

APAN PS5400: Managing Data

Week 2: Introduction to Relational Data Bases

Lecturer: François Scharffe

Recap of last week

- Varieties of data
- Operations on data
- Structured vs Semi-structured vs. Unstructured data
- Big data
- Data vs. Information vs. Knowledge

This week

- Relational database model
 - Relations, tuples
- Relational database tables and schemas
- Data types for relational databases
- Relational Database Management System (RDBMS)
- SQL as a query language
- Installing Postgres SQL on your computer

Relations: Logical view

- A relation is a set of tuples
 - Recall a set cannot have duplicates and the order of elements is of no significance
- A tuple is a sequence of elements
 - Tuples can have duplicates
 - Order of elements in a tuple is significant

Examples

A tuple: <“jane”, 25, “female”>

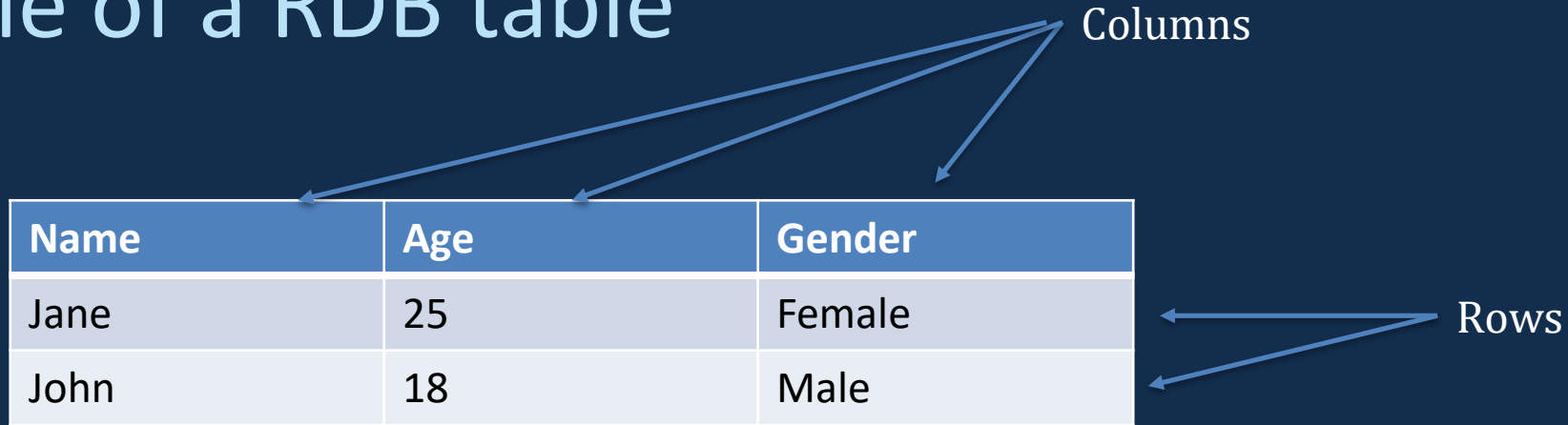
A set of tuples: {<“jane”, 25, “female”>, <“john”, 18, “male”>}

Note the values that occur in the corresponding positions in the two tuple.

Relational Database Model

- A relational database (RDB) consists of tables
 - A table is a set of tuples
 - But not any set of tuples can be a table (in an RDB)
- An RDB table
 - Is a set of tuples in which every tuple has the same number of elements
 - In contrast to NoSQL databases, which we will study later
 - Each component of every tuple must be atomic (no composite structures)
 - The data types of the corresponding elements in each position in each tuple must be the same

Example of a RDB table



Name	Age	Gender
Jane	25	Female
John	18	Male

- This is just a graphical representation of a set of tuples
- The first row consists of name of columns. It is meta-data, not data.
- The name of the column plus the data type of values in the column is called a field or an attribute.
 - The second field is named 'Age' and stores integers (data type).
 - All the values in the each column must be of the same data type (or a special value called 'null').
- The order of the values in a row matter. Interchanging 'Male' and 'John' would result in 'Male' being treated as a name.

Schema

- We want to distinguish between the structure of a RDB table and the data stored in it at any given moment.
- The data can change frequently but the structure changes very infrequently.
- The structure is often referred to as the 'Schema' of a table in textbooks.
 - A table is its schema plus the data stored in it (data with meta-data)
 - Warning: RDB practitioners use 'Schema' to refer to the structure of an entire database as opposed to the structure of a table.

Schema Example

Friends (name of the schema, not part of the schema)

Name: Varchar (String)

Age: Integer

Gender: Varchar (String)

Also represented as:

Friends(Name: Varchar (String), Age: Integer, Gender: Varchar (String))

Or as:

Friends(Name, Age Gender)

Data Types

- `char(n)` Fixed length character string, with user-specified length `n`
- `varchar(n)` Variable length character strings, with user-specified maximum length `n`.
- `int`. Integer (a finite subset of the integers that is machine-dependent).
- `smallint`. Small integer (a machine-dependent subset of the integer domain type).
- `numeric(p,n)`. Fixed point number, with user-specified precision of `p` digits, with `n` digits to the right of decimal point.
- `real`, `double precision`. Floating point and double-precision floating point numbers, with machine-dependent precision.
- `float(n)`. Floating point number, with user-specified precision of at least `n` digits.
- Most systems also have other types such as `Date`, `Time`, `URL`, etc.

Creating a RDB table

- Creating a table is the same as creating a schema for the table
 - Also adding some constraints and keys—more on that later
- Specify the name of the table (e.g., Friends)
- Specify the fields/attributes and their data types
- We will study next week how to create tables using SQL (Structured Query Language)

Relational Databases (RDB) and Relational Database Management Systems (RDBMS)

- A relational database consists of a set of relational tables
 - Plus, possibly, some foreign key constraints—more on that later
- A Relational Database Management System is a set of programs for managing and facilitating all the operations on a database while maintaining the integrity and security of the data
 - We will be using the Postgres RDBMS in this class

Drawbacks of using file systems to store data

Before the use of databases, data was stored in files

- Data redundancy and inconsistency
 - Multiple file formats, duplication of information in different files
- Difficulty in accessing data
 - Need to write a new program to carry out each new task
- Data isolation — the files are not connected with each other
- Integrity problems
 - Integrity constraints (e.g., $GPA \geq 0$) become “buried” in program code rather than being stated explicitly
 - Hard to add new constraints or change existing ones

drawbacks of using a file system (continued)

- No way of enforcing atomicity of updates
 - Failure may leave the data in an inconsistent state
 - Example: Transfer of funds from one account to another should either complete or not happen at all
- No system for managing concurrent accesses by different users (over and above that provided by operating system)
 - Concurrent accesses needed for performance
 - Uncontrolled concurrent accesses can leave the data in an inconsistent state
- Security problems
 - Can't give users access to some but not all the data

A DMBS provides a solution to all these problems.

RDBMS and ACID properties

- A database transaction is any sequence of operation of reading (querying) data from the database or writing (inserting, modifying) data into the database, which can be viewed as a single task.
 - E.g., transferring funds from one bank account into another account
- An RDBMS must manage multiple concurrent transactions from multiple users at the same time without any unintended and undesirable effects.
- RDBMS provide ACID (Atomicity, Consistency, Isolation, Durability) guarantees on transactions
 - Atomicity: Every transaction is either entered into the DB in its entirety or not at all
 - Consistency: Only those transactions are executed which will not make a consistent DB inconsistent
 - Isolation: Each transaction is executed as if it is the only transaction being executed at that time
 - Durability: Once a transaction is committed, it remains in the DB even in the face of system failures
- Note: Different type of NoSQL databases relax these ACID guarantees in different ways

SQL (Structured Query Language)

- Can be used to create, drop, and alter tables (schema)
 - We will study that next time
- Can be also be used to
 - Retrieve data from tables
 - Insert new data into tables
 - Update (modify) existing data in tables
 - We will study how to do that this lecture

<i>customer_id</i>	<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

<i>account_number</i>	<i>balance</i>
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

<i>customer_id</i>	<i>account_number</i>
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

From Database System Concepts, 6th Ed

Basic Form of SQL queries

select *list of column names (required)*
from *list of tables (required)*
where *Boolean expression (optional)*

```
select  account_number
from    account
where   balance > 700
```



A-201
A-217

Result of executing a query is itself a table.

```
select  D.customer_id, D.account_number
from    account AS A, depositor as D
where   A.balance > 700 AND
        A.account_number = D.account_number
```



019-28-3746	A-201
321-12-3123	A-217

```
select  D.customer_id, D.account_number
from    account AS A, depositor AS D
where   A.balance > 700 AND
        A.account_number = D.account_number
```

alias

The condition *A.account_number = D.account_number* is called a **join** on the tables account and depositor on the account_number field

Why do we need a join in the above query?

- The technical explanation is in terms of the cross product operation in relational algebra. We won't cover relational algebra in this class.
- Informal explanation: It is a reference from account_numbers in A to account_numbers in D. Think of it as A asking D to provide information about such and such account number, D looking up information for that account_number in its table. So the account_numbers in D must match (i.e., =) those sent by A.
- We will look at joins next

Joins of tables

A join (more precisely, natural join) of two tables X and Y (written as $X \bowtie Y$) is a table Z s.t.

- the schema of Z consists of the attributes of X followed by the attributes of Y but containing only a single copy of any attributes they may have in common, and
- The tuples of Z are created by
 - For each tuple of X appending each tuple of Y to create a tuple $t_x t_y$
 - Eliminating those $t_x t_y$ in which it doesn't contain the same values for attributes common to the schema of X and Y
 - For those $t_x t_y$ that survive step 2, eliminating any values coming from t_y that duplicate the values coming from t_x on attributes common to X and Y

Enrollment

Name	Course
Smith	CS101
Smith	Phil222
Liu	Bio540
Liu	CS101

Location

Course	Room
CS101	Fu24
Phil222	P454
Bio540	Lew606
Math650	M245
Math245	M456

Join of Enrollment with Location

Name	Course	Room
Smith	CS101	Fu24
Smith	Phil222	P454
Liu	Bio540	Lew606
Liu	CS101	Fu24

```
select    D.customer_id, D.account_number
from      account AS A, depositor AS D
where     A.balance > 700 AND
           A.account_number = D.account_number
```

This query can be understood as saying: do a join on *account* and *depositor* tables, select only those rows in the joined table where *balance* > 700 and show me the *customer_id* and *account_number* fields in the joined table.

SQL allows duplicates by default in query results.

To force the elimination of duplicates, insert the keyword **distinct** after select.

Find the customer_city of all customers from the customer table, and remove duplicates

```
SELECT DISTINCT customer_city  
FROM customer
```

An asterisk (*) in the SELECT clause denotes all attributes

```
SELECT *  
FROM customer  
WHERE customer_city = "Palo Alto"
```

String operations

SQL includes a string-matching operator for comparisons on character strings. The operator “like” uses patterns that are described using the special character percent (%). The % character matches any substring.

Find all customers whose name includes the substring “ind”.

```
select *  
from customer  
where customer_name like '%ind%'
```

Ordering the display of results

- List in alphabetic order the names of all customers
select *customer_name*
from *customer*
order by *customer_name*
- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
 - Example: **order by** *customer_name* **desc**

Aggregate functions

SQL supports the following aggregation functions:

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values

IMPORTANT: If the **SELECT** clause contains an aggregate function then it can contain no other attribute (except in **GROUP BY** queries --coming up next).

Find the average balance of accounts

```
select avg (balance)
from account
```

Find the total number of accounts whose balance is greater than 500

```
SELECT count(*)
FROM account
WHERE balance > 500
```

Find the sum of balance in all the accounts

```
SELECT sum(balance)
FROM account
```


<i>customer_id</i>	<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

<i>account_number</i>	<i>balance</i>
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

<i>customer_id</i>	<i>account_number</i>
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

Write a query to find the average balance of those customers who live in Harrison.

DO IT NOW IN CLASS—5 minutes

Answer:

```
SELECT AVG(balance)
FROM customer AS C, account AS A, depositor AS D
WHERE C.customer_city = "Harrison" AND
C.customer_id = D.customer_id AND
D.account_number = A.account_number
```

Aggregate Functions and Group BY

Find the average salary of instructors in each department

```
select dept_name, avg (salary)
from instructor
group by dept_name;
```

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



dept_name	salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

From Database System Concepts, 6th Edition

Restriction on Aggregation with Group By

Attributes in **select** clause other than aggregate functions must appear in **group by** list

```
/* erroneous query */  
select dept_name, ID, avg (salary)  
from instructor  
group by dept_name;
```

Note: A very common mistake. But note that having ID in the SELECT clause in this query does not make sense since ID is not an attribute of departments (unlike average salary).

Group By with Having clause

Find the names and average salaries of all departments whose average salary is greater than 50,000

```
select dept_name, avg (salary)
from instructor
group by dept_name
having avg (salary) > 50000;
```

Note: predicates in the **having** clause are filters on groups whereas predicates in the **where** clause are filters on the tuples that go into groups.

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.
- We will not go into this. You can learn to do that in the SQL elective course
- Examples in next slide

Examples of subqueries

Find courses offered in Fall 2009 and in Spring 2010

```
select distinct course_id
from section
where semester = ' Fall' and year= 2009 and
       course_id in
           (select course_id
            from section
            where semester = ' Spring' and year= 2010);
```

Find courses offered in Fall 2009 but not in Spring 2010

```
select distinct course_id
from section
where semester = ' Fall' and year= 2009 and
       course_id not in (select course_id
                          from section
                          where semester = ' Spring' and year= 2010);
```

Next Week

- SQL to create databases and tables
- SQL to insert data and update data
- Primary keys
- Integrity constraints