

RELATIONAL DATABASES

1. Basics of relational databases

Carmelo Vaccaro

University of Paris-Saclay

Master 1 - AI
2022/23: first semester

The content of these slides is taken from the book
Database Systems - The Complete Book,
by Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom,
published by Pearson, 2014.

1. The Worlds of Database Systems

Databases today 1/2

Databases today are essential to every business. Whenever you visit a web site there is a database behind the scenes serving up the information you request.

Corporations maintain all their important records in databases. Databases are likewise found at the core of many scientific investigations.

The power of databases comes from a body of knowledge and technology that has developed over several decades and is embodied in specialized software called a **database management system**, or **DBMS**.

A DBMS is a powerful tool for creating and managing large amounts of data efficiently and allowing it to persist over long periods of time, safely. These systems are among the most complex types of software available.

What is a database?

A **database** is a collection of information that exists over a long period of time and that is managed by a database management system.

What a database management system is expected to do

A database management system is expected to:

- 1 Allow users to create new databases and specify their **schemas** (logical structure of the data), using a specialized **data-definition language**.
- 2 Allow users to **query** and modify the data, using an appropriate language.
- 3 Support the storage of very large amounts of data over a long period of time, allowing efficient access to the data for queries and database modifications.
- 4 Enable **durability**, the recovery of the database in the face of failures, errors of many kinds, or intentional misuse.
- 5 Control access to data from many users at once and ensuring **isolation** (not allowing unwanted interactions among users) and **atomicity** (no actions on the data to be performed partially but not completely).

File systems: before the appearance of database management systems 1/2

Before the appearance of database management systems data were managed using file systems.

File systems provide some of item (3) above: they store data over a long period of time, and they allow the storage of large amounts of data.

File systems: before the appearance of database management systems 2/2

File systems do not guarantee the other parts of (3) and the whole (1), (2), (4), (5).

Indeed with file systems:

- a schema for the data is limited to the creation of directory structures for files (1);
- a query language for the data in files is not supported (2);
- access to data items whose location in a particular file is not known is not efficient (3);
- data can be lost if not been backed up (4);
- concurrent access to files by several users or processes is allowed but modifications of the same file at about the same time by two users or processes is not prevented (5).

Enter database management systems

The first commercial database management systems appeared in the late 1960's.

Examples:

- Banking systems: maintaining accounts and making sure that system failures do not cause money to disappear.
- Airline reservation systems: ensure that data will not be lost and accept very large volumes of small actions by customers.
- Corporate record keeping: employment and tax records, inventories, sales records, and a great variety of other types of information, much of it critical.

Early database management systems

The early DBMS's required the programmer to visualize data much as it was stored.

These database systems used several different data models for describing the structure of the information in a database, like for example the “hierarchical” or tree-based model and the graph-based “network” model.

A problem with these early models and systems was that they did not support high-level query languages.

Even very simple queries required considerable effort.

Enter relational database systems

Following a famous paper written by Ted Codd in 1970, database systems changed significantly.

Codd proposed that database systems should present the user with a view of data organized as tables called **relations**.

Behind the scenes, there is a complex data structure that allows rapid response to a variety of queries. But, unlike the programmers for earlier database systems, the programmer of a relational system would not be concerned with the storage structure.

Queries could be expressed in a very high-level language (like SQL), which greatly increased the efficiency of database programmers.

Beyond relational database systems






















By 1990, relational database systems were the norm. Yet the database field has continued to evolve, and new issues and approaches to the management of data surface regularly.

Thanks to the rise in object-oriented programming, programmers and designers began to treat the data in their databases as objects.

This allows for relations between data to be relations to objects and their attributes and not to individual fields.

Since 2000's database management systems not based on the relational model (NoSQL) have become popular.

Ranking of most popular DBMS

Rank			DBMS	Database Model	Score		
Sep 2022	Aug 2022	Sep 2021			Sep 2022	Aug 2022	Sep 2021
1.	1.	1.	Oracle 	Relational, Multi-model 	1238.25	-22.54	-33.29
2.	2.	2.	MySQL 	Relational, Multi-model 	1212.47	+9.61	-0.06
3.	3.	3.	Microsoft SQL Server 	Relational, Multi-model 	926.30	-18.66	-44.55
4.	4.	4.	PostgreSQL 	Relational, Multi-model 	620.46	+2.46	+42.95
5.	5.	5.	MongoDB 	Document, Multi-model 	489.64	+11.97	-6.87
6.	6.	6.	Redis 	Key-value, Multi-model 	181.47	+5.08	+9.53
7.	 8.	 8.	Elasticsearch	Search engine, Multi-model 	151.44	-3.64	-8.80
8.	 7.	 7.	IBM Db2	Relational, Multi-model 	151.39	-5.83	-15.16
9.	9.	 11.	Microsoft Access	Relational	140.03	-6.47	+23.09
10.	10.	 9.	SQLite 	Relational	138.82	-0.05	+10.17

Source: <https://db-engines.com/en/ranking>

2. Basics of the relational model

The relational model

The relational model represents data as a two-dimensional table called a relation.

The next is an example of a relation, which we shall call *Movies*. The rows each represent a movie, and the columns each represent a property of movies.

Figure: The relation *Movies*

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

Attributes

The columns of a relation are named *attributes*.

In the table *Movies* the attributes are *title*, *year*, *length*, and *genre*.
Attributes appear at the tops of the columns.

Usually, an attribute describes the meaning of entries in the column below.
For instance, the column with attribute *length* holds the length, in minutes, of each movie.

The *schema* for a relation consists of the name of the relation and its set of attributes.

Convention: we show the schema for the relation with the relation name followed by a parenthesized list of its attributes.

Thus, the schema for the relation *Movies* seen before is

Movies(title, year, length, genre)

Remark on the order of attributes

The attributes in a relation schema are a set, not a list, so their order is unimportant.

However, sometimes we will define a “standard” order for the attributes and use the same order in every place where the relation appears.

Database and database schema

In the relational model, a *database* consists of one or more relations.

The *database schema* of a database is the set of schemas for the relations of the database.

Tuples

The rows of a relation, other than the header row containing the attribute names, are called *tuples*.

A tuple has one component for each attribute of the relation.

For instance, the first of the tuples in the figure below has the four components *Gone With the Wind*, 1939, 231, and *drama* for attributes *title*, *year*, *length*, and *genre*, respectively.

Figure: The relation Movies

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

Relation instances

A set of tuples for a given relation is called an *instance* of that relation.

The relational model requires that each component of each tuple be *atomic*, that is, it must be of some elementary type such as integer or string.

It is not permitted for a value to be a record structure, set, list, array, or any other type that reasonably can have its values broken into smaller components.

Associated with each attribute of a relation is a *domain*, that is, a particular elementary type. The components of any tuple of the relation must have, in each component, a value that belongs to the domain of the corresponding column.

For example, tuples of the *Movies* relation must have a first component that is a string, second and third components that are integers, and a fourth component whose value is a string.

It is possible to include the domain, or data type, for each attribute in a relation schema like the following example:

Movies(title:string, year:integer, length:integer, genre:string)

Remark on the order of tuples

Relations are multisets of tuples, not lists of tuples, so the order in which the tuples of a relation are presented is immaterial.

Changes in a database

Databases are generally not static and change over time. Normally new tuples are inserted, existing tuples are changed or removed.

The schema of a relation can also be changed, for example by changing the name of attributes or adding or deleting attributes.

However adding or deleting attributes can be very expensive in a real database system because each of perhaps millions of tuples needs to be rewritten to add or delete components.

Also, if we add an attribute, it may be difficult or even impossible to generate appropriate values for the new component in the existing tuples.

Key of a relation

A set of attributes forms a *key* for a relation if we do not allow two tuples in a relation instance to have the same values in all the attributes of the key.

A key is therefore a constraint on the database.

Key of a relation: example

We can declare that the relation *Movies* has a key consisting of the two attributes *title* and *year*. That is, we assume that there are not two different movies having the same title and the same year.

The title by itself does not form a key, since sometimes “remakes” of a movie appear. For example, there are three movies named *King Kong*, each made in a different year. The year alone too is not a key since there are usually many movies made in the same year.

Key of a relation: notation

We indicate the attribute or attributes that form a key for a relation by underlining the key attribute(s). For instance, the *Movies* relation could have its schema written as:

Movies(title, year, length, genre)

Key of a relation: remark 1

The statement that a set of attributes forms a key for a relation is a statement about all possible instances of the relation, not a statement about a single instance, not even about all tuples present in the database.

For example, looking only at the tiny relation of the figure below we might imagine that genre by itself forms a key, since we do not see two tuples that agree on the value of their genre components. However, we can easily imagine that if the relation instance contained more movies, there would be many dramas, many comedies, and so on.

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

Key of a relation: remark 2

Many real-world databases use artificial keys doubting that it is safe to make any assumption about the values of attributes outside their control.

Indeed in a real professional database we could not take the risk to have two different movies with the same name and year, even if this is very unlikely.

In order to avoid this risk a unique ID can be created for every tuple of a database.

For example, companies generally assign employee ID's to all employees, and these ID's are carefully chosen to be unique numbers. One purpose of these ID's is to make sure that in the company database each employee can be distinguished from all others, even if there are several employees with the same name.

Example of a database

See document distributed.

3. Review of SQL

SQL is the principal language used to describe and manipulate relational databases.

There is a current standard for SQL, called SQL-99. Most commercial database management systems implement something similar, but not identical to, the standard.

There are two aspects to SQL:

- 1 the *data definition* sublanguage for declaring database schemas;
- 2 the *data manipulation* sublanguage for querying (asking questions about) databases and for modifying the database.

SQL makes a distinction between three kinds of relations:

- 1 Stored relations, which are called *tables*. These are the kind of relation we deal with ordinarily — a relation that exists in the database and that can be modified by changing its tuples, as well as queried.
- 2 *Views*, which are relations defined by a computation. These relations are not stored, but are constructed, in whole or in part, when needed.
- 3 Temporary tables, which are constructed by the SQL language processor when it performs its job of executing queries and data modifications.
These relations are then thrown away and not stored.

In this part of the course, we will learn how to declare tables. Later we will treat the declaration and definition of views; temporary tables are never declared.

The SQL CREATE TABLE statement

The SQL CREATE TABLE statement declares the schema for a stored relation. It gives a name for the table, its attributes, and their data types. It also allows us to declare a key, or even several keys, for a relation.

The CREATE TABLE statement allows also to declare other forms of constraints, and to declare *indexes* (data structures that speed up many operations on the table, treated later in the course).

Data types

The primitive data types supported by SQL systems are:

- 1 Character strings: of fixed length, *CHAR*(*n*); of varying length, *VARCHAR*(*n*). (*n* is a natural number)
- 2 Bit strings: of fixed length, *BIT*(*n*); of varying length, *BIT VARYING*(*n*).
- 3 *BOOLEAN*, logical values taking three possible values: *TRUE*, *FALSE*, and *UNKNOWN*.
- 4 Integer numbers, *INT* or *INTEGER* (these names are synonyms) and *SHORTINT*.
- 5 Floating-point numbers: *FLOAT* or *REAL* (these are synonyms), *DOUBLE PRECISION*, *DECIMAL*(*n,d*) and *NUMERIC*.
- 6 Dates (*DATE*) and times (*TIME*).

Character strings

In SQL strings are surrounded by single-quotes, not double-quotes (so in SQL you write 'foo' and not "foo").

There are two character strings data types in SQL: *CHAR*(*n*) and *VARCHAR*(*n*) for a natural number *n*.

Both *CHAR*(*n*) and *VARCHAR*(*n*) are strings of up to *n* characters. The difference is implementation-dependent; typically *CHAR* implies that short strings are padded to make exactly *n* characters, while *VARCHAR* can have less than *n* characters.

Bit strings - Integer numbers

BIT(n) denotes bit strings of length n , while *BIT VARYING*(n) denotes bit strings of length up to n . They are analogous to *CHAR*(n) and *VARCHAR*(n) respectively.

INT and *INTEGER* (these names are synonyms) denote typical integer values. The type *SHORTINT* also denotes integers, but the number of bits permitted may be less, depending on the implementation.

Floating-point numbers

Floating-point numbers can be represented in a variety of ways.

FLOAT or *REAL* (these are synonyms) may be used for typical floating point numbers. A higher precision can be obtained with the type *DOUBLE PRECISION*.

SQL also has types that are real numbers with a fixed decimal point. For example, *DECIMAL*(n,d) allows values that consist of n decimal digits, with the decimal point assumed to be d positions from the right. Thus, 0123.45 is a possible value of type *DECIMAL*(6,2).

NUMERIC is almost a synonym for *DECIMAL*, although there are possible implementation-dependent differences.

A date value is the keyword *DATE* followed by a quoted string of the form 'yyyy-mm-dd'.

The first four characters are digits representing the year. Then come a hyphen and two digits representing the month. Finally there is another hyphen and two digits representing the day. Note that single-digit months and days are padded with a leading 0.

For example, *DATE* '1948-05-14' represents the date of 14 May 1948

A time value is the keyword *TIME* and a quoted string. This string has two digits for the hour, on the 24-hour clock. Then come a colon, two digits for the minute, another colon, and two digits for the second.

If fractions of a second are desired, we may continue with a decimal point and as many significant digits as we like.

For instance, *TIME* '15:48:02.517' represents 3:48 plus 2.517 seconds in the afternoon.

Different SQL implementations may provide other representations for dates and times.

A simple table declaration

The simplest form of declaration of a relation schema consists of the keywords *CREATE TABLE* followed by the name of the relation and a parenthesized, comma-separated list of the attribute names and their types.

For example the relation with the following schema

*Movies(title: string, year: integer, length: integer, genre: string,
studioName: string, producerC#: integer)*

can be declared as

*CREATE TABLE Movies (title CHAR(100), year INT, length INT,
genre CHAR(10), studioName CHAR(30), producerC# INT);*

Remark: SQL is not case sensitive

SQL is not case sensitive, therefore the keyword “CREATE TABLE” is equivalent to “create table” or “cREAtE taBLe”.

However the names of databases, tables and columns can be case sensitive depending on the specific implementation of SQL.

The previous “CREATE TABLE” statement could have been written

```
create table Movies (title char(100), year int, length int,  
genre char(10), studioName char(30), producerC# int)
```

or

```
CREATE TABLE Movies (title CHAR(100), year INT, length  
INT, genre CHAR(10), studioName CHAR(30), producerC# INT)
```

or even

```
cREAtE taBLe Movies (title ChAr(100), year inT, length InT,  
genre ChaR(10), studioName ChAr(30), producerC# Int))
```

Another table declaration

The relation *MovieStar* with the following schema

MovieStar(name: string, address: string, gender: char, birthdate: date)

can be declared as

```
CREATE TABLE MovieStar (name CHAR(30),  
address VARCHAR(255), gender CHAR(1), birthdate DATE);
```

Remark. For the name a fixed-length string of 30 characters is used, padding a name out with blanks at the end if necessary and truncating a name to 30 characters if it is longer.

For the address a variable-length character strings of up to 255 characters is used.

Deleting a relation

To delete a relation R we use the SQL statement:

DROP TABLE R;

This removes the table, including all of its current tuples.

The relation R will be no longer part of the database schema, and we can no longer access any of its tuples.

Modifying relation schemas

To modify the schema of an existing relation we use the statement with the keywords *ALTER TABLE* and the name of the relation.

There are several options, the most important of which are

- 1 *ADD* followed by an attribute name and its data type.
- 2 *DROP* followed by an attribute name.

Modifying relation schemas: example 1

We want to modify the *MovieStar* relation seen before by adding the attribute *phone*. Then we use the following statement:

```
ALTER TABLE MovieStar ADD phone CHAR(16);
```

The *MovieStar* schema now becomes

MovieStar(name: string, address: string, gender: char, birthdate: date,
phone: string)

In the actual relation, tuples would all have components for phone, but the value of each of these components is set to the special null value, *NULL*.

Later we will see how it is possible to choose another “default” value to be used instead of *NULL* for unknown values.

Modifying relation schemas: example 2

We want to delete the attribute *birthdate* of the *MovieStar* relation. We use the following statement:

```
ALTER TABLE MovieStar DROP birthdate;
```

As a result, the schema for *MovieStar* no longer has that attribute and becomes

```
MovieStar(name: string, address: string, gender: char, phone: string)
```

All tuples of the current *MovieStar* instance have the component for *birthdate* deleted and we can no longer access any of these components.

Default values

In any place where we declare an attribute and its data type we may add the keyword *DEFAULT* and an appropriate value in order to specify a *default* value.

That value is either *NULL* or a constant. Certain other values that are provided by the system, such as the current time, may also be options.

Remark: if we do not specify a default value, the default value will be *NULL*.

Default values: example 1

We could use the character ? as the default for an unknown gender. Also we could use the earliest possible date, *DATE '0000-00-00'* for an unknown birthdate.

We could replace the declarations of gender and birthdate in *MovieStar* by:

```
gender CHAR(1) DEFAULT '?',  
birthdate DATE DEFAULT DATE '0000-00-00'
```

Thus the declaration of the *MovieStar* table becomes

```
CREATE TABLE MovieStar (name CHAR(30), address VARCHAR(255),  
gender CHAR(1) DEFAULT '?',  
birthdate DATE DEFAULT DATE '0000-00-00');
```

Default values: example 2

We could have declared the default value for the new attribute *phone* to be *'unlisted'*.

In that case the appropriate *ALTER TABLE* statement would have been

```
ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT 'unlisted';
```

Key constraints

There are three types of key constraints: 1) *unique* values; 2) *primary key*; 3) *foreign key*.

Any attempt to violate these constraints is rejected by the system.

Unique values and primary key

Let S be a set of attributes. If S is declared to have *unique* values then two different tuples cannot agree on all the attributes in S unless at least one of them is *NULL* (but they can agree on a proper subset of S).

If S is declared to be a *primary key* then the same constraints as for unique values hold and moreover no tuple is allowed to have a *NULL* value in an attribute in S .

If S is declared to be unique but not a primary key then the *NULL* value is admissible in one or more attributes of S .

A relation can have more than one unique constraint but it cannot have more than one primary key constraint.

Suppose that in the *Movies* table the set of attributes made by *title* and *year* is declared to be unique. Then this means that we cannot accept two tuples having the same values in both *title* and *year* unless at least one of them is *NULL*.

Also two tuples can agree on *title* and have non-null values in *year*, on conditions that the latter are different; and two tuples can agree on *year* and have non-null values in *title*, on conditions that the latter are different

The following instance of *Movies* is admissible:

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>
Star Wars	1977	231	sciFi
Star Wars	1978	231	sciFi
Star Wars 2	1977	231	sciFi
Star Wars	NULL	231	sciFi
Star Wars	NULL	231	sciFi
NULL	1977	231	sciFi
NULL	1977	231	sciFi
NULL	NULL	231	sciFi
NULL	NULL	231	sciFi

but we would have an error if we now tried to insert the tuple (Star Wars, 1977, 152, drama).

Now suppose that in the *Movies* table the set of attributes made by *title* and *year* is declared to be a primary key. Then not only we cannot accept two tuples having the same values in both *title* and *year*, but we cannot accept either a tuple having *NULL* in *title* or *year*.

Foreign key constraints

Let R_1 and R_2 be two relations not necessarily distinct, let A_1 and A_2 be sets of attributes of R_1 and R_2 respectively such that A_1 and A_2 have the same number of elements. If we declare that A_1 is a *foreign key of R_1 referencing A_2* then:

- 1 we must specify a bijection ϕ from A_1 to A_2 ;
- 2 the attributes of A_2 must be declared *UNIQUE* or the *PRIMARY KEY* for R_2 ;
- 3 given a tuple t_1 of R_1 there must exist a tuple t_2 of R_2 such that for any attribute a of A_1 such that t_1 is not null in a , then the value of t_1 in a is equal to the value of t_2 in $\phi(a)$.

Declaring key constraints

Unique values constraints are declared using the keyword *UNIQUE*.

Primary key constraints are declared using the keyword *PRIMARY KEY*.

Foreign key constraints are declared using the keywords *REFERENCES* and *FOREIGN KEY*.

Declaring primary key constraints 1/3

To declare that a set of attributes verifies a key constraint we add to the *CREATE TABLE* statement an additional declaration in the following way.

Suppose that we want to declare that in the *Movies* table the attributes *title* and *year* are a primary key. Then the *CREATE TABLE* statement is the following:

```
CREATE TABLE Movies (title CHAR(100), year INT,  
length INT, genre CHAR(10), studioName CHAR(30),  
producerC# INT, PRIMARY KEY (title, year));
```

Declaring primary key constraints 2/3

If the set of attributes verifying a key constraint is made by a single attribute then this constraint can also be declared by adding the corresponding keyword to the attribute name when that attribute is listed in the *CREATE TABLE* statement.

If the *MovieStar* has the attribute *name* as primary key then the *CREATE TABLE* statement is the following:

```
CREATE TABLE MovieStar (name CHAR(30) PRIMARY KEY,  
address VARCHAR(255), gender CHAR(1), birthdate DATE);
```

Declaring primary key constraints 3/3

The previously seen method to declare key constraints holds also when the key is made by a single attribute. For instance for the example of the previous slide the *CREATE TABLE* statement can be the following:

```
CREATE TABLE MovieStar (name CHAR(30),  
    address VARCHAR(255), gender CHAR(1),  
    birthdate DATE, PRIMARY KEY (name));
```

Declaring unique constraints

The syntax for unique constraints is the same as for primary keys : we only need to replace the keyword *PRIMARY KEY* with *UNIQUE*.

The only difference is that since more than one unique constraint is possible, then the following statement is admissible:

```
CREATE TABLE Movies (title CHAR(100), year INT,  
length INT, genre CHAR(10), studioName CHAR(30),  
producerC# INT, UNIQUE (title), UNIQUE (year));
```


Declaring foreign key constraints 1/2

To declare that a set of attributes verifies a foreign key constraint we add to the *CREATE TABLE* statement an additional declaration in the following way.

Suppose that we want to declare that in the *Studio* table the attribute *presC#* is a foreign key referencing the attribute *cert#* of the table *MovieExec*.

Then the *CREATE TABLE* statement is the following:

```
CREATE TABLE Studio (name CHAR(30), address VARCHAR(255),  
presC# INT, FOREIGN KEY (pres#) REFERENCES MovieExec(cert#)).
```

This type of declaration works with any number of attributes.

Declaring foreign key constraints 2/2

When the foreign key is made by a single attribute, then we can declare it in the following way:

```
CREATE TABLE Studio (name CHAR(30), address VARCHAR(255),  
    presC# INT REFERENCES MovieExec(cert#)).
```

The end.