Principles of Database Systems (CS307)

Lecture 2: Introduction to SQL, and Basic SQL

Zhong-Qiu Wang

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

Introduction to SQL

How Do We Manage Data in a Database?

- Usually, a special language is needed
  - Be able to use a language to query a database
    - Either interactively or from within a program

- Query language
  - Query data
  - Modify data

Some History

- ALPHA
  - Codd’s querying language

- Find the supplier names and locations of those suppliers who supply part 15.

$$(r_1[2],\ r_1[3]):\ P_1r_1 \wedge \exists P_2 r_2 (r_2[2] = 15 \wedge r_2[1] = r_1[1]).$$

- Find the locations of suppliers and the parts being supplied by them (omitting those suppliers who are supplying no parts at this time).

$$(r_1[3],\ r_2[2]):\ P_1r_1 \wedge P_2r_2 \wedge (r_1[1] = r_2[1]).$$

Codd, E.F., "Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control, San Diego.

Some History

- ALPHA
  - Codd's querying language

- Find the supplier names and locations of those suppliers who supply part 15.

$$(r_1[2],\ r_1[3]):\ P_1r_1 \wedge \exists P_2r_2(r_2[2] = 15 \wedge r_2[1] = r_1[1]).$$

- Find the locations of suppliers and the parts being supplied by them (omitting those suppliers who are supplying no parts at this time).

$$(r_1[3],\ r_2[2]):\ P_1r_1 \wedge P_2r_2 \wedge (r_1[1] = r_2[1]).$$

Codd, E.F., "Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access, and Control, San Diego.

However, it didn't excite enthusiasm at IBM

Some History

- "SEQUEL: A Structured English Query Language"
  - IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
  - An "easy" language, with an English-like syntax

Don Chamberlin with Ray Boyce (1974)

SEQUEL: A STRUCTURED ENGLISH QUERY LANGUAGE

by

Donald D. Chamberlin
Raymond F. Boyce

IBM Research Laboratory
San Jose, California

ABSTRACT: In this paper we present the data manipulation facility for a structured English query language (SEQUEL) which can be used for accessing data in an integrated relational data base. Without resorting to the concepts of bound variables and quantifiers SEQUEL identifies a set of simple operations on tabular structures, which can be shown to be of equivalent power to the first order predicate calculus. A SEQUEL user is presented with a consistent set of keyword English templates which reflect how people use tables to obtain information. Moreover, the SEQUEL user is able to compose these basic templates in a structured manner in order to form more complex queries. SEQUEL is intended as a data base sublanguage for both the professional programmer and the more infrequent data base user.

Computing Reviews Categories: 3.5, 3.7, 4.2

Key Words and Phrases: Query Languages
Data Base Management Systems
Information Retrieval
Data Manipulation Languages

Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A structured English query language. In Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control (SIGFIDET '74). Association for Computing Machinery, New York, NY, USA, 249–264. DOI:https://doi.org/10.1145/800296.811515

Some History

- SEQUEL was then renamed as Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features
  - Not all examples here may work on your particular system.

A huge problem that appears everywhere and many times

Year	Name	Alias	Comments
1986	SQL-86	SQL-87	First formalized by ANSI
1989	SQL-89	FIPS 127-1	Minor revision that added integrity constraints, adopted as FIPS 127-1
1992	SQL-92	SQL2, FIPS 127-2	Major revision (ISO 9075), *Entry Level* SQL-92 adopted as FIPS 127-2
1999	SQL:1999	SQL3	Added regular expression matching, recursive queries (e.g. transitive closure), triggers, support for procedural and control-of-flow statements, nonscalar types (arrays), and some object-oriented features (e.g. structured types), support for embedding SQL in Java (SQL/OLB) and vice versa (SQL/JRT)
2003	SQL:2003		Introduced XML-related features (SQL/XML), window functions, standardized sequences, and columns with autogenerated values (including identity columns)
2006	SQL:2006		ISO/IEC 9075-14:2006 defines ways that SQL can be used with XML. It defines ways of importing and storing XML data in an SQL database, manipulating it within the database, and publishing both XML and conventional SQL-data in XML form. In addition, it lets applications integrate queries into their SQL code with XQuery, the XML Query Language published by the World Wide Web Consortium (W3C), to concurrently access ordinary SQL-data and XML documents.[33]
2008	SQL:2008		Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers, TRUNCATE statement,[34] FETCH clause
2011	SQL:2011		Adds temporal data (PERIOD FOR)[35] (more information at: Temporal database#History). Enhancements for window functions and FETCH clause.[36]
2016	SQL:2016		Adds row pattern matching, polymorphic table functions, JSON
2019	SQL:2019		Adds Part 15, multidimensional arrays (MDarray type and operators)

https://en.wikipedia.org/wiki/SQL

Standardization of SQL

Year	Name	Alias	Comments
1986	SQL-86	SQL-87	First formalized by ANSI
1989	SQL-89	FIPS 127-1	Minor revision that added integrity constraints, adopted as FIPS 127-1
1992	SQL-92	SQL2, FIPS 127-2	Major revision (ISO 9075), *Entry Level* SQL-92 adopted as FIPS 127-2
1999	SQL:1999	SQL3	Added regular expression matching, recursive queries (e.g. transitive closure), triggers, support for procedural and control-of-flow statements, nonscalar types (arrays), and some object-oriented features (e.g. structured types), support for embedding SQL in Java (SQL/OLB) and vice versa (SQL/JRT)
2003	SQL:2003		Introduced XML-related features (SQL/XML), window functions, standardized sequences, and columns with autogenerated values (including identity columns)
2006	SQL:2006		ISO/IEC 9075-14:2006 defines ways that SQL can be used with XML. It defines ways of importing and storing XML data in an SQL database, manipulating it within the database, and publishing both XML and conventional SQL-data in XML form. In addition, it lets applications integrate queries into their SQL code with XQuery, the XML Query Language published by the World Wide Web Consortium (W3C), to concurrently access ordinary SQL-data and XML documents.[33]
2008	SQL:2008		Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers, TRUNCATE statement,[34] FETCH clause
2011	SQL:2011		Adds temporal data (PERIOD FOR)[35] (more information at: Temporal database#History). Enhancements for window functions and FETCH clause.[36]
2016	SQL:2016		Adds row pattern matching, polymorphic table functions, JSON
2019	SQL:2019		Adds Part 15, multidimensional arrays (MDarray type and operators)

https://en.wikipedia.org/wiki/SQL

Standardization of SQL (any other examples of standardization?)

Basic Syntax of SQL

```
select * from lab where time = '3-34';
```

`select …`

- followed by the names of the columns you want to return

`from …`

- followed by the name of the tables that you want to query

`where …`

- followed by filtering conditions

Competing Languages of SQL

- QUEL, born at Berkeley, and associated with INGRES, was highly regarded
  - Ingres Database

QUEL ---------- Ingres → SEQUEL (SQL) ---------- Postgres

QUEL	SQL

`create student(name = c10, age = i4, sex = c1, state = c2)`
 	
`range of s is student`
`append to s (name = "philip", age = 17, sex = "m", state = "FL")`
 	
`retrieve (s.all) where s.state = "FL"`
 	
`replace s (age=s.age+1)`
 	
`retrieve (s.all)`
 	
`delete s where s.name="philip"`
	`create table student(name char(10), age int, sex char(1), state char(2));`
	
	`insert into student (name, age, sex, state) values ('philip', 17, 'm', 'FL');`
	
	`select * from student where state = 'FL';`
	
	`update student set age=age+1;`
	
	`select * from student;`
	
	`delete from student where name='philip';`

Michael Stonebraker (1943-)
Turing Award 2014

Competing Languages of SQL

- QBE (Query by Example)
  - A visual querying tool (visual but with characters)
  - Created by IBM as well

```
id:
address:
time: 3-34
capacity:
teacher:
```

```
select * from lab where time = '3-34';
```

Moshé M. Zloof

Two Main Components for a Query Language

- From Codd's seminal paper:
  - A good database language should allow to deal as easily with contents (data) as containers (tables)
  - … Something that SQL does reasonably well

Two Main Components for a Query Language

- Data Definition Language (DDL)
  - The SQL data-definition language (DDL) allows the specification of information about relations, including:
    - The schema for each relation.
    - The type of values associated with each attribute.
    - The Integrity constraints
    - The set of indices to be maintained for each relation.
    - Security and authorization information for each relation.
    - The physical storage structure of each relation on disk.

Two Main Components for a Query Language

- Data Definition Language (DDL)
  - The SQL data-definition language (DDL) allows the specification of information about relations, including:
    - The schema for each relation
    - The type of values associated with each attribute
    - The Integrity constraints
    - The set of indices to be maintained for each relation
    - Security and authorization information for each relation
    - The physical storage structure of each relation on disk

```
create
alter
drop
```

Two Main Components for a Query Language

- Data Manipulation Language (DML)
  - Provides the ability to:
    - Query information from the database
    - Insert tuples into, delete tuples from, and modify tuples in the database.

```
select
insert
delete
update
```

Something about SQL

- Simple?
  - As you can see, it seems to be simple
  - But it becomes difficult when you combine operations
    - We will talk about it later
- Standard?
  - We have mentioned standardization of SQL before
  - However, no product fully implements it
    - Different product implements SQL differently
    - … and introduces dialects

SQL is one of a few languages where you spend more time thinking about how you are going to do things than actually coding them.

“Problems” in SQL

- SQL ≠ Relational Database
  - SQL wasn't designed as a "relationally correct" language
    - In some respects, it is very lax
    - But easy to use, however
    - And, easy to misuse
      - So, using it well is difficult
- Sometimes, you will get results even if the SQL is wrong
  - SQL is not as strict as C or Java; it will not give you error messages if your table cannot fit the requirement of the theory
    - “Wrong results” without warnings

“Problems” in SQL

- SQL ≠ Relational Database
  - SQL wasn't designed as a "relationally correct" language
    - In some respects, it is very lax
    - But easy to use, however
    - And, easy to misuse
      - So, using it well is difficult
- Sometimes, you will get results even if the SQL is wrong
  - SQL is not as strict as C or Java; it will not give you error messages if your table cannot fit the requirement of the theory
    - “Wrong results” without warnings

Be careful when designing your SQL queries

“Problems” in SQL

- Key propriety of relations (in Codd's original paper)

ALL ROWS ARE DISTINCT

  - This can be enforced for tables in SQL
    - (But you have to create your tables well)

  - But it is not enforced for query results in SQL
    - You must be extra-careful if the result of a query is the starting point for another query, which happens often. (i.e., combined queries)

Create Tables

- A comma-separated list of a column-name followed by spaces and a datatype specifies the columns in the table

Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

Example

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

Create Tables

- Table names
  - Case-insensitive (Usually, by default)
    - But, in some database systems, the case sensitivity is quite different
    - Try, or find the reference for the specific database system

```
CREATE TABLE tablename
CREATE TABLE TABLENAME
CREATE TABLE tABleNamE

/* Same table names */
```

For example, MariaDB is system-dependent with respect to case sensitivity

`https://mariadb.com/kb/en/identifier-case-sensitivity/`

Create Tables

- Table names
  - Case-insensitive (Usually, by default)
    - But, in some database systems, the case sensitivity is quite different
    - Try, or find the reference for the specific database system

For example, MariaDB is system-dependent with respect to case sensitivity

`https://mariadb.com/kb/en/identifier-case-sensitivity/`

```
CREATE TABLE tablename
CREATE TABLE TABLENAME
CREATE TABLE tABleNamE

/* Same table names */
```

Actually, keywords are case-insensitive as well

```
create table tablename
CREATE table TABLENAME
CReaTE tABle tABleNamE

/* Same keywords */
```

Create Tables

- Table names
  - Case-insensitive (Usually, by default)
    - But, in some database systems, the case sensitivity is quite different
    - Try, or find the reference for the specific database system
- Naming Convention
  - Underscores as word separators (instead of CamelCase in Java)

```
CREATE TABLE tablename
CREATE TABLE TABLENAME
CREATE TABLE tABleNamE

/* Same table names */
```

Actually, keywords are case-insensitive as well

```
create table tablename
CREATE table TABLENAME
CReaTE tABle tABleNamE

/* Same keywords */
```

Create Tables

- Table names
  - Case-insensitive (Usually, by default)
    - But, in some database systems, the case sensitivity is quite different
    - Try, or find the reference for the specific database system
- Naming Convention
  - Underscores as word separators (instead of CamelCase in Java)
- Be careful with double quotes
  - … which represents a “case-sensitive” name

```
CREATE TABLE tablename
CREATE TABLE TABLENAME
CREATE TABLE tABleNamE

/* Same table names */
```

Actually, keywords are case-insensitive as well

```
create table tablename
CREATE table TABLENAME
CReaTE tABle tABleNamE

/* Same keywords */
```

Create Tables

- An SQL relation is defined using the create table command:

```
create table r (
    A1 D1, A2 D2, ..., An Dn,
    (integrity-constraint1),
    ...,
    (integrity-constraintk)
)
```

  - `r` is the name of the relation
  - each `Ai` is an attribute name in the schema of relation `r`
  - `Di` is the data type of values in the domain of attribute `Ai`

Data Types

- Text data types
  - `char(length)` -- fixed-length strings
  - `varchar(max length)` -- non-fixed-length text
  - `varchar2(max length)` ORACLE -- Oracle's transformation of `varchar`
  - `clob` -- very long text (like GB-level text)
    - Or, `text` MySQL

Data Types

- Numerical types
  - `int` -- Integer (a finite subset of the integers that is machine-dependent)
  - `float(n)` -- Floating point number, with user-specified precision of at least *n* digits
  - `real` -- Floating point and double-precision floating point numbers, with machine-dependent precision
  - `numeric(p, d)`
    - Fixed point number, with user-specified precision of `p` digits, with `d` digits to the right of decimal point
    - E.g., `numeric(3,1)`, allows 44.5 to be stores exactly, but not 444.5 or 0.32
    - In SQL Server, it is also called `decimal`

Data Types

- Date types
  - `date` -- YYYY-MM-DD
  - `datetime` -- YYYY-MM-DD HH:mm:SS

  - `timestamp` -- YYYY-MM-DD HH:mm:SS
    - But it is in the UNIX timestamp format

    - Value range (if stored in the signed 32-bit integer format):
      From *1970-01-01* 00:00:01 UTC to *2038-01-19* 03:14:07 UTC

    - More reading about the “Year 2038 Problem” of the `timestamp` data type:
      `https://en.wikipedia.org/wiki/Year_2038_problem`

Data Types

- Binary data types
  - `raw(max length)`
  - `varbinary(max length)`
  - `blob` -- binary large object
  - `bytea` -- used in PostgreSQL

Constraints

- Can you find any problem in this statement?

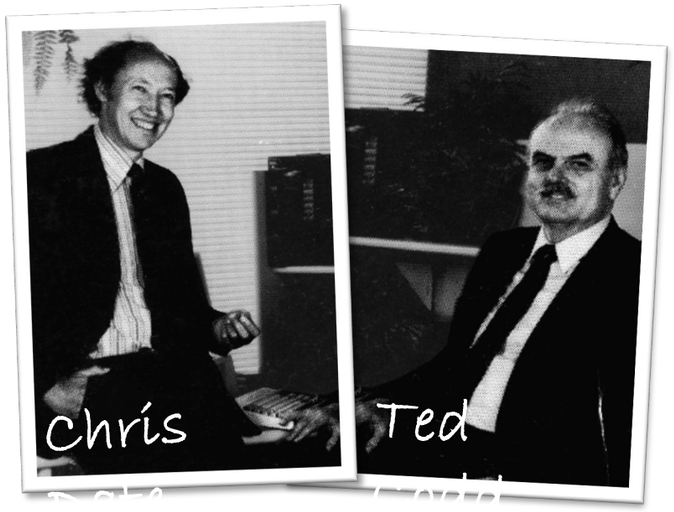
```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

Constraints

- Can you find any problem in this statement?
  - It is valid and can be accepted by most DBMS
  - But it does nothing to enforce that we have a valid “relation” in Codd’s sense

```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

Constraints



- Ted Codd and Chris Date
  - Worked on improving the relational theory
- Chris Date's work
  - Ensuring that only correct data that fits the theory can enter the database
  - Data inside the database remains correct
    - No need to double check for application programs

Constraints **are declarative rules that the DBMS will check every time new data will be added, when data is changed, or even when data is deleted, in order to prevent any inconsistency.**

* Any operation that violates a constraint fails and returns an error.

Constraints: Not NULL

- NULL values

```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

- We don't want someone with no ID and name

```
create table people (
    peopleid int not null,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4),
    died numeric(4)
)
```

- Use `not null` to indicate that these columns are mandatory

Constraints: Not NULL

- NULL values

```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

```
create table people (
    peopleid int not null,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4),
    died numeric(4)
)
```

- We don't want someone with no ID and name
- Use `not null` to indicate that these columns are mandatory

We can still have rows that with `NULL` values in the columns of `born`, `died`, and `first_name`

Constraints: Not NULL

- NULL values

```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

```
create table people (
    peopleid int not null,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4),
    died numeric(4)
)
```

Why is only surname mandatory?

- It depends on the requirement. In this movie database case, some actors may be known as their stage names instead of real names. (Lady Gaga vs. Stefani Joanne Angelina Germanotta)

Takeaway: design your table according to the requirements

Constraints: Not NULL

- NULL values

```
create table people (
    peopleid int,
    first_name varchar(30),
    surname varchar(30),
    born numeric(4),
    died numeric(4)
)
```

```
create table people (
    peopleid int not null,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4),
    died numeric(4)
)
```

Similar to the column `born`

- We can either accept that
  - A row is created before we have information of that person's birth date
  - Or we require that the information should be found before entering the data

Comments

```
/* Multi-line
comments */

-- Single line comments, similar to double back-slashes in Java and C++

// *Some DBMS also support double back-slashes, like SQL Server
```

Comments

```
create table people (
    peopleid int not null,
    first_name varchar(30),
    surname varchar(30) not null,  -- the actual surname or the stage name
    born numeric(4) not null,  -- the birth date is mandatorybefore entering the data
    died numeric(4)
)
```

- Add comments to the definition of tables

Constraints: Unique

- So far, nothing would prevent us from entering two same rows with different IDs

peopleid	first_name	surname	born	died
1	Alfred	Hitchcock	1899	1980
2	Alfred	Hitchcock	1899	1980
3	....	...	...	...

```
create table people (
    peopleid int not null
                    primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4)
)
```

Constraints: Unique

- A unique constraint (on a combination of multiple columns)

peopleid	first_name	surname	born	died
1	Alfred	Hitchcock	1899	1980
2	Alfred	Hitchcock	1899	1980
3	....	...	...	...

```
create table people (
    peopleid int not null
                    primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4),
    unique (first_name, surname)
)
```

The combination of `(first_name, surname)` cannot be the same for any two rows

- But you still can have people with the same first name or surname, respectively

Constraints: Unique

- A unique constraint (on a single column)

peopleid	first_name	surname	born	died
1	Alfred	Hitchcock	1899	1980
2	Alfred	Hitchcock	1899	1980
3	....	...	...	...

```
create table people (
    peopleid int not null
                    primary key,
    first_name varchar(30) unique,
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4)
)
```

No identical first names for any two people here

- But it is not what we want in this table

Constraints: Primary Key

- The main key for the table, and indicates two things:
  - the value is mandatory
  - that the values are unique (no duplicates allowed in the column)

```
create table people (
    peopleid int not null
                primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4)
)
```

Constraints: Primary Key

- The main key for the table, and indicates two things:
  - the value is mandatory (i.e., `not null`)
  - that the values are unique (no duplicates allowed in the column, i.e., `unique`)

```
create table people (
    peopleid int not null
                 primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4)
)
```

`primary key` implies `not null`, so `not null` here is redundant (but doesn't hurt)

Constraints: Check

- A column must satisfy a certain boolean expression test
  - The most generic constriant type

You must ensure that the person died after born

A useful trick to standardize names

- Such that there won't be rows with the same name of "Alfred Hitchcock", "ALFRED HITCHCOCK", and "alfred hitchcock".
- `upper(string)` is a function in PostgreSQL

```
create table people (
    peopleid int not null
                 primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4),
    unique (first_name, surname),
    check (died - born >= 0),
    check (first_name = upper(first_name)),
    check (surname = upper(surname))
)
```

Named Constraints

- A name can be assigned to the constraints
  - … in order to refer to them easier in some other operations
    - PostgreSQL will give a name to the constraints if you don’t assign a name explicitly

```
create table people (
    peopleid int not null
                primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4),
    unique (first_name, surname),
    check (died - born >= 0),
    check (first_name = upper(first_name)),
    check (surname = upper(surname))
)
```

```
create table people (
    peopleid int not null
                primary key,
    first_name varchar(30),
    surname varchar(30) not null,
    born numeric(4) not null,
    died numeric(4),
    unique (first_name, surname),
    constraint validate_birthdate check (died - born >= 0),
    constraint standardize_first_name check (first_name = upper(first_name)),
    constraint standardize_surname check (surname = upper(surname))
)
```

Referential Integrity

- Check constaints are static
  - Once it is written into the table, the criteria cannot be updated automatically

```
create table movies (
    movieid int not null primary key,
    title varchar (60) not null,
    country char (2) not null,
    year_released numeric(4) not null
    check (year_released >= 1895),
    unique (title, country, year_released)
)
```

- It is very difficult to perform static checks
  - Too many countries; country names and codes may change

Referential Integrity

- Check constaints are static
  - Once it is written into the table, the criteria cannot be updated automatically

```
create table movies (
    movieid int not null primary key,
    title varchar (60) not null,
    country char (2) not null,
    year_released numeric(4) not null
    check (year_released >= 1895),
    unique (title, country, year_released)
)
```

country_code	country_name	continent
US	United States	AMERICA
CN	China	ASIA
RU	Russia	EUROPE

Referential Integrity

- The `country` column in `movies` should be linked with the `country_code` column in another table (called reference table)

Foreign Key

- Format:
  - `foreign key (Am, ..., An ) references r`

```
create table movies (
    movieid int not null primary key,
    title varchar (60) not null,
    country char (2) not null,
    year_released numeric(4) not null
    check (year_released >= 1895),
    unique (title, country, year_released),
    foreign key (country) references country_list (country_code)
)
```

Meaning of this foreign key:

- The `country` column in this table (movies) refers to the `country_code` column in the table called `country_list`

Tip:

- `country_code` should be a key (primary key or unique) in the table `country_list`

Foreign Key

- Format:
  - `foreign key (Am, ..., An ) references r`

Movie ID	Movie Title	Country	Year
0	Citizen Kane	US	1941
1	La règle du jeu	FR	1939
2	North By Northwest	US	1959
3	Singin' in the Rain	US	1952
4	Rear Window	US	1954

Movie Entities

Directed By

Movie ID	Director ID
0	2
1	5
2	1

Foreign keys required in this table

Director ID	Director_ Firstname	Director_ Lastname	Born	Died
1	Alfred	Hitchcock	1899	1980
2	Orson	Welles	1915	1985
3	....	...	...	...

Director Entities

Foreign Key

- However, in some cases, foreign keys can be a problem
  - Especially in big data processing applications
    - E.g., Alibaba Java Coding Guideline (阿里巴巴Java开发手册)

SQL Rules

6. **[Mandatory]** *Foreign key* and *cascade update* are not allowed. All foreign key related logic should be handled in application layer.

> Note:
>
> e.g. Student table has *student_id* as primary key, score table has *student_id* as foreign key. When *student.student_id* is updated, *score.student_id update* is also triggered, this is called a *cascading update*. *Foreign key* and *cascading update* are suitable for single machine, low parallel systems, not for distributed, high parallel cluster systems. *Cascading updates* are strong blocked, as it may lead to a DB update storm. *Foreign key* affects DB insertion efficiency.

https://github.com/alibaba/p3c
https://alibaba.github.io/Alibaba-Java-Coding-Guidelines/#sql-rules

Foreign Key

- However, in some cases, foreign keys can be a problem
  - Especially in big data processing applications
    - E.g., Alibaba Java Coding Guideline (阿里巴巴Java开发手册)

(三) SQL 语句

6. 【强制】不得使用外键与级联，一切外键概念必须在应用层解决。

   说明：（概念解释）学生表中的 student_id 是主键，那么成绩表中的 student_id 则为外键。如果更新学生表中的 student_id，同时触发成绩表中的 student_id 更新，即为级联更新。外键与级联更新适用于单机低并发，不适合分布式、高并发集群；级联更新是强阻塞，存在数据库更新风暴的风险；外键影响数据库的插入速度。

https://github.com/alibaba/p3c
https://alibaba.github.io/Alibaba-Java-Coding-Guidelines/#sql-rules

Summary: How to Create Tables

- Creating tables requires:
  - Proper modelling
  - Defining keys
  - Determining correct data types
  - Defining constraints

- Boring, but important
  - No further checks in the application programs; most things are ensured in the database layer

Updates to Tables

- Insert
  - `insert into instructor values ('10211', 'Smith', 'Biology', 66000);`
- Delete
  - Remove all tuples from the student relation
    - `delete from movies`

Updates to Tables

- Drop Table
  - `drop table r`
- Alter
  - `alter table r add A D`
    - where A is the name of the attribute (column) to be added to relation `r` and D is the data type of A.
    - All existing tuples in the relation are assigned null as the value for the new attribute.
  - `alter table r drop column A`
    - where A is the name of an attribute of relation r
    - Dropping of attributes not supported by many databases.
    - (There are some other things that can be “dropped”, including checks, foreign keys, indexes, etc.)

Updates to Tables

- More about insert

```
create table lab (
    id serial primary key,
    address varchar(20) not null,
    time varchar(20) not null,
    capacity int,
    teacher varchar(20),
    unique (address,time)
);
```

Values must match column names one by one

```
insert into lab (address, time, capacity, teacher) values ('402','2-78',36,'yueming');

insert into lab (address, time, teacher) values ('402','2-78','yueming');

insert into lab (address, time, teacher) values ('408','2-78','o''relly');
```

Updates to Tables

- More about insert

```
create table lab (
    id serial primary key,
    address varchar(20) not null,
    time varchar(20) not null,
    capacity int,
    teacher varchar(20),
    unique (address,time)
);


insert into lab (address, time, capacity, teacher) values ('402','2-78',36,'yueming');

insert into lab (address, time, teacher) values ('402','2-78','yueming');

insert into lab (address, time, teacher) values ('408','2-78','o''relly');
```

Missing columns and values for "nullable" columns are allowed

- … and a NULL willl be inserted

* But if you miss a mandatory column (such as address), an error will occur.

Updates to Tables

- More about insert

```
create table lab (
    id serial primary key,
    address varchar(20) not null,
    time varchar(20) not null,
    capacity int,
    teacher varchar(20),
    unique (address,time)
);


insert into lab (address, time, capacity, teacher) values ('402','2-78',36,'yueming');

insert into lab (address, time, teacher) values ('402','2-78','yueming');

insert into lab (address, time, teacher) values ('408','2-78','o''relly');
```

* Use two single quotes to represent a single quote in the content (i.e., escape charactor)

Basic SQL

Select

- `select * from` [tablename]
  - The `select` clause lists the attributes desired in the result of a query
  - To display the full content of a table, you can use select *
    - * : all columns

```
select A1, A2, ..., An
 from r1, r2, ..., rm
       where P
```

Select

- `select * from` [tablename]
  - The `select` clause lists the attributes desired in the result of a query
  - To display the full content of a table, you can use select *
    - * : all columns

```
select A1, A2, ..., An
 from r1, r2, ..., rm
       where P
```

- Such a query is frequently used in interactive tools (especially when you don't remember column names ...)
  - But you should not use it, though, in application programs

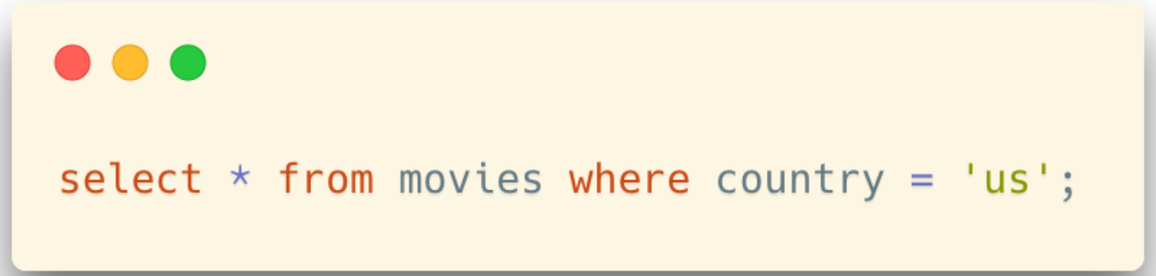
Restrictions

- When tables contains thousands or millions or billions of rows, you are usually interested in only a small subset, and only want to return some of the rows

peopleid	first_name	surname	born	died

Restrictions

- Filtering
  - Performed in the “`where`” clause
  - Conditions are usually expressed by a column name
    - … followed by a comparison operator and the value to which the content of the column is compared
  - Only rows for which the condition is true will be returned



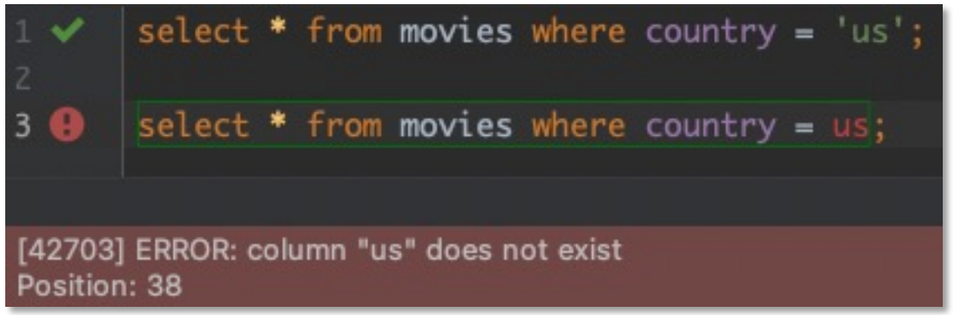
```
select * from movies where country = 'us';
```

Comparison

- You can compare to:
  - a number
  - a string constant
  - another column (from the same table or another, we'll see queries involving several tables later)
  - the result of a function (we'll see them soon)

String Constants

- Be aware that string constants must be quoted between single-quotes
  - If they aren't quoted, they will be interpreted as column names
  - * Same thing with Oracle if they are double-quoted



Filtering

- Note that a filtering condition returns a subset
  - If you return all the columns from a table without duplicates, it won't contain duplicates either and will be a valid "relation"

```
select country from movies;
```

	country
1	ru
2	eg
3	ma
4	ar
5	in
6	in
7	pk
8	dk
9	jp
10	eg
11	us
12	ca
13	ru
14	be
15	br
16	my
17	cn
18	de

Select without From or Where

- An attribute can be a literal with no from clause

```
select '437'
```

  - Results is a table with one column and a single row with value “437”
  - Can give the column a name using:

```
select '437' as FOO
```

- An attribute can be a literal with from clause

```
select 'A' from movies
```

  - Result is a table with one column and N rows (number of tuples in the movies table), each row with value “A”

Select without From or Where

- An attribute can be a literal with no from clause

```
select '437'
```

A common way to test expressions

  - Results is a table with one column and a single row with value “437”
  - Can give the column a name using:

```
select '437' as FOO
```

- An attribute can be a literal with from clause

```
select 'A’ from movies
```

  - Result is a table with one column and N rows (number of tuples in the movies table), each row with value “A”

Arithmetic Expression

- The `select` clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples

	runtime
1	161
2	102
3	90
4	94
5	130
6	159
7	<null>
8	102
9	108
10	<null>
11	106
12	<null>
13	100
14	95
15	<null>

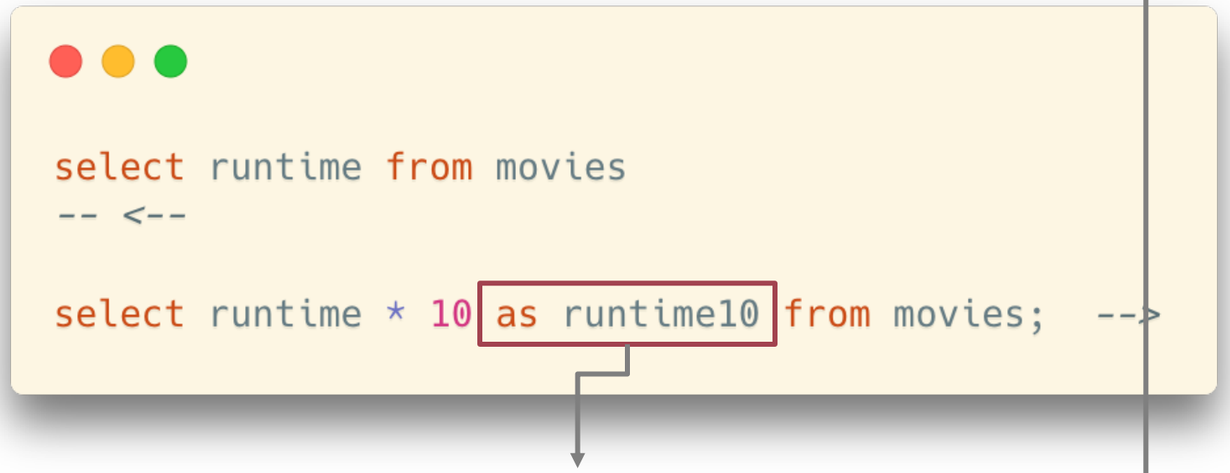
```
select runtime from movies
-- <--

select runtime * 10 as runtime10 from movies;  -->
```

	runtime10
1	1610
2	1020
3	900
4	940
5	1300
6	1590
7	<null>
8	1020
9	1080
10	<null>
11	1060
12	<null>
13	1000
14	950
15	<null>

Arithmetic Expression

- The `select` clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples

	runtime
1	161
2	102
3	90
4	94
5	130
6	159
7	<null>
8	102
9	108
10	<null>
11	106
12	<null>
13	100
14	95
15	<null>



	runtime10
1	1610
2	1020
3	900
4	940
5	1300
6	1590
7	<null>
8	1020
9	1080
10	<null>
11	1060
12	<null>
13	1000
14	950
15	<null>

Logical Connectives

- and, or, not
  - Just like in programming languages
  - All logical operators have different precedence
    - For example, and is “stronger” than or

Table 1-1. Operator Precedence (decreasing)

Operator/Element	Associativity	Description
::	left	PostgreSQL-style typecast
[ ]	left	array element selection
.	left	table/column name separator
-	right	unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
IS		test for TRUE, FALSE, UNKNOWN, NULL
ISNULL		test for NULL
NOTNULL		test for NOT NULL
(any other)	left	all other native and user-defined operators
IN		set membership
BETWEEN		containment
OVERLAPS		time interval overlap
LIKE ILIKE		string pattern matching
< >		less than, greater than
=	right	equality, assignment
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

https://www.postgresql.org/docs/7.2/sql-precedence.html

Logical Connectives

- and, or, not
  - Just like in programming languages
  - All logical operators have different precedence
    - For example, and is “stronger” than or.

```
select * from movies
where (country = 'us' or country = 'gb') and (year_released between 1940 and 1949);
```

```
select * from movies
where country = 'us' or country = 'gb' and year_released between 1940 and 1949;
```

Differences?

Logical Connectives

- Use parentheses to specify that the “or” should be evaluated before the “and”, and that the conditions filter
  - 1) British or American films
  - 2) that were released in the 1940s

```
select * from movies
where (country = 'us' or country = 'gb') and (year_released between 1940 and 1949);
```


```
select * from movies
where country = 'us' or country = 'gb' and year_released between 1940 and 1949;
```

Logical Connectives

- Question:
  - Find the Chinese movies from the 1940s and American movies from the 1950s

Logical Connectives

- Question:
  - Find the Chinese movies from the 1940s and American movies from the 1950s

```
select * from movies
where (country = 'cn'
  and year_released between 1940 and 1949)
or (country = 'us'
  and year_released between 1950 and 1959)
```

In this case, parentheses are optional – but they don't hurt

- The parentheses make the statement easier to understand

Logical Connectives

- The operands of the logical connectives can be expressions involving the comparison operators `<`, `<=`, `>`, `>=`, `=`, and `<>`.
  - Note that there are two ways to write “not equal to”: `!=` and `<>`
  - Comparisons can be applied to results of arithmetic expressions
- Beware that "bigger" and "smaller" have a meaning that depends on the data type
  - It can be tricky because most products implicitly convert one of the sides in a comparison between values of differing types

```
2 < 10       -- true
'2' < '10'   -- false

'2-JUN-1883'>'1-DEC-2056'  -- single-quoted, treated as strings but not dates
```

Logical Connectives

- `in()`
  - It can be used as the equivalent for a series of equalities with `or`
  - It may make a comparison clearer than a parenthesized expression
  - * Some advanced features of `in()` will be introduced when learning subqueries

```
where (country = 'us' or country = 'gb')
  and year_released between 1940 and 1949

where country in ('us', 'gb')
  and year_released between 1940 and 1949
```

Logical Connectives

- Negation
  - All comparisons can be negated with `not`

```
-- exclude all movies selected in the previous page

where not ((country in ('us', 'gb')) and (year_released between 1940 and 1949))
where (country not in ('us', 'gb')) or (year_released not between 1940 and 1949)  -- equivalent query
```

`between` Comparison Operator

- `between ... and ...`
  - shorthand for: >= and <=

```
year_released between 1940 and 1949

-- It's shorthand for this:
year_released >= 1940 and year_released <= 1949
```

`between` Comparison Operator

- `between ... and ...`
  - shorthand for: >= and <=

```
year_released between 1940 and 1949

-- It's shorthand for this:
year_released >= 1940 and year_released <= 1949
```

not “<”

like

- For strings, you also have like `which` is a kind of regex (regular expression) for dummies.
- `like` compares a string to a pattern that can contain two wildcard characters:
  - `%` meaning "any number of characters, including none”
  - `_` meaning "one and only one character"

like

```sql
select * from movies where title not like '%A%' and title not like '%a%';

select * from movies where upper(title) not like '%A%';
-- not recommended due to the performance cost of upper()
```

- This expression for instance returns films the title of which doesn't contain any A
  - This A might be the first or last character as well
  - Note that if the DBMS is case sensitive, you need to cater both for upper and lower case
  - Function calls could slow down queries; use with caution

Date

- Date formats
  - Beware also of date formats, and of conflicting European/American formats which can be ambiguous for some dates. Common problem in multinational companies.

```
DD/MM/YYYY
MM/DD/YYYY
YYYY/MM/DD
```

Date

```
select * from forum_posts where post_date >= '2018-03-12';
select * from forum_posts where post_date >= date('2018-03-12');
select * from forum_posts where post_date >= date('12 March, 2018');
```

- Whenever you are comparing data of slightly different types, you should use functions that "cast" data types
  - It will avoid bad surprises
  - The functions don't always bear the same names but exist with all products
- Default formats vary by product, and can often be changed at the DBMS level
  - So, better to use explicit date types and functions other than strings
  - Conversely, you can format something that is internally stored as a date and turn it into a character string that has almost any format you want

Date and Datetime

- If you compare datetime values to a date (without any time component) the SQL engine will not understand that the date part of the datetime should be equal to that date
  - Rather, it will consider that the date that you have supplied is actually a datetime, with the time component that you can read below
    - `date('2020-03-20')` is equal to `datetime('2020-03-20 00:00:00')`
- Date functions
  - Many useful date functions when manipulating date and datetime values
  - However, most of them are DBMS-dependent

```
select date_eq_timestamp(date('2018-03-12'), date('2018-02-12') + interval '1 month');  -- true
```

NULL

- In a language such as Java, you can compare a reference to `null`, because `null` is defined as the ‘0’ address.
  - In C, you can also compare a pointer to `NULL` (pointer is C-speak for reference)

NULL

- Not in SQL, where `NULL` denotes that a value is missing
  - `Null` in SQL is not a value
    - … and if it's not a value, hard to say if a condition is true.
    - A lot of people talk about "null values", but they have it wrong
  - Most expression with `NULL` is evaluated to `NULL`

```sql
select * from movies where runtime is null;

select * from movies where runtime = null;  -- warning in DataGrip; not the same as "is null"
```

Some Functions

- Show DDL of a table

```
desc movies;  -- Oracle, MySQL

describe table movies  -- IBM DB2

\d movies  -- PostgreSQL

.schema movies  -- SQLite
```

Some Functions – Compute and Derive

- One important feature of SQL is that you don't need to return data exactly as it was stored
  - Operators, and many (*mostly DBMS specific*) functions allow to return transformed data

Some Functions

- A simple transformation is concatenating two strings together
  - Most products use || (two vertical bars) to indicate string concatenation
  - SQL Server, though, uses +, and MySQL a special `concat()` function that also exists in some other products

```
select title
       || ' was released in '
       || year_released movie_release
from movies
where country = 'us';
```

	movie_release
1	Variety was released in 2008
2	Inglourious Basterds was released in 2009
3	La grande vadrouille was released in 1966
4	Pulp Fiction was released in 1994
5	Schindler's List was released in 1993
6	Star Wars was released in 1977
7	The Dark Knight was released in 2008
8	The Godfather was released in 1972
9	The Shawshank Redemption was released in 1994
10	Titanic was released in 1997
11	Charade was released in 1963
12	North by Northwest was released in 1959
13	Singin' in the Rain was released in 1952
14	Rear Window was released in 1954
15	City Lights was released in 1931

Some Functions

- A simple transformation is concatenating two strings together
  - Most products use || (two vertical bars) to indicate string concatenation
  - SQL Server, though, uses +, and MySQL a special `concat()` function that also exists in some other products

```
select title
       || ' was released in '
       || year_released movie_release
from movies
where country = 'us';
```

	movie_release
1	Variety was released in 2008
2	Inglourious Basterds was released in 2009
3	La grande vadrouille was released in 1966
4	Pulp Fiction was released in 1994
5	Schindler's List was released in 1993
6	Star Wars was released in 1977
7	The Dark Knight was released in 2008
8	The Godfather was released in 1972
9	The Shawshank Redemption was released in 1994
10	Titanic was released in 1997
11	Charade was released in 1963
12	North by Northwest was released in 1959
13	Singin' in the Rain was released in 1952
14	Rear Window was released in 1954
15	City Lights was released in 1931

Note that you can give a name to an expression

- This will be used as column header
- It also becomes a "virtual column" if you turn the query into a "virtual table"

Some Functions

- A simple transformation is concatenating two strings together
  - Most products use || (two vertical bars) to indicate string concatenation
  - SQL Server, though, uses +, and MySQL a special `concat()` function that also exists in some other products

```
select title
       || ' was released in '
       || year_released movie_release
from movies
where country = 'us';
```

Although YEAR_RELEASED is actually a number, it's implicitly turned into a string by the DBMS.

- In that case it's not a big issue, but it would be better to use a function to convert explicitly.

```
select title
       || ' was released in '
       || cast(year_released as varchar) movie_release
from movies
where country = 'us';
```

Some Functions

- When to use functions
  - An example of showing a result that isn't stored as such is computing an age
    - You should never store an age; it changes all the time!
    - If you want to display the age of people who are alive, you must compute their age by subtracting the year when they were born from the current year.

Some Functions

- When to use functions
  - An example of showing a result that isn't stored as such is computing an age
    - You should never store an age; it changes all the time!
    - If you want to display the age of people who are alive, you must compute their age by subtracting the year when they were born from the current year.

  - In the table people:
    - Alive – died is `null`
    - Age: `<this year> - born`

```
select peopleid, surname,
       date_part('year', now()) - born as age
from people
where died is null;
```

7	7	Caroline	Aaron	1952	<null>	F
8	8	Quinton	Aaron	1984	<null>	M
9	9	Dodo	Abashidze	1924	1990	M

Some Functions

- Numerical functions

```
round(3.141592, 3)  -- 3.142
trunc(3.141592, 3)  -- 3.141
```

- More string functions

```
upper('Citizen Kane')
lower('Citizen Kane')
substr('Citizen Kane', 5, 3)  -- 'zen'
trim('   Oops  ')  -- 'Oops'
replace('Sheep', 'ee', 'i')  -- 'Ship'
```

Some Functions

- Type casting
  - `cast(column as type)`

```
select cast(born as char)||'abc' from people;
select cast(born as char(2)) ||'abc' from people;
select cast(born as char(10)) ||'abc' from people;
select cast(born as varchar) ||'abc' from people;
select cast(born as varchar(2)) ||'abc' from people;
```

Case

- A very useful construct is the CASE … END construct that is similar to IF or SWITCH statements in a program

```
CASE input_expression
    WHEN when_expression THEN result_expression
    [ ...n ]
    [ELSE else_result_expression]
END
```

```
CASE
    WHEN Boolean_expression THEN result_expression
    [ ...n ]
    [ELSE else_result_expression]
END
```

Case

- Example 1: Show the corresponding words of the gender abbreviations

```
select peopleid, surname,
    case gender
        when 'M' then 'male'
        when 'F' then 'female'
    end as gender_2
from people
where died is null;
```

*Similar to the switch-case statement in Java and C

Case

- Example 2: Decide whether someone’s age is older/younger than a pivot

Case

- Example 2: Decide whether someone's age is older/younger than a pivot

A horrible solution!

```
case age
    when 30 then 'younger than 44'
    when 31 then 'younger than 44'
    when 32 then 'younger than 44'
    when 33 then 'younger than 44'
    when 34 then 'younger than 44'
    when 35 then 'younger than 44'
    when 36 then 'younger than 44'
    ...
    when 43 then 'younger than 44'
    when 44 then '44 years old'
    when 45 then 'older than 44'
    ...
end as status
```

Case

- Example 2: Decide whether someone's age is older/younger than a pivot
  - CASE

```
select peopleid, surname,
    case (date_part('year', now()) - born > 44)
        when true then 'older than 44'
        when false then 'younger than 44'
        else '44 years old'
    end as status
from people
where died is null;
```

Case

- Example 2: Decide whether someone's age is older/younger than a pivot
  - CASE
  - CASE WHEN

```
select peopleid, surname,
    case
        when (date_part('year', now()) - born > 44) then 'older than 44'
        when (date_part('year', now()) - born < 44) then 'younger than 44'
        else '44 years old'
    end as status
from people
where died is null;
```

Case

- Example 2: Decide whether someone's age is older/younger than a pivot
  - CASE
  - CASE WHEN

```
select peopleid, surname,
    case
        when (date_part('year', now()) - born > 44) then 'older than 44'
        when (date_part('year', now()) - born < 44) then 'younger than 44'
        else '44 years old'
    end as status
from people
where died is null;
```

The ELSE branch

- Return a default value when all `when` criteria are not met
- If no else, `NULL` will be returned

Case

- About the `NULL` value
  - Use the “`is null`” criteria

```
select surname,
    case
        when died is null then 'alive and kicking'
        else 'passed away'
    end as status
from people
```

More on Retrieving Data

Distinct

Distinct

- No duplicated identifier
  - **Some rules** must be respected if you want to obtain valid results when you apply new operations to result sets
    - They must be mathematical sets, i.e., no duplicates

Distinct

- If we run a query such as the one below
  - Many identical rows
    - In other words, we may be obtaining a table, but it's not a relation because many rows cannot be distinguished

```
select country from movies
where year_released=2000;
```

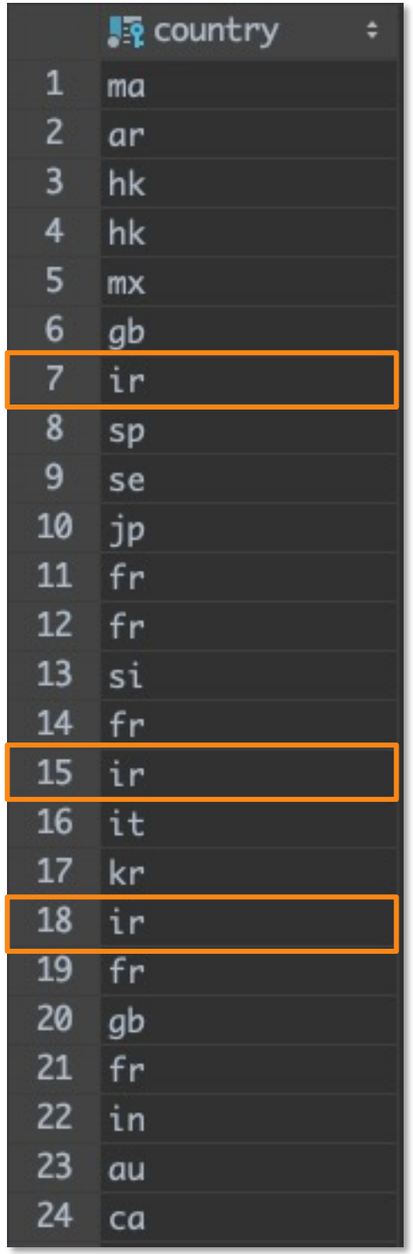
	country
1	ma
2	ar
3	hk
4	hk
5	mx
6	gb
7	ir
8	sp
9	se
10	jp
11	fr
12	fr
13	si
14	fr
15	ir
16	it
17	kr
18	ir
19	fr
20	gb
21	fr
22	in
23	au
24	ca

Duplicated country codes in the query result

- But their original rows are not considered duplicated tuples

Distinct

- The result of the query is in fact completely uninteresting
  - Whenever we are only interested in countries in table movies, it can only be for one of two reasons:
    - See a list of countries that have movies
    - Or, for instance, see which countries appear most often



	country
1	ma
2	ar
3	hk
4	hk
5	mx
6	gb
7	ir
8	sp
9	se
10	jp
11	fr
12	fr
13	si
14	fr
15	ir
16	it
17	kr
18	ir
19	fr
20	gb
21	fr
22	in
23	au
24	ca

Duplicated country codes in the query result

- But their original rows are not considered duplicated tuples

Distinct

- If we only are interested in the different countries, there is the special keyword `distinct`.

```
select distinct country
from movies
where year_released=2000;
```

23 rows

	country
1	si
2	mx
3	cn
4	sp
5	dk
6	gb
7	se
8	tw
9	ar
10	ca
11	pt
12	jp
13	us
14	kr
15	ma
16	de
17	au
18	in
19	hk
20	it
21	gr
22	ir
23	fr

No duplicated results in the country code list now

- All of them are different now, and hence it is a relation!

Distinct

- Multiple columns after the keyword `distinct`
  - It will eliminate those rows where all the selected fields are identical
  - The selected combination `(country, year_released)` will be identical

```
select distinct country, year_released
from movies
where year_released in (2000,2001);
```

44 rows

	country	year_released
1	nz	2001
2	ar	2001
3	mx	2000
4	kr	2001
5	in	2001
6	ma	2000
7	si	2000
8	ca	2001
9	uy	2001
10	pt	2001
11	fr	2000
12	de	2000
13	us	2001
14	au	2001
15	au	2000
16	hu	2001
17	ie	2001
18	sp	2000
19	in	2000
20	us	2000
21	nl	2001
22	hk	2001
23	tw	2000

Lab Session (Week 2)

- We will have more examples for manipulating a table and inserting data into tables

Please remember to upload your "*Undergraduate Students Declaration Form*" to the correct place