

# Principles of Database Systems (CS307)

## Lecture 2: Introduction to Relation Model and SQL

**Yuxin Ma**

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 (7<sup>th</sup> Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.

# Relational Model

# Relation Schema and Instance

- $A_1, A_2, \dots, A_n$  are attributes
- $R = (A_1, A_2, \dots, A_n)$  is a **relation schema**
  - Example on the right side:  
*instructor* = (*ID*, *name*, *dept\_name*, *salary*)
- $r(R)$  denotes a relation instance  $r$  defined over schema  $R$ 
  - Or to say, the entire table on the right side
- An element  $t$  of relation  $r$  is called a **tuple**
  - ... and is represented by a row in a table

The relation schema ("R")

$A_1$	$A_2$	$A_3$	$A_4$
<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

$r(R)$

A tuple

# Relation Schema and Instance

- An analogy to programming languages:
  - Relation - Variables
  - Relation schema – Variable types
  - Relation instance – Value(s) stored in the variable

# Something More about Attributes

- **Domain** (of an attribute): The set of allowed values for the attribute
  - E.g., the domain for *dept\_name* is the set of all existing departments in the university
- Being **atomic** (or “**atomicity**”): indivisible
  - E.g., *salary* is considered indivisible, and full names of instructors may not always be indivisible (which can be further divided into “family name” and “surname name”)

# NULL in Attributes

- **null**: a special marker that represents an “unknown” status
  - $\text{null} \neq 0$ ;  $\text{null} \neq 0.0$ ;  $\text{null} \neq \text{false}$ 
    - Null is null, a marker indicating that a data value is missing
  - In practice (SQL for example), it is not advised to treat **null** as a “value”
    - ... however, you can always see the term “null value” in technical articles
- Complications will be introduced when null appears in many operations
  - “three-valued logic”, “ $1 + \text{null} = \text{null}$ ”, etc., which will be introduced later

# Relations are Unordered

- Order of tuples is **irrelevant**
  - ... that is, tuples may be stored in an arbitrary order
- Example: *instructor* relation with unordered tuples

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

Considered the same relation

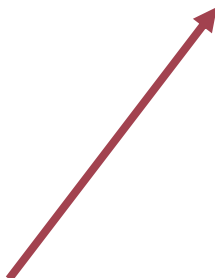
# Database Schema

- Database schema is the **logical structure** of the database
  - It contains a set of **relation schemas** and a set of **integrity constraints**
- Database instance is a **snapshot** of the data in the database at a given instant in time

## Schema

*instructor(ID, name, dept\_name, salary)*

An **instance** of the schema:



<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000



# Keys

- Let  $K \subseteq R$ 
  - $K$  is a **superkey** of  $R$  if values for  $K$  are sufficient to identify a unique tuple of each possible relation  $r(R)$ 
    - E.g.,  $\{ID\}$  and  $\{ID, name\}$  are both **superkeys** of instructor
    - If  $K$  is a superkey, any superset  $K'$  of  $K$  where  $K' \subseteq R$  is a superkey as well
  - Superkey  $K$  is a **candidate key** if  $K$  is minimal, i.e., no proper subset of  $K$  is a superkey
    - E.g.,  $\{ID\}$  is a candidate key for *instructor*

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

*instructor*

# Keys

- Let  $K \subseteq R$ 
  - One of the candidate keys is selected to be the **primary key**
    - We mark the primary key with an underline:  
*instructor* = (ID, name, dept\_name, salary)

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

*instructor*

# Keys

- Let  $K \subseteq R$ 
  - **Foreign key** constraint: Values in one relation must appear in another relation
    - E.g., *dept\_name* in *instructor* is a foreign key from *instructor* referencing *department*

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

*instructor*

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Physics	Watson	70000
Finance	Painter	120000
History	Painter	50000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Biology	Watson	90000
Comp. Sci.	Taylor	100000
History	Painter	50000
Comp. Sci.	Taylor	100000
Music	Packard	80000
Physics	Watson	70000
Finance	Painter	120000

*department*

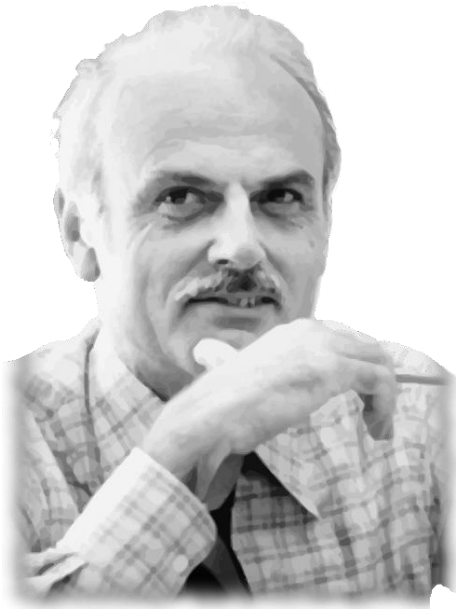
# Introduction to SQL

# How Do We Manage Data in a Database?

- Some examples
  - Return the instructor(s) with the highest salary but not from science departments
  - Update instructors' salary in the physics department by increasing 5%
- 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.

$$(r_1[2], r_1[3]): P_1 r_1 \wedge \exists P_2 r_2 (r_2[2] = 15 \wedge r_2[1] = r_1[1]).$$

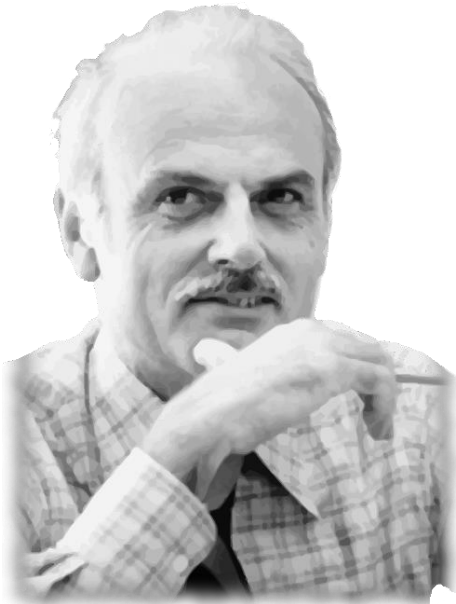
- Find the locations of suppliers and the parts being supplied by them (omitting those suppliers who are supplying no parts at this time).

$$(r_1[3], r_2[2]): P_1 r_1 \wedge P_2 r_2 \wedge (r_1[1] = r_2[1]).$$

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.

$$(r_1[2], r_1[3]): P_1 r_1 \wedge \exists P_2 r_2 (r_2[2] = 15 \wedge r_2[1] = r_1[1]).$$

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

$$(r_1[3], r_2[2]): P_1 r_1 \wedge P_2 r_2 \wedge (r_1[1] = r_2[1]).$$

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

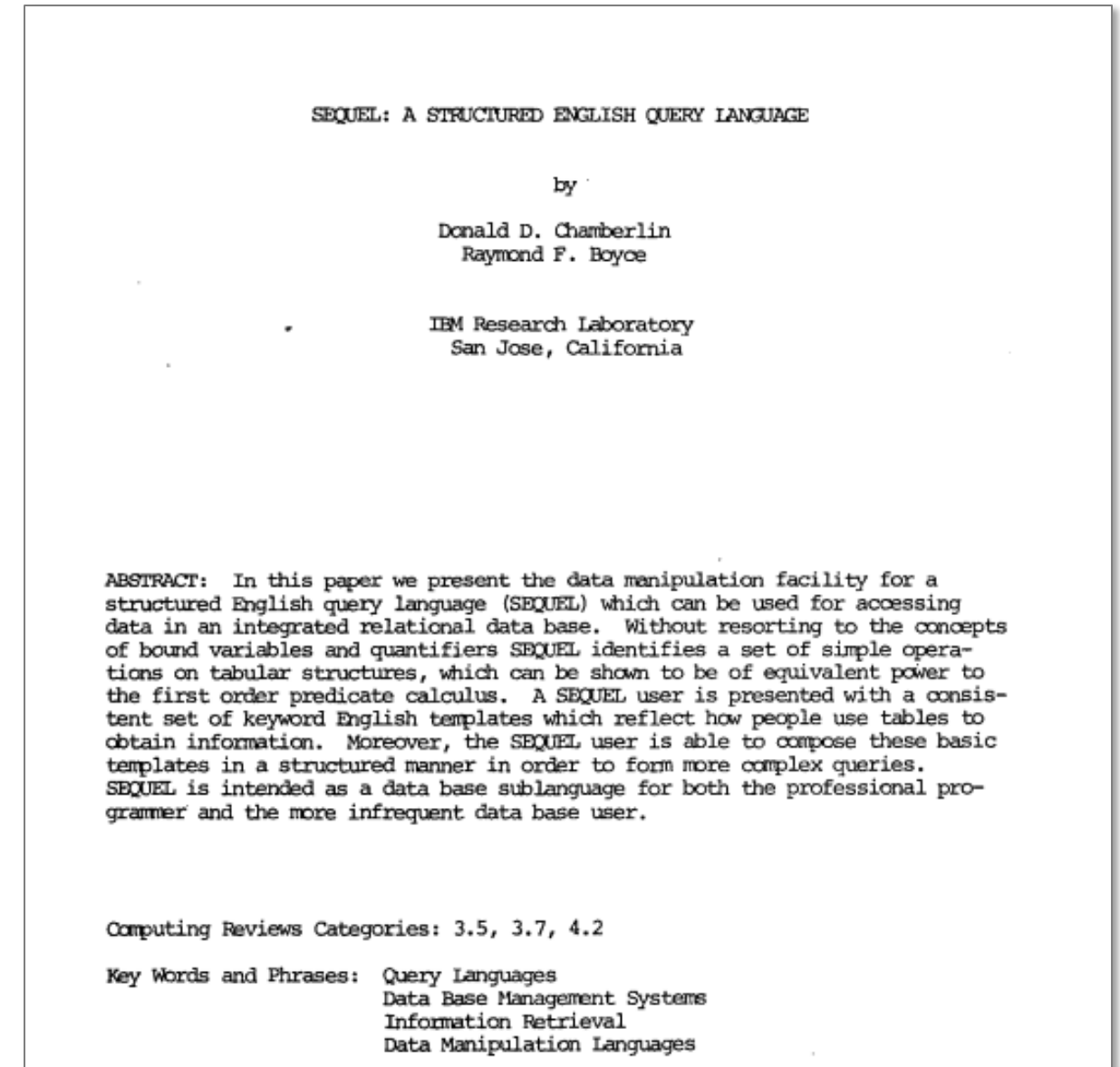
However, it didn't excite enthusiasm at  
IBM

# 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)



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!) A huge problem that has appeared (or will appear) many times at many places
  - 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.

# Standardization of SQL

Year	Official standard	Informal name	Comments
1986, 1987	ANSI X3.135:1986, <a href="#">ISO/IEC 9075</a> :1987, FIPS PUB 127	<a href="#">SQL-86</a> , SQL-87	First formalized by ANSI, adopted as <a href="#">FIPS</a> PUB 127
1989	ANSI X3.135-1989, ISO/IEC 9075:1989, FIPS PUB 127-1	<a href="#">SQL-89</a>	Minor revision that added integrity constraints, adopted as FIPS PUB 127-1
1992	ANSI X3.135-1992, ISO/IEC 9075:1992, FIPS PUB 127-2	<a href="#">SQL-92</a> , SQL2	Major revision (ISO 9075), <i>Entry Level</i> SQL-92, adopted as FIPS PUB 127-2
1999	ISO/IEC 9075:1999	<a href="#">SQL:1999</a> , SQL3	Added regular expression matching, <a href="#">recursive queries</a> (e.g., <a href="#">transitive closure</a> ), <a href="#">triggers</a> , support for procedural and control-of-flow statements, nonscalar types (arrays), and some object-oriented features (e.g., <a href="#">structured types</a> ), support for embedding SQL in Java ( <a href="#">SQL/OLB</a> ) and vice versa ( <a href="#">SQL/JRT</a> )
2003	ISO/IEC 9075:2003	<a href="#">SQL:2003</a>	Introduced <a href="#">XML</a> -related features ( <a href="#">SQL/XML</a> ), <a href="#">window functions</a> , standardized sequences, and columns with autogenerated values (including identity columns)
2006	ISO/IEC 9075-14:2006	<a href="#">SQL:2006</a>	Adds Part 14, 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 <a href="#">XQuery</a> , the XML Query Language published by the World Wide Web Consortium ( <a href="#">W3C</a> ), to concurrently access ordinary SQL-data and XML documents. <sup>[30]</sup>
2008	ISO/IEC 9075:2008	<a href="#">SQL:2008</a>	Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers, TRUNCATE statement, <sup>[31]</sup> FETCH clause
2011	ISO/IEC 9075:2011	<a href="#">SQL:2011</a>	Adds temporal data (PERIOD FOR) <sup>[32]</sup> (more information at <a href="#">Temporal database#History</a> ). Enhancements for <a href="#">window functions</a> and FETCH clause. <sup>[33]</sup>
2016	ISO/IEC 9075:2016	<a href="#">SQL:2016</a>	Adds row pattern matching, polymorphic table functions, operations on <a href="#">JSON</a> data stored in character string fields
2019	ISO/IEC 9075-15:2019	<a href="#">SQL:2019</a>	Adds Part 15, multidimensional arrays (MDarray type and operators)
2023	ISO/IEC 9075:2023	<a href="#">SQL:2023</a>	Adds data type JSON (SQL/Foundation); Adds Part 16, Property Graph Queries (SQL/PGQ)

# Standardization of SQL

(any other examples of standardization?)

Year	Official standard	Informal name	Comments
1986, 1987	ANSI X3.135:1986, <a href="#">ISO/IEC 9075</a> :1987, FIPS PUB 127	<a href="#">SQL-86</a> , SQL-87	First formalized by ANSI, adopted as <a href="#">FIPS</a> PUB 127
1989	ANSI X3.135-1989, ISO/IEC 9075:1989, FIPS PUB 127-1	<a href="#">SQL-89</a>	Minor revision that added integrity constraints, adopted as FIPS PUB 127-1
1992	ANSI X3.135-1992, ISO/IEC 9075:1992, FIPS PUB 127-2	<a href="#">SQL-92</a> , SQL2	Major revision (ISO 9075), <i>Entry Level</i> SQL-92, adopted as FIPS PUB 127-2
1999	ISO/IEC 9075:1999	<a href="#">SQL:1999</a> , SQL3	Added regular expression matching, <a href="#">recursive queries</a> (e.g., <a href="#">transitive closure</a> ), <a href="#">triggers</a> , support for procedural and control-of-flow statements, nonscalar types (arrays), and some object-oriented features (e.g., <a href="#">structured types</a> ), support for embedding SQL in Java ( <a href="#">SQL/OLB</a> ) and vice versa ( <a href="#">SQL/JRT</a> )
2003	ISO/IEC 9075:2003	<a href="#">SQL:2003</a>	Introduced <a href="#">XML</a> -related features ( <a href="#">SQL/XML</a> ), <a href="#">window functions</a> , standardized sequences, and columns with autogenerated values (including identity columns)
2006	ISO/IEC 9075-14:2006	<a href="#">SQL:2006</a>	Adds Part 14, 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 <a href="#">XQuery</a> , the XML Query Language published by the World Wide Web Consortium ( <a href="#">W3C</a> ), to concurrently access ordinary SQL-data and XML documents. <sup>[30]</sup>
2008	ISO/IEC 9075:2008	<a href="#">SQL:2008</a>	Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers, TRUNCATE statement, <sup>[31]</sup> FETCH clause
2011	ISO/IEC 9075:2011	<a href="#">SQL:2011</a>	Adds temporal data (PERIOD FOR) <sup>[32]</sup> (more information at <a href="#">Temporal database#History</a> ). Enhancements for <a href="#">window functions</a> and FETCH clause. <sup>[33]</sup>
2016	ISO/IEC 9075:2016	<a href="#">SQL:2016</a>	Adds row pattern matching, polymorphic table functions, operations on <a href="#">JSON</a> data stored in character string fields
2019	ISO/IEC 9075-15:2019	<a href="#">SQL:2019</a>	Adds Part 15, multidimensional arrays (MDarray type and operators)
2023	ISO/IEC 9075:2023	<a href="#">SQL:2023</a>	Adds data type JSON (SQL/Foundation); Adds Part 16, Property Graph Queries (SQL/PGQ)

# 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
<pre>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"</pre>	<pre>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';</pre>

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



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

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

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 */
```

# 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-constraint 1),  
    ...,  
    (integrity-constraint k)  
)
```

- $r$  is the name of the relation
- each  $A_i$  is an attribute name in the schema of relation  $r$
- $D_i$  is the data type of values in the domain of attribute  $A_i$

# 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` 

# 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 stored 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](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)  
)
```

- 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

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 mandatory before 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	...	...	...	...

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



```
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)  
)
```

# 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	...	...	...	...

No identical first names for any two people here

- But it is not what we want in this table



```
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)  
)
```

# 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 constraint 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

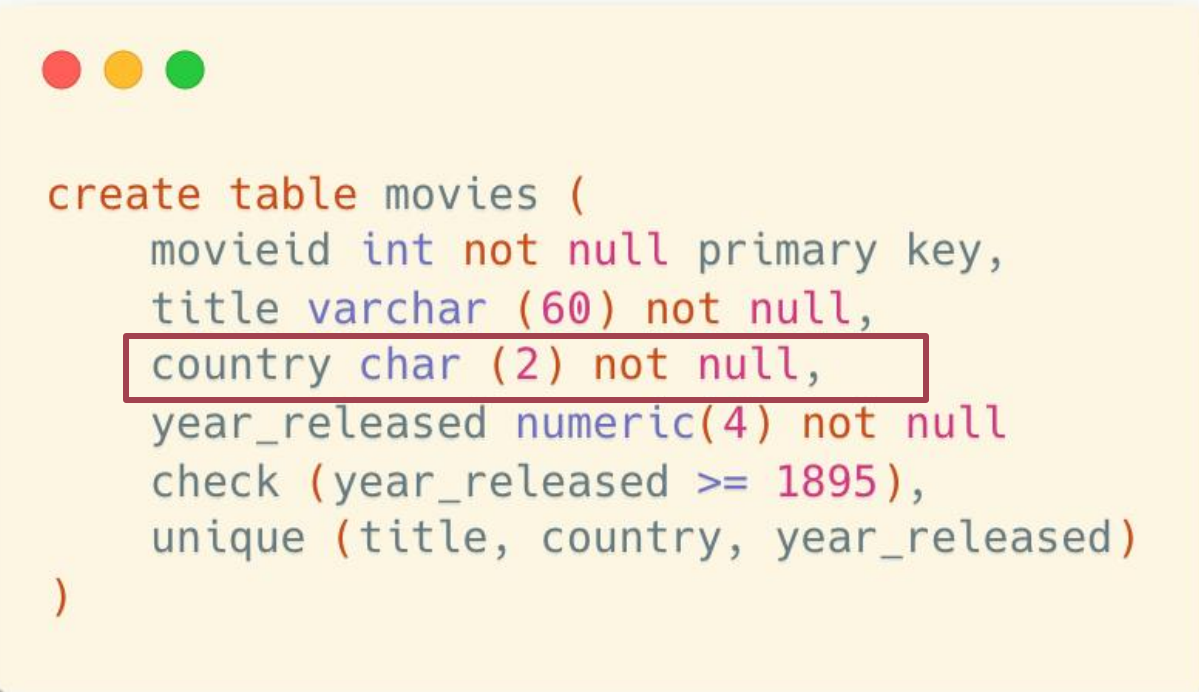
```
create table people (  
    peopleid int not null  
                primary key,  
    first_name varchar(30),  
    surname varchar(30) not null,  
    born numeric(4) not null,  
    died numeric(4),  
    unique (first_name, surname),  
    check (died - born >= 0),  
    check (first_name = upper(first_name)),  
    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 constraints 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 constraints 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 (A<sub>m</sub>, ..., A<sub>n</sub>) 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 (A<sub>m</sub>, ..., A<sub>n</sub>) 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

```
insert into lab (address, onetime, 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 will 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 character)