. In the example above, the content of the customers table is deleted. The table still exists and is empty. Instead, one should use DROP.

```sql
DROP TABLE customers;
```

DROP deletes the content of the table and its definition. the table does not exists any more.

We recommend `www.w3schools.com/sql` for details of the syntax and behaviour of standard SQL constructs. For proprietary syntax and behaviour, you may want to consult the SQLite documentation at `www.sqlite.org/docs.html` and the PostgreSQL documentation at `www.postgresql.org/docs`, respectively.

Primary Keys

```
CREATE TABLE customers (
first_name VARCHAR(64),
last_name VARCHAR(64),
email VARCHAR(64),
dob DATE,
since DATE,
customerid VARCHAR(16) PRIMARY KEY,
country VARCHAR(16));
```

A primary key is a set of attributes that identifies uniquely a record. You cannot have two records with the same value of their primary key in the same table. We can declare a primary key column constraint with the keyword `PRIMARY KEY`. Each table has only one primary key. In the example above, the primary key of a customer is its customer identification number.

It is common practice to underline the primary key when informally writing the schema. $customers(first\_name, last\_name, email, dob, since, \underline{customerid}, country)$

We insert one customer.

```
INSERT INTO customers VALUES(
    'Carole',
    'Yoga',
    'cyoga@glarge.org',
    '1989-08-01',
    '2016-09-15',
    'Carole89',
    'France');
```

We insert another customer with the same customer identification number.

```sql
INSERT INTO customers VALUES(
  'Carole',
  'Yoga',
  'cyoga@glarge.org',
  '1989-08-01',
  '2016-09-15',
  'Carole89',
  'France');
```

```
UNIQUE constraint failed: customers.customerid
```

The primary key constraint prevents any operation or transaction that violates the constraint, for instance, inserting another customer with the same customer identification number. The operation or the transaction is aborted and rolled back.

In the SQL standard and in most systems prime attributes (attributes composing a primary key) cannot be null. This is not the case in SQLite and IBM DB2. In these two systems one must explicitly declare a `NOT NULL` constraint for these attributes.

Primary Keys

```sql
CREATE TABLE games(
name VARCHAR(32),
 version CHAR(3),
 price NUMERIC,
PRIMARY KEY (name, version));
```

The primary key can be composite: it is the combination of several attributes. The composite primary key constraint is declared as a table constraint with the keyword `PRIMARY KEY` after the row declarations. In the example above, the primary key of a game is the combination of its name and version.

It is common practice to underline primary key attributes (prime attributes) when informally writing the schema. $games(\underline{name}, \underline{version}, price)$

Primary Keys

```
INSERT INTO games VALUES ('Aerified2', '1.0', 5);
INSERT INTO games  VALUES ('Aerified2', '1.0', 6);
```

```
Error: UNIQUE constraint failed: games.name, games.version
```

One cannot have two games with the same name and version. There however can be several games with the same name and several games with the same version.

```
INSERT INTO games VALUES ('Aerified2', '2.0', 5);
INSERT INTO games  VALUES ('Aerified3', '1.0', 6);
```

```sql
CREATE TABLE games(
name VARCHAR(32),
version CHAR(3),
price NUMERIC NOT NULL);
```

A not null constraint guarantees that no value of the corresponding field in any record of the table can be set to null. A not null constraint is always declared as a row constraint. When it is explicit, it is declared with the keyword `NOT NULL`. In the example above, the price of a game can never be null.

Null Values

```
INSERT INTO games (name, version) VALUES ('Aerified2', '1.0');
```

```
INSERT INTO games VALUES ('Aerified2', '1.0', null);
```

```
Error: NOT NULL constraint failed: games.price
```

Both operations above attempt to insert a null value for the price. An error is raised. The operation or the transaction violating the constraints is aborted and rolled back.

Alternatively, we could set a default value at table creation time with the following declaration when creating the table `games`.

```
price NUMERIC DEFAULT 1.00,
```

According to the SQL standard and in most systems declaring a `NOT NULL` constraint on a prime attribute raises an error. This can be justified by the fact that a `PRIMARY KEY` constraint is equivalent, from the point of view of integrity constraints, as a combination of `UNIQUE` and `NOT NULL` constraints. This is not the case in for SQLite and IBM DB2, for which the `NOT NULL` constraint must be added explicitly if one wants to prevent prime attributes from being null. This is particularly annoying as it hinders the portability of the otherwise standard SQL DDL code. In addition, a primary key has other effects not discussed here on the tuning of the system (it creates and maintains indexes for fast access).

SQLite Apology: "According to the SQL standard, PRIMARY KEY should always imply NOT NULL. Unfortunately, due to a bug in some early versions, this is not the case in SQLite. Unless the column is an INTEGER PRIMARY KEY or the table is a WITHOUT ROWID table or the column is declared NOT NULL, SQLite allows NULL values in a PRIMARY KEY column. SQLite could be fixed to conform to the standard, but doing so might break legacy applications. Hence, it has been decided to merely document the fact that SQLite allowing NULLs in most PRIMARY KEY columns."
`http://www.sqlite.org/lang_createtable.html`

```sql
CREATE TABLE customers (
  first_name VARCHAR(64),
  last_name VARCHAR(64),
  email VARCHAR(64) UNIQUE,
  dob DATE,
  since DATE,
  customerid VARCHAR(16),
  country VARCHAR(16),
  UNIQUE (first_name, last_name));
```

A unique constraint on an attribute or a combination of attributes guarantees the table cannot contain two records with the same value in the corresponding field or combination of fields. This is declared as a row or a table constraint with the `UNIQUE` keyword. In the example above, the email of a customer must be unique and the combination of her first name and last name must be unique. This is the same as a primary key constraint except that it does allow null values.

Foreign Key

A foreign key constraint enforces referential integrity. The values in the columns for which the constraint is declared must exists in the corresponding columns of the referenced table.

Foreign Key

The referenced columns are usually required to be the primary key of the referenced table. Some systems relax this requirement. We shall enforce it strictly for the sake of portability.

```
CREATE TABLE customers (
 first_name VARCHAR(64),
 last_name VARCHAR(64),
 email VARCHAR(64),
 dob DATE,
 since DATE,
 customerid VARCHAR(16) PRIMARY KEY,
 country VARCHAR(16));

CREATE TABLE games(
 name VARCHAR(32),
 version CHAR(3),
 price NUMERIC,
 PRIMARY KEY (name, version));
```

Foreign Key

```
CREATE TABLE downloads(
customerid VARCHAR(16) REFERENCES customers (customerid),
name VARCHAR(32),
version CHAR(3),
FOREIGN KEY (name, version) REFERENCES games(name, version));
```

A foreign key is declared using the keyword `REFERENCES` as a row constraint and the keywords `FOREIGN KEY` and `REFERENCES` as a table constraint. In the example above, the customer identification number and the combination of name and version of the game she downloaded as recorded in the `downloads` table must correspond to an existing customer and an existing game, they must be the corresponding primary keys, `customerid` in the `customers` table and the combination of `name` and `version` in the `games` table, respectively. Note that, surprisingly, a null value does not violate referential integrity in SQL.

Foreign Key

```
CREATE TABLE downloads(
customerid VARCHAR(16) REFERENCES customers (customerid),
name VARCHAR(32),
version CHAR(3),
FOREIGN KEY (name, version REFERENCES games(name, version));
```

When the primary key referenced is composite, the `FOREIGN KEY` constraint is declared as a table constraint using the additional keyword `FOREIGN KEY`. All foreign key constraints can be declared as table constraints.

Foreign Key

```
INSERT INTO downloads VALUES ('Adam1983', 'Biodex', '1.0');
```

```
Error: FOREIGN KEY constraint failed
```

The column `customerid` in the table `downloads` references the primary key `customerid` of the table `customers` The column `customerid` in the table `downloads` can only take values that appears in the column `customerid` of the table `customers`. The customer `Adam1983` does not exist. The same thing happens for the game `Biodex 1.0`.

Foreign Key

```
INSERT INTO customers VALUES ('Deborah', 'Ruiz', 'druiz0@drupal.org', '1984-08-01', '2016-10-17', '
      Deborah84', 'Singapore');
INSERT INTO games VALUES ('Aerified', '1.0', 12);
INSERT INTO downloads VALUES ('Deborah84', 'Aerified', '1.0');
```

```
DELETE FROM customers WHERE country='Singapore';
```

```
Error: FOREIGN KEY constraint failed
```

We cannot delete the Singapore customers because some of them have downloaded some games.

Check

```
CREATE TABLE games (
  name VARCHAR(32),
  version CHAR(3),
  price DECIMAL(2,2) NOT NULL CHECK (price > 0));
```

A check constraint enforces any other condition that can be expressed in SQL. A check constraint is declared as a row or a table constraint with the `CHECK` keyword followed by the SQL condition in parenthesis. If the condition involves more than one row, it has to be a table constraint. In the example above, the price of game can not be zero or negative.

Check

The SQL standard caters for very expressive CHECK constraints. It even allows that CHECK constraints can be implemented as assertions outside tables (using the construct `CREATE ASSERTION`) so that they can involve several tables and use aggregate functions, nested queries etc. In practice, the vendors only offer check constraints that are limited to very simple row and table checks. This is a great pity since the technology exists, is very well understood and its non availability jeopardises the integrity of enterprise data. The vendors usually advocate the use of triggers and stored functions, which is notoriously difficult and, therefore, error-prone. It largely defeats the objective of integrity management.

Check

```
INSERT INTO games VALUES ('Aerified', '1.0', 12);
INSERT INTO games VALUES ('Aerified', '1.1', 3.99);
```

```
UPDATE games SET price = price - 10;
```

```
Error: CHECK constraint failed: games
```

Discounting all the prices by 10 dollars creates negative prices. This operation is aborted and rolled back.

Consider the following schema.

```
CREATE TABLE IF NOT EXISTS customers (
 first_name VARCHAR(64) NOT NULL,
 last_name VARCHAR(64) NOT NULL,
 email VARCHAR(64) UNIQUE NOT NULL,
 dob DATE NOT NULL,
 since DATE NOT NULL,
 customerid VARCHAR(16) PRIMARY KEY,
 country VARCHAR(16) NOT NULL);

CREATE TABLE downloads(
 customerid VARCHAR(16) REFERENCES customers(customerid)
 name VARCHAR(32),
 version CHAR(3),
 PRIMARY KEY (customerid, name, version),
```

Propagating Updates and Deletions

downloads

name	version	customerid
Skype	1.0	tom1999
Comfort	1.1	john88
Skype	2.0	tom1999
. . .		

customers

customerid	email	. . .
tom1999	tlee@gmail.com	. . .
john88	al@hotmail.com	. . .
walnuts	dcs@nus.edu.sg	. . .
. . .		

What happens if the customer identification number `john88` is changed to `john1988` in the table `downloads`?
What happens if the customer identification number `tom1999` is changed to `thom1999` in the table `customers`?
What happens if the record for the customer with identification number `tom1999` is deleted from the table `customer`?

Propagating Updates and Deletions

```
CREATE TABLE downloads(
 customerid VARCHAR(16) REFERENCES customers(customerid)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
 name VARCHAR(32),
 version CHAR(3),
 PRIMARY KEY (customerid, name, version),
 FOREIGN KEY (name, version) REFERENCES games(name, version)
    ON UPDATE CASCADE
    ON DELETE CASCADE);
```

The annotations `ON UPDATE/DELETE` with the option `CASCADE` propagate the update or deletion. They can also be used with the self-describing options: `NO ACTION`, `SET DEFAULT` and `SET NULL`.

Propagating Updates and Deletions

Update and deletion cascades are powerful mechanisms that can result in chain reactions when they are chains of foreign key dependencies.

There is an even more powerful but difficult to program generalisation of such propagation mechanisms in the form of triggers using stored functions. Triggers with stored procedures are very expressive but leave the responsibility of the control to the programmer.

The exact syntax and behaviour of these constructs may vary from one DBMS to another and from version to the next.

This is the complete schema for our example

```
CREATE TABLE IF NOT EXISTS customers (
 first_name VARCHAR(64) NOT NULL,
 last_name VARCHAR(64) NOT NULL,
 email VARCHAR(64) UNIQUE NOT NULL,
 dob DATE NOT NULL,
 since DATE NOT NULL,
 customerid VARCHAR(16) PRIMARY KEY,
 country VARCHAR(16) NOT NULL);

CREATE TABLE IF NOT EXISTS games(
 name VARCHAR(32),
 version CHAR(3),
 price NUMERIC NOT NULL,
 PRIMARY KEY (name, version));

 CREATE TABLE downloads(
 customerid VARCHAR(16) REFERENCES customers(customerid)
    ON UPDATE CASCADE ON DELETE CASCADE
    DEFERRABLE INITIALLY DEFERRED,
 name VARCHAR(32),
 version CHAR(3),
 PRIMARY KEY (customerid, name, version),
 FOREIGN KEY (name, version) REFERENCES games(name, version)
    ON UPDATE CASCADE ON DELETE CASCADE
    DEFERRABLE INITIALLY DEFERRED);
```

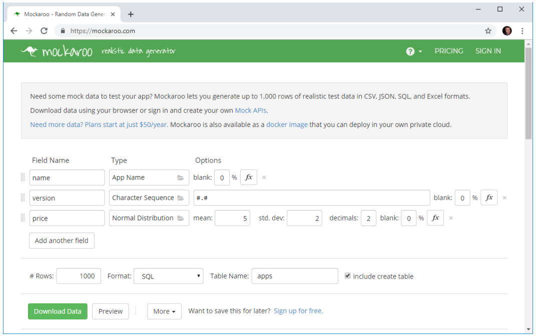
It is generally a good idea to constraint all attributes not to be null unless there is a good design or tuning reason for not doing so.

Think carefully about which foreign keys should be subject to cascade.

It is generally a good idea to defer all the constraints that can be deferred. A deferred constraint is checked at the end of a transaction and not immediately after each operation.

Given the complete schema for the three tables, try and find out the scenarii in which an operation (insertion, deletion, update) on a table or a transaction containing a set of operation on one or more tables violate which constraint on which table.

Populating Tables



Need some mock data to test your app? Mockaroo lets you generate up to 1,000 rows of realistic test data in CSV, JSON, SQL, and Excel formats.
Download data using your browser or sign in and create your own Mock APIs.
Need more data? Plans start at just $50/year. Mockaroo is also available as a docker image that you can deploy in your own private cloud.

Field Name	Type	Options
name	App Name	blank: 0 %
version	Character Sequence	#.# blank: 0 %
price	Normal Distribution	mean: 5 std. dev: 2 decimals: 2 blank: 0 %

Rows: 1000 Format: SQL Table Name: apps ☑ include create table

Want to save this for later? Sign up for free.

We generate the customers and games tables with Mockaroo (www.mockaroo.com).

The generator generates an SQL file with the CREATE TABLE and INSERT statements.

```
CREATE TABLE games (
  name VARCHAR(50),
   version VARCHAR(50),
   price DECIMAL(2,2)
);
INSERT INTO games (name, version, price) VALUES ('Voyatouch', '6.6', 6.36);
INSERT INTO games (name, version, price) VALUES ('Voltsillam', '7.4', 3.76);
INSERT INTO games (name, version, price) VALUES ('Stim', '8.2', 3.04);
INSERT INTO games (name, version, price) VALUES ('Y-Solowarm', '8.9', 9.99);
INSERT INTO games (name, version, price) VALUES ('Quo Lux', '9.9', 2.64);
INSERT INTO games (name, version, price) VALUES ('Biodex', '5.3', 7.3);
INSERT INTO games (name, version, price) VALUES ('Biodex', '8.7', 4.13);
```

Populating Tables

You can now delete the tables you have created. DROP deletes both the content of the table and the table definition.

```
sqlite> DROP TABLE downloads
sqlite> DROP TABLE customers
sqlite> DROP TABLE games
```

From Luminus download the files [a] `AppStoreSchema.sql` `AppStoreCustomers.sql`, `AppStoreGames.sql` and `AppStreDownloads.sql`. Place them in the SQLite directory (or use the SQLite `.cd` command to point at their location, in which case you may need to open a new database file). These four SQL files (you can read them) are creating and populating the database or our application. Execute them. `.read` executes the SQL files in SQLite. The file `AppStoreClean.sql` cleans up the database if you need to start again.

[a]We use variants of the same data and that the results on your computer may differ slightly from the results given in the slides for illustration purposes.

```
sqlite> .read AppStoreSchema.sql
sqlite> .read AppStoreCustomers.sql
sqlite> .read AppStoreGames.sql
sqlite> .read AppStoreDownloads.sql
```

The following query prints the customers table.

```
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
. . .						

The following query prints the games table.

```sql
SELECT *
FROM games;
```

name	version	price
Aerified	1.0	12
Aerified	1.1	3.99
Aerified	1.2	1.99
...		

The following query prints the `downloads` table.

```sql
SELECT *
FROM downloads;
```

customerid	name	version
Adam1983	Biodex	1.0
Adam1983	Domainer	2.1
Adam1983	Subin	1.1
. . .		

You can give a name to a query. This is called a view. Once created, a view can be queried like any other table.

```sql
CREATE VIEW singapore_customers1 AS
SELECT c.first_name, c.last_name, c.email, c.dob, c.since, c.customerid
FROM customers c
WHERE country='Singapore';
```

```sql
SELECT *
FROM singapore_customers1;
```

Creating a view is generally a better option than creating and populating a table, temporary or not.

```
CREATE TABLE singapore_customers2 (
  first_name VARCHAR(64) NOT NULL,
  last_name VARCHAR(64) NOT NULL,
  email VARCHAR(64) UNIQUE NOT NULL,
  dob DATE NOT NULL,
  since DATE NOT NULL,
  customerid VARCHAR(16) PRIMARY KEY REFERENCES customers(customerid));

INSERT INTO singapore_customers2
SELECT c.first_name, c.last_name, c.email, c.dob, c.since, c.customerid
FROM customers c
WHERE country='Singapore';
```

Logical Data Independence

There are further issues concerning views that we are not discussing here: view updates and materalised views, for instance.

Logical Data Independence

```
DROP VIEW singapore_customers1;
DROP TABLE singapore_customers2;
```