CMSC424: Database Design
Module: Relational Model; SQL

Instructor: Amol Deshpande
amol@umd.edu

CMSC424: Database Design

Module: Relational Model + SQL

Relational Model

Instructor: Amol Deshpande
amol@cs.umd.edu

Relational Model

- Book Chapters (6th Edition)
  - 2.1, 2.2, 2.4

- Key Topics
  - Relational Model Key Concepts
  - Domains of Table Attributes
  - Null Values
  - Schema Diagrams

Relational Data Model

Introduced by Ted Codd (late 60's – early 70's)

- *Before = "Network Data Model" (Cobol as DDL, DML)*
- *Very contentious: Database Wars (Charlie Bachman vs. Ted Codd)*

Relational data model contributes:

1. *Separation of logical, physical data models (data independence)*
2. *Declarative query languages*
3. *Formal semantics*
4. *Query optimization (key to commercial success)*

1st prototypes:

- *Ingres → CA*
- *Postgres → Illustra → Informix → IBM*
- *System R → Oracle, DB2*

Key Abstraction: Relation

Account =

bname	acct_no	balance
Downtown	A-101	500
Brighton	A-201	900
Brighton	A-217	500

Terms:

- Tables (aka: Relations)

Why called Relations?

*Closely correspond to mathematical concept of a **relation***

Relations

Account =

bname	acct_no	balance
Downtown	A-101	500
Brighton	A-201	900
Brighton	A-217	500

Considered equivalent to…

{ (Downtown, A-101, 500),
(Brighton, A-201, 900),
(Brighton, A-217, 500) }

Relational database semantics defined in terms of mathematical relations

Relations

Account =

bname	acct_no	balance
Downtown	A-101	500
Brighton	A-201	900
Brighton	A-217	500

Considered equivalent to…

{ (Downtown, A-101, 500),
(Brighton, A-201, 900),
(Brighton, A-217, 500) }

Terms:

- Tables (aka: Relations)
- Rows (aka: tuples)
- Columns (aka: attributes)
- Schema (e.g.: Acct_Schema = (bname, acct_no, balance))

Definitions

Relation Schema (or Schema)

A list of attributes and their domains

E.g. **account**(account-number, branch-name, balance)

Programming language equivalent: A variable (e.g. x)

Relation Instance

A particular instantiation of a relation with actual values

Will change with time

bname	acct_no	balance
Downtown	A-101	500
Brighton	A-201	900
Brighton	A-217	500

Programming language equivalent: Value of a variable

Definitions

Domains of an attribute/column

The set of permitted values

e.g., bname must be String, balance must be a positive real number

We typically assume domains are **atomic,** i.e., the values are treated as indivisible (specifically: you can't store lists or arrays in them)

Null value

A special value used if the value of an attribute for a row is:

unknown (e.g., don't know address of a customer)

inapplicable (e.g., "spouse name" attribute for a customer)

withheld/hidden

Different interpretations all captured by a single concept – leads to major headaches and problems

Tables in a University Database

classroom(building, room_number, capacity)

department(dept_name, building, budget)

course(course_id, title, dept_name, credits)

instructor(ID, name, dept_name, salary)

section(course_id, sec_id, semester, year, building,
room_number, time_slot_id)

teaches(ID, course_id, sec_id, semester, year)

student(ID, name, dept_name, tot_cred)

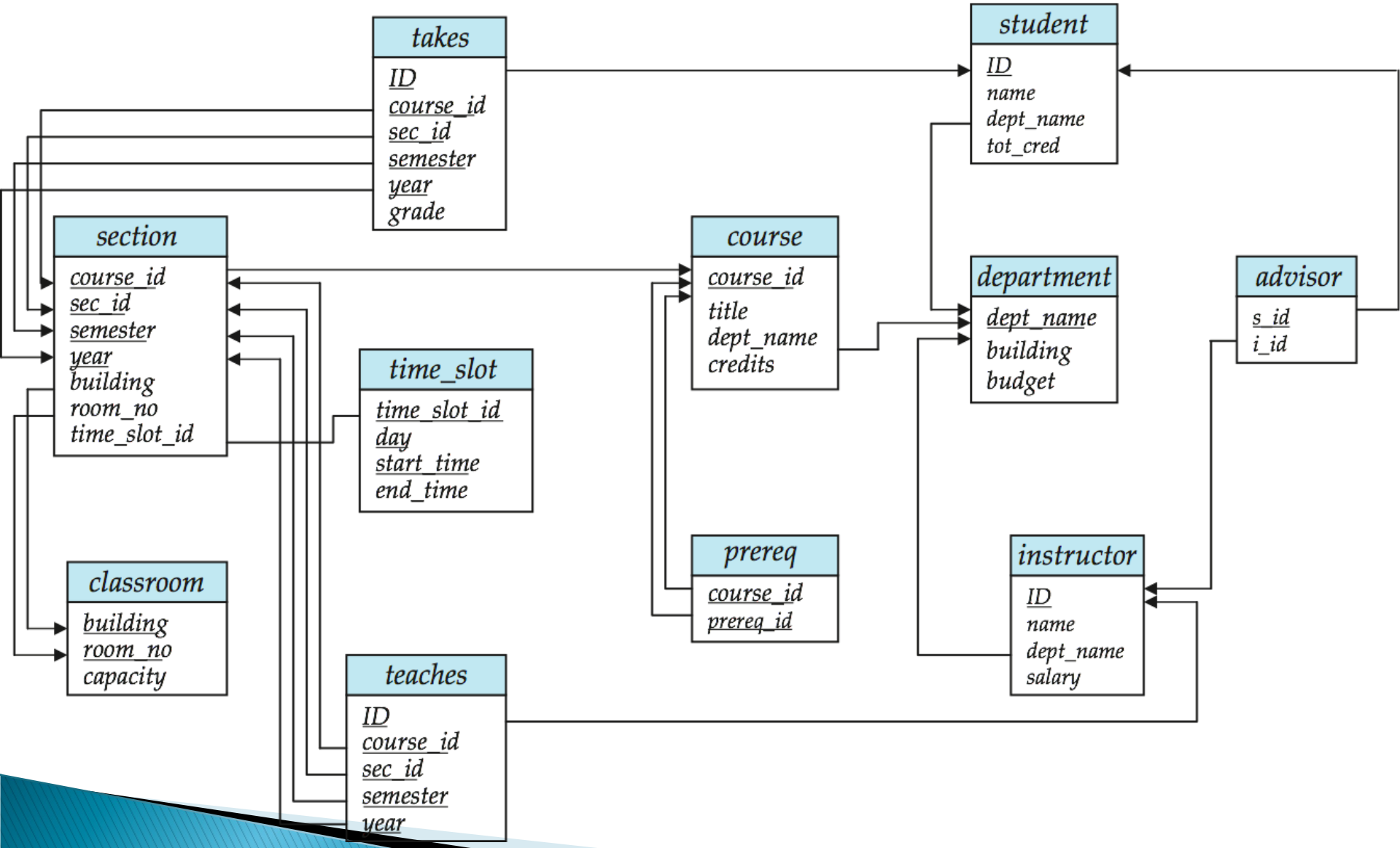
takes(Id, course_id, sec_id, semester, year, grade)

advisor(s_ID, i_ID)

time_slot(time_slot_id, day, start_time, end_time)

prereq(course_id, prereq_id)

Schema Diagram for University Database

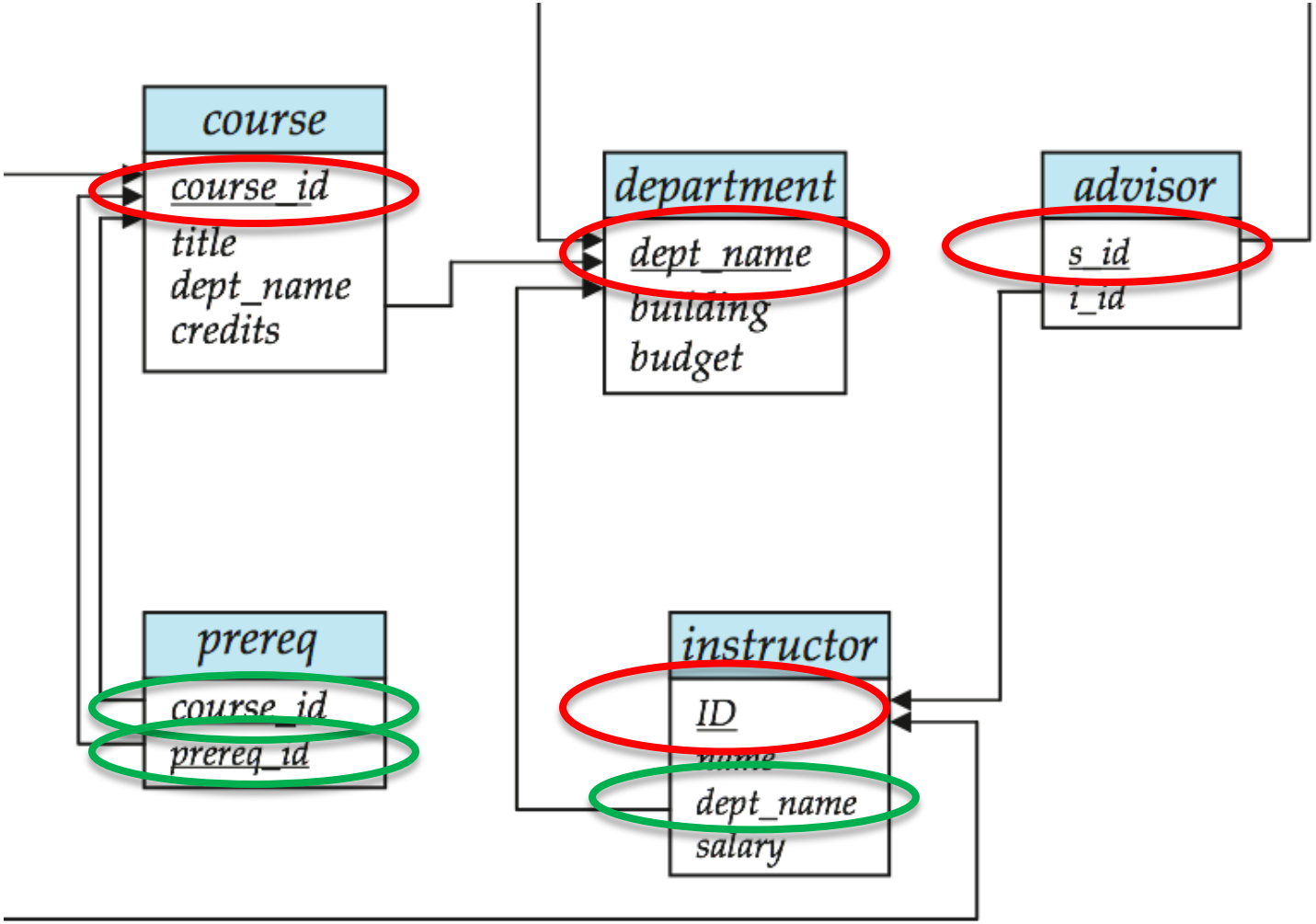


Schema Diagram for University Database

Primary Keys
Unique within a relation instance i.e., no two tuples with the same value

For "prereq":
combination of course_id, and prereq_id is unique

Foreign Keys
Used to connect tuples across relations



Referential Integrity

- Shouldn't have undefined or "dangling" connections (aka pointers)
  - e.g., what if there is no "History" tuple in "department"?
- Significant consistency issue with many NoSQL systems
  - RDBMS typically enforce this (if defined at schema level)

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

Figure 2.5 The *department* relation.

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

Instructor relation

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Basics and DDL

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL Basics and DDL

- Book Chapters (6th Edition)
  - 3.1, 3.2

- Key Topics
  - SQL Overview
  - How to create relations using SQL
  - How to insert/delete/update tuples

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86, SQL-89, SQL-92
  - SQL:1999, SQL:2003, SQL:2008
- 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.
- Several alternative syntaxes to write the same queries

Different Types of Constructs

- **Data definition language** (DDL): Defining/modifying schemas
  - **Integrity constraints:** Specifying conditions the data must satisfy
  - **View definition:** Defining views over data
  - **Authorization:** Who can access what
- **Data-manipulation language** (DML): Insert/delete/update tuples, queries
- **Transaction control:**
- **Embedded SQL:** Calling SQL from within programming languages
- **Creating indexes, Query Optimization control…**

Data Definition Language

The SQL **data-definition language (DDL)** allows the specification of information about relations, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- Also: other information such as
  - The set of indices to be maintained for each relations.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

SQL Constructs: Data Definition Language

- CREATE TABLE <name> (<field> <domain>, …)

```
create table department
    (dept_name varchar(20),
     building varchar(15),
     budget numeric(12,2) check (budget > 0),
     primary key (dept_name)
    );
```

```
create table instructor (
     ID         char(5),
     name       varchar(20) not null,
     dept_name  varchar(20),
     salary     numeric(8,2),
     primary key (ID),
     foreign key (dept_name) references department
)
```

SQL Constructs: Data Definition Language

- CREATE TABLE <name> (<field> <domain>, …)

```
create table department
    (dept_name varchar(20) primary key,
     building varchar(15),
     budget numeric(12,2) check (budget > 0)
);
```

```
create table instructor (
     ID        char(5) primary key,
     name    varchar(20) not null,
     d_name  varchar(20),
     salary    numeric(8,2),
     foreign key (d_name) references department
)
```

SQL Constructs: Data Definition Language

- drop table student
- delete from student
  - Keeps the empty table around
- alter table
  - alter table student add address varchar(50);
  - alter table student drop tot_cred;

SQL Constructs: Insert/Delete/Update Tuples

- INSERT INTO <name> (<field names>) VALUES (<field values>)

  insert into *instructor* **values** ('10211', 'Smith', 'Biology', 66000);

  insert into *instructor (name, ID)* **values** ('Smith', '10211');

  -- NULL for other two

  insert into *instructor (ID) values ('10211');*

  -- FAIL

- DELETE FROM <name> WHERE <condition>

  delete from department **where** budget < 80000;

  - Syntax is fine, but this command **may be rejected** because of referential integrity constraints.

SQL Constructs: Insert/Delete/Update Tuples

- DELETE FROM <name> WHERE <condition>

delete from department **where** budget < 80000;

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

Figure 2.5 The *department* relation.

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

Instructor relation

We can choose what happens:
(1) Reject the delete, or
(2) Delete the rows in Instructor (may be a cascade), or
(3) Set the appropriate values in Instructor to NULL

SQL Constructs: Insert/Delete/Update Tuples

- DELETE FROM <name> WHERE <condition>

delete from department **where** budget < 80000;

```
create table instructor
    (ID          varchar(5),
     name            varchar(20) not null,
     dept_name       varchar(20),
     salary          numeric(8,2) check (salary > 29000),
     primary key (ID),
     foreign key (dept_name) references department
        on delete set null
    );
```

We can choose what happens:
(1) Reject the delete (nothing), or
(2) Delete the rows in Instructor (on delete cascade), or
(3) Set the appropriate values in Instructor to NULL (on delete set null)

SQL Constructs: Insert/Delete/Update Tuples

- DELETE FROM <name> WHERE <condition>
  - Delete all classrooms with capacity below average

    delete from classroom **where** capacity <

    (**select avg(**capacity) **from** classroom);

  - Problem: as we delete tuples, the average capacity changes

  - Solution used in SQL:
    - First, compute **avg** capacity and find all tuples to delete
    - Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

  - E.g. consider the query: delete the smallest classroom

SQL Constructs: Insert/Delete/Update Tuples

- UPDATE <name> SET <field name> = <value> WHERE <condition>
  - Increase all salaries's over $100,000 by 6%, all other receive 5%.
  - Write two update statements:

```
update instructor
set salary = salary * 1.06
where salary > 100000;

update instructor
set salary = salary * 1.05
where salary ≤ 10000;
```

  - The order is important
  - Can be done better using the case statement

SQL Constructs: Insert/Delete/Update Tuples

- UPDATE <name> SET <field name> = <value> WHERE <condition>
  - Increase all salaries's over $100,000 by 6%, all other receive 5%.
  - Can be done better using the case statement

```
update instructor
set salary =
        case
            when salary > 100000
                then salary * 1.06
            when salary <= 100000
                then salary * 1.05
        end;
```

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Querying Basics

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL Querying Basics

- Book Chapters (6th Edition)
  - 3.3

- Key Topics
  - Single-table Queries in SQL
  - Multi-table Queries using Cartesian Product
  - Difference between Cartesian Product and “Natural Join”
  - Careful with using “natural join” keyword

Basic Query Structure

select $A_1, A_2, \ldots, A_n$ ← Attributes or expressions

from $r_1, r_2, \ldots, r_m$ ← Relations (or queries returning tables)

where P ← Predicates

Find the names of all instructors:
select *name*
from *instructor*

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 3.2 Result of "**select** *name* **from** *instructor*".

Basic Query Structure

select $A_1, A_2, \ldots, A_n$ ← Attributes or expressions

from $r_1, r_2, \ldots, r_m$ ← Relations (or queries returning tables)

where P ← Predicates

Find the names of all instructor departments:
select *dept_name*
from *instructor*

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 3.3 Result of "**select** *dept_name* **from** *instructor*".

Basic Query Structure

select A_1, A_2, ..., A_n ← Attributes or expressions

from r_1, r_2, ..., r_m ← Relations (or queries returning tables)

where P ← Predicates

Find the names of all instructors:
select *name*
from *instructor*

Remove duplicates:
select distinct *name*
from *instructor*

Order the output:
select distinct *name*
from *instructor*
order by *name* **asc**

Apply some filters (predicates):
select *name*
from *instructor*
where salary > 80000 **and** dept_name = 'Finance';

Basic Query Constructs

Select all attributes:
select *
from *instructor*

Expressions in the select clause:
select *name, salary < 100000*
from *instructor*

Find the names of all instructors:
select *name*
from *instructor*

More complex filters:
select *name*
from *instructor*
where (dept_name != 'Finance' **and** salary > 75000)
or (dept_name = 'Finance' **and** salary > 85000);

A filter with a subquery:
select *name*
from *instructor*
where dept_name in (**select** dept_name **from**
department **where** budget < 100000);

Python Equivalent

More complex filters:
select *name*
from *instructor*
where (dept_name != 'Finance' **and** salary > 75000)
or (dept_name = 'Finance' **and** salary > 85000);

```
instructor = [ (10101, 'Srinivasan', 65000, 'Comp. Sci.'),
                            (12121, 'Wu', 90000, 'Finance'), …]

for t in instructor:
    if (t[3] != 'Finance' and t[2] > 75000) or
       (t[3] = 'Finance' and t[2] > 85000):
          print(t[1])
```

Basic Query Constructs

Renaming tables or output column names:

select *i.name, i.salary * 2* **as** *double_salary*
from *instructor i*
where *i.salary < 80000* **and** *i.name like '%g_';*

Find the names of all instructors:

select *name*
from *instructor*

More complex expressions:

select *concat(name, concat(', ', dept_name))*
from *instructor;*

Careful with NULLs:

select *name*
from *instructor*
where *salary < 100000* **or** *salary >= 100000;*

Wouldn't return the instructor with NULL salary (if any)

Multi-table Queries

Cartesian product:

select *

from *instructor, teaches*

```
for i in instructor:
    for t in teaches:
        print(i + t)
```

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Physics	95000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Physics	95000	10101	CS-101	1	Fall	2009
12121	Wu	Physics	95000	10101	CS-315	1	Spring	2010
12121	Wu	Physics	95000	10101	CS-347	1	Fall	2009
12121	Wu	Physics	95000	10101	FIN-201	1	Spring	2010
12121	Wu	Physics	95000	15151	MU-199	1	Spring	2010
12121	Wu	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Physics	95000	10101	CS-101	1	Fall	2009
15151	Mozart	Physics	95000	10101	CS-315	1	Spring	2010
15151	Mozart	Physics	95000	10101	CS-347	1	Fall	2009
15151	Mozart	Physics	95000	10101	FIN-201	1	Spring	2010
15151	Mozart	Physics	95000	15151	MU-199	1	Spring	2010
15151	Mozart	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	10101	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 3.6 The Cartesian product of the *instructor* relation with the *teaches* relation.

Multi-table Queries

Use predicates to only select “matching” pairs:

select *

from *instructor i, teaches t*

where *i.ID = t.ID;*

Cartesian product:

select *

from *instructor, teaches*

Identical (in this case) to using a natural join:

select *

from *instructor* **natural join** *teaches;*

Multi-table Queries

```
for i in instructor:
    for t in teaches:
        if i.ID == t.ID:
            print(i + t)
```

With one ID column removed

select *
from *instructor* ***natural join*** *teaches*

ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Figure 3.8 The natural join of the *instructor* relation with the *teaches* relation.

Multi-table Queries

Use predicates to only select "matching" pairs:

select *

from *instructor i, teaches t*

where *i.ID = t.ID;*

Cartesian product:

select *

from *instructor, teaches*

Identical (in this case) to using a natural join:

select *

from *instructor* **natural join** *teaches;*

Natural join does an equality on common attributes – doesn't work here:

select *

from *instructor* **natural join** *advisor;*

Instead can use "on" construct (or where clause as above):

select *

from *instructor* **join** *advisor* **on** *(i_id = id);*

Multi-table Queries

3-Table Query to get a list of instructor-teaches-course information:

select *i.name* **as** *instructor_name, c.title* **as** *course_name*
from *instructor i, course c, teaches*
where *i.ID = teaches.ID and c.course_id = teaches.course_id;*

Beware of unintended common names (happens often)
You may think the following query has the same result as above – it doesn't

select *name*, *title*
from *instructor* **natural join** *course* **natural join** *teaches;*

I prefer avoiding "natural joins" for that reason

Note: On the small dataset, the above two have the same answer, but not on the large dataset. Large dataset has cases where an instructor teaches a course from a different department.

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Aggregates

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL Aggregates

- Book Chapters (6th Edition)
  - 3.7.1-3.7.3

- Key Topics
  - Basic aggregates
  - Aggregation with “grouping”
  - “Having” clause to select among groups

Aggregates

Other common aggregates:
max, min, sum, count, stdev, …

select count (**distinct** *ID*)
from *teaches*
where *semester* = ’Spring’ **and** *year* = 2010

Find the average salary of instructors in the Computer Science
select *avg(**salary)*
from *instructor*
where *dept_name = ‘Comp. Sci’;*

Can specify aggregates in any query.

Find max salary over instructors teaching in S’10
select max(*salary)*
from *teaches* **natural join** *instructor*
where *semester* = ’Spring’ **and** *year* = 2010;

Aggregate result can be used as a scalar.
Find instructors with max salary:
select *
from *instructor*
where *salary* **= (select max***(salary)* **from** *instructor);*

Aggregates

Aggregate result can be used as a scalar.
Find instructors with max salary:

```
select *
from instructor
where salary = (select max(salary) from instructor);
```

Following doesn't work:

```
select *
from instructor
where salary = max(salary);

select name, max(salary)
from instructor
where salary = max(salary);
```

Aggregates: Group By

Split the tuples into groups, and computer the aggregate for each group

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

Aggregates: Group By

Split the tuples into groups, and computer the aggregate for each group

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

```
instructor = [ (10101, ‘Srinivasan’, 65000, ‘Comp. Sci.’),
                            (12121, ‘Wu’, 90000, ‘Finance’), …]


dept_names = set ( [t[3] for t in instructor] )

for dept_name in dept_names:
     this_group = [t for t in instructor if t[3] == dept_name]
     this_avg = average ( [t[2] for t in this_group] )
     print(“{} - {}”.format(dept_name, this_avg))
```

Aggregates: Group By

Find the number of instructors in each department who teach a course in the Spring 2010 semester.

```
Partial Query 1:
select
from instructor natural join teaches
where semester = 'Spring' and year = 2010
```

ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Aggregates: Group By

Find the number of instructors in each department who teach a course in the Spring 2010 semester.

Partial Query 2:

```
select dept_name, count(*)
from instructor natural join teaches
where semester = 'Spring' and year = 2010
group by dept_name
```

Doesn't work – double counts "Katz" who teaches twice in Spring 2010

Final:

```
select dept_name, count(distinct ID)
from instructor natural join teaches
where semester = 'Spring' and year = 2010
group by dept_name
```

Aggregates: Group By

Attributes in the select clause must be aggregates, or must appear in the group by clause. Following wouldn't work

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

"having" can be used to select only some of the groups.

select *dept_name*, **avg** (*salary*)
from *instructor*
group by *dept_name*
having avg(*salary***)** *> 42000;*

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Different Types of Joins, and Set Operations

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL Querying Basics

- Book Chapters (6th Edition)
  - 4.1, 3.5
- Key Topics
  - Outer Joins
  - Anti-joins, Semi-joins
  - Set Operations

Multi-table Queries

Cartesian product:
select *
from *R, S*

A	B
a	1
b	1
c	2

R

×

B	C
1	x
3	y
4	z

S

=

R.A	R.B	S.B	S.C
a	1	1	x
a	1	3	y
a	1	4	z
b	1	1	x
b	1	3	y
b	1	4	z
c	2	1	x
c	2	3	y
c	2	4	z

Multi-table Queries

Natural Join:
select *
from *R natural join S*

A	B
a	1
b	1
c	2

R

⋈

B	C
1	x
3	y
4	z

S

=

R.A	B	S.C
a	1	x
b	1	x

Equivalent to:
select *R.A, R.B, S.C*
from *R, S*
where R.B = S.B

Equivalent to:
select *R.A, R.B, S.C*
from *R* ***join*** *S* ***on*** (R.B = S.B)

Equivalent to:
select *R.A, R.B, S.C*
from *R* ***join*** *S* ***on*** (B)

Outer joins: Why?

Natural Join:
select *
from *R natural join S*

A	B
a	1
b	1
c	2

R

⋈

B	C
1	x
3	y
4	z

S

=

R.A	B	S.C
a	1	x
b	1	x

Often need the ”non-matching” tuples in the result

"Left" Outerjoin

```
select *
from R natural left outer join S
```

A	B
a	1
b	1
c	2

R

⋈

B	C
1	x
3	y
4	z

S

=

R.A	B	S.C
a	1	x
b	1	x
c	2	NULL

NULL is a keyword in SQL

```
select *
from R left outer join S on (R.B = S.B)
```

=

```
select *
from R natural join S on (R.B = S.B)
UNION ALL
select R.A, R.B, NULL
from R where R.B NOT IN
    (select S.B from S)
```

"Right" Outerjoin

select *
from *R right natural outer join S*

A	B
a	1
b	1
c	2

R

⟖

B	C
1	x
3	y
4	z

S

=

R.A	B	S.C
a	1	x
b	1	x
NULL	3	y
NULL	4	z

select *
from *R right outer join S on (R.B = S.B)*

select *
from *R natural join S on (R.B = S.B)*
UNION ALL
***select NULL,** S.B, S.C*
from** S **where** S.B **NOT IN
***(select** R.B **from** R)*

"Full" Outerjoin

select *
from *R natural full outer join S*

A	B
a	1
b	1
c	2

R

⋈

B	C
1	x
3	y
4	z

S

=

R.A	B	S.C
a	1	x
b	1	x
c	2	NULL
NULL	3	y
NULL	4	z

select *
from *R full outer join S on (R.B = S.B)*

Semi-joins

R SEMI-JOIN S = tuples of R that do have a “match” in S

Not an SQL keyword, but useful concept to understand – often implemented in database systems as an operator

A	B
a	1
b	1
c	2

$\ltimes$

B	C
1	x
3	y
4	z

=

R.A	R.B
a	1
b	1

Can be written in SQL as:

```
select *
from R
where B in (select B from S);
```

Semi-joins

R SEMI-JOIN S = tuples of R that do have a “match” in S

Not an SQL keyword, but useful concept to understand – often implemented in database systems as an operator

A	B
a	1
b	1
c	2

⋉

B	C
1	x
3	y
4	z

=

R.A	R.B
a	1
b	1

A	B
a	1
b	1
c	2

⋊

B	C
1	x
3	y
4	z

=

S.B	S.C
1	x

Anti-joins

R ANTI-JOIN S = tuples of R that do NOT have a “match” in S

Not an SQL keyword, but useful concept to understand – often implemented in database systems as an operator

A	B
a	1
b	1
c	2

▷

B	C
1	x
3	y
4	z

=

R.A	R.B
c	2

Can be written in SQL as:

```
select *
from R
where B not in (select B from S);
```

Anti-joins

R ANTI-JOIN S = tuples of R that do NOT have a “match” in S

Not an SQL keyword, but useful concept to understand – often implemented in database systems as an operator

A	B
a	1
b	1
c	2

▷

B	C
1	x
3	y
4	z

=

R.A	R.B
c	2

A	B
a	1
b	1
c	2

◁

B	C
1	x
3	y
4	z

=

S.B	S.C
3	y
4	z

Set operations

Find courses that ran in Fall 2009 or Spring 2010

(**select** *course_id* **from** *section* **where** *semester* = 'Fall' **and** *year* = 2009)
union
(**select** *course_id* **from** *section* **where** *semester* = 'Spring' **and** *year* = 2010);

In both:

(**select** *course_id* **from** *section* **where** *semester* = 'Fall' **and** *year* = 2009)
intersect
(**select** *course_id* **from** *section* **where** *semester* = 'Spring' **and** *year* = 2010);

In Fall 2009, but not in Spring 2010:

(**select** *course_id* **from** *section* **where** *semester* = 'Fall' **and** *year* = 2009)
except
(**select** *course_id* **from** *section* **where** *semester* = 'Spring' **and** *year* = 2010);

Set operations: Duplicates

Union/Intersection/Except eliminate duplicates in the answer (the other SQL commands don't) (e.g., try 'select dept_name from instructor').

Can use "union all" to retain duplicates.

NOTE: The duplicates are retained in a systematic fashion (for all SQL operations)

Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:

- *m + n* times in *r* **union all** *s*
- min(*m,n)* times in *r* **intersect all** *s*
- max(0, *m – n)* times in *r* **except all** *s*

CMSC424: Database Design

Module: Relation Model + SQL

Sets vs Multisets;
Relational Algebra

Instructor: Amol Deshpande
amol@umd.edu

Relational Algebra

- Book Chapters (6th Edition)
  - 2.5, 2.6, 6.1.1-6.1.3 (expanded treatment of 2.5, 2.6)
  - Multiset Relational Algebra Paragraph (Section 6.1, page 238)

- Key Topics
  - Relational query languages and what purpose they serve
  - Sets vs Multisets

Procedural vs Declarative Languages

- Procedural/imperative query languages
  - Support a set of data-oriented operations
  - Usually need to specify the sequence of steps to be taken to get to the output
  - Often map one-to-one with the physical operators that are implemented
    - Large gap between those two ➔ more opportunities to optimize
- Declarative query languages
  - Specify the desired outcome, typically as a function over the inputs
- A different issue that how ”high-level” or abstract the language is
- Most languages today are somewhere in-between
  - SQL is more declarative than procedural

Relational Query Languages

- Example schema: *R(A, B)*
- Practical languages
  - SQL
    - select A from R where B = 5;
  - Datalog (sort of practical)
    - q(A) :- R(A, 5)
- Formal languages
  - Relational algebra

    $\pi_A (\sigma_{B=5} (R))$
  - Tuple relational calculus

    $\{ t : \{A\} \mid \exists s : \{A, B\} (R(A, B) \wedge s.B = 5) \}$
  - Domain relational calculus
    - Similar to tuple relational calculus

Relational Algebra

- Procedural language

- Six basic operators
  - select
  - project
  - union
  - set difference
  - Cartesian product
  - rename

- The operators take one or more relations as inputs and give a new relation as a result.

Select Operation

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

SQL Equivalent:

```
select distinct *
from r
where A = B and D > 5
```

Unfortunate naming confusion

Project

Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\pi_{A,D}(r)$

A	D
α	7
α	7
β	3
β	10

A	D
α	7
β	3
β	10

SQL Equivalent:

```
select distinct A, D
from r
```

Set Union, Difference

Relation r, s

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r ∪ s:

A	B
α	1
α	2
β	1
β	3

r – s:

A	B
α	1
β	1

Must be compatible schemas

What about intersection ?

Can be derived

$r \cap s = r - (r - s);$

SQL Equivalent:

```
select * from r
union/except/intersect
select * from s;
```

This is one case where duplicates are removed.

Cartesian Product

Relation r, s

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

$r \times s$:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

SQL Equivalent:

```
select distinct *
from r, s
```

Does not remove duplicates.

Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.

  Example:

  $$\rho_X(E)$$

  returns the expression *E* under the name *X*

  If a relational-algebra expression *E* has arity *n*, then

  $$\rho_{X\ (A1, A2, \ldots, An)}(E)$$

  returns the result of expression *E* under the name *X*, and with the attributes renamed to $A_1, A2, \ldots, An$.

Relational Algebra

- Those are the basic operations
- What about SQL Joins ?
  - Compose multiple operators together

$$\sigma_{A=C}(r \times s)$$

- Additional Operations
  - Set intersection
  - Natural join
  - Division
  - Assignment

Additional Operators

- Set intersection (∩)
  - r ∩ s = r – (r – s);
  - SQL Equivalent: intersect

- Assignment (←)
  - A convenient way to right complex RA expressions
  - Essentially for creating "temporary" relations
    - $temp1 \leftarrow \prod_{R\text{-}S} (r)$
  - SQL Equivalent: "create table as…"

Additional Operators: Joins

- Natural join ($\bowtie$)
  - A Cartesian product with equality condition on common attributes
  - Example:
    - if *r* has schema *R(A, B, C, D)*, and if *s* has schema *S(E, B, D)*
    - Common attributes: *B* and *D*
    - Then:

$$r \bowtie s = \Pi_{r.A,\ r.B,\ r.C,\ r.D,\ s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$

- SQL Equivalent:
  - select r.A, r.B, r.C, r.D, s.E from r, s where r.B = s.B and r.D = s.D,
    OR
  - select * from r natural join s

Additional Operators: Joins

- Equi-join
  - A join that only has equality conditions

- Theta-join ($\bowtie_\theta$)
  - $r \bowtie_\theta s = \sigma_\theta(r \times s)$

- Left outer join ($⟕$)
  - Say *r(A, B), s(B, C)*
  - We need to somehow find the tuples in *r* that have no match in *s*
  - Consider: $(r - \pi_{r.A,\, r.B}(r \bowtie s))$

  - We are done:

$$(r \bowtie s) \quad \cup \quad \rho_{temp\ (A,\, B,\, C)} \left((r - \pi_{r.A,\, r.B}(r \bowtie s)) \times \{(\text{NULL})\} \right)$$

Additional Operators: Join Variations

- Tables: r(A, B), s(B, C)

name	Symbol	SQL Equivalent	RA expression
cross product	$\times$	select * from r, s;	$r \times s$
natural join	$\bowtie$	natural join	$\pi_{r.A,\ r.B,\ s.C}\sigma_{r.B\ =\ s.B}(r\ x\ s)$
theta join	$\bowtie_\theta$	from .. where θ;	$\sigma_\theta(r\ x\ s)$
equi-join	$\bowtie_\theta$ *(theta must be equality)*		
left outer join	$r \ ⟕ \ s$	left outer join (with "on")	(see previous slide)
full outer join	$r \ ⟗ \ s$	full outer join (with "on")	–
(left) semijoin	$r \ltimes s$	none	$\pi_{r.A,\ r.B}(r \bowtie s)$
(left) antijoin	$r \triangleright s$	none	$r - \pi_{r.A,\ r.B}(r \bowtie s)$

Example Query

- Find the largest salary in the university
  - Step 1: find instructor salaries that are less than some other instructor salary (i.e. not maximum)
    - using a copy of *instructor* under a new name *d*
    - $\Pi_{instructor.salary} (\sigma_{instructor.salary < d.salary} (instructor \times \rho_d (instructor)))$
  - Step 2: Find the largest salary
    - $\Pi_{salary} (instructor) - \Pi_{instructor.salary} (\sigma_{instructor.salary < d.salary} (instructor \times \rho_d (instructor)))$

Example Queries

- Find the names of all instructors in the Physics department, along with the *course_id* of all courses they have taught
  - Query 1

$$\Pi_{instructor.ID,course\_id}(\sigma_{dept\_name=\text{“Physics”}}(\sigma_{instructor.ID=teaches.ID}(instructor \times teaches)))$$

  - Query 2

$$\Pi_{instructor.ID,course\_id}(\sigma_{instructor.ID=teaches.ID}(\sigma_{dept\_name=\text{“Physics”}}(instructor) \times teaches))$$

Duplicates

- By definition, *relations* are *sets*
  - So → No duplicates allowed
- Problem:
  - Not practical to remove duplicates after every operation
  - Why ?
- So…
  - SQL by default does not remove duplicates
- SQL follows *bag* semantics, not *set* semantics
  - Implicitly we keep count of number of copies of each tuple

Formal Semantics of SQL

- RA can only express `SELECT DISTINCT` queries

- To express SQL, must extend RA to a bag algebra
  → *Bags (aka: multisets) like sets, but can have duplicates*

e.g: {5, 3, 3}

e.g: homes =

cname	ccity
Johnson	Brighton
Smith	Perry
Johnson	Brighton
Smith	R.H.

- Next: will define RA*: a bag version of RA

Formal Semantics of SQL: RA*

1. σ^*_p (r): *preserves copies in* r

e.g: $\sigma^*_{city = Brighton}$ (homes) =

cname	ccity
Johnson	Brighton
Johnson	Brighton

2. $\pi^*_{A1, \ldots, An}$ (r): *no duplicate elimination*

e.g: π^*_{cname} (homes) =

cname
Johnson
Smith
Johnson
Smith

Formal Semantics of SQL: RA*

3. $r \cup^* s$: *additive union*

A	B
1	α
1	α
2	β

r

$\cup^*$

A	B
2	β
3	α
1	α

s

$=$

A	B
1	α
1	α
2	β
2	β
3	α
1	α

4. r -* s: *bag difference*

e.g: r -* s =

A	B
1	α

s -* r =

A	B
3	α

Formal Semantics of SQL: RA*

5. r $\times^*$ s: *cartesian product*

A	B
1	α
1	α
2	β

$\times^*$

C
+
-

$=$

A	B	C
1	α	+
1	α	-
1	α	+
1	α	-
2	β	+
2	β	-

Formal Semantics of SQL

Query:

```
SELECT      a1, ….., an
FROM        r1, ….., rm
WHERE       p
```

Semantics: $\pi^*_{A1, \ldots, An} (\sigma^*_p (r_1 \times^* \ldots \times^* r_m))$ (1)

Query:

```
SELECT DISTINCT   a1, ….., an
FROM              r1, ….., rm
WHERE             p
```

Semantics: *What is the only operator to change in (1)?*

$\pi_{A1, \ldots, An} (\sigma^*_p (r_1 \times^* \ldots \times^* r_m))$ (2)

Set/Bag Operations Revisited

- Set Operations
  - UNION ≡ ∪
  - INTERSECT ≡ ∩
  - EXCEPT ≡ -

Bag Operations

UNION ALL ≡ ∪*
INTERSECT ALL ≡ ∩*
EXCEPT ALL ≡ -*

Duplicate Counting:

Given m copies of t in r, n copies of t in s, how many copies of t in:

`r UNION ALL s?` A: $m + n$

`r INTERSECT ALL s?` *A: min (m, n)*

`r EXCEPT ALL s?` *A: max (0, m-n)*

Operations RA Doesn't Support

- … but are supported in most data systems today
- Grouping
  - Going from: {(1, a), (1, b), (2, a)} → {(1, (a, b)), (2, (a))
- Aggregates
- Nesting and unnesting

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Nested Subqueries

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL Nested Subqueries

- Book Chapters (6th Edition)
  - 3.8

- Key Topics
  - Subqueries
  - Boolean operations with Subqueries

Nested Subqueries

- Provides a way to "compose" simpler queries into more complex tasks
- **Goal 1:** Construct scalars or sets or tables that can be used in other queries
  - compute the max salary across instructor and use it in elsewhere

```
select *
from instructor
where salary = (select max(salary) from instructor);
```

  - create a set with all courseids that start with "S" and use it to find who takes those courses

```
select *
from takes
where course_id in (select course_id
                    from courses
                    where title like 'S%')
```

Nested Subqueries

- Provides a way to "compose" simpler queries into more complex tasks
- **Goal 1:** Construct scalars or sets or tables that can be used in other queries
  - Construct a table with average salary in each department

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
        from instructor
        group by dept_name)
where avg_salary > 42000;
```

  - Preferable to use WITH

```
with temp as (select dept_name, avg(salary) as avg_salary
                from instructor
                group by dept_name)
select dept_name, avg_salary
from temp
where avg_salary > 42000;
```

Note: can be done using "having" more simply

Nested Subqueries

- Provides a way to “compose” simpler queries into more complex tasks
- **Goal 1:** Construct scalars or sets or tables that can be used in other queries
  - Find all course_ids taught in Spring 2010, so we can check which were also offered in Fall 2009

```
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);
```

Nested Subqueries

- Provides a way to "compose" simpler queries into more complex tasks
- **Goal 2:** As "functions" of tuples that require other relations
  - e.g., for each instructor, return the string length of name

```
select id, len(name)
from instructor;
```

  - but what if we need to find the # of courses they have taught?

```
select id, (select count(*)
            from teaches
            where teaches.id = instructor.id)
from instructor;
```

Think of this like a function that takes in a instructor.id as input

Nested Subqueries

- Provides a way to “compose” simpler queries into more complex tasks
- **Goal 2:** As “functions” of tuples that require other relations
  - A function to count the number of instructors in a department

```
select dept_name,
        (select count(*)
          from instructor
          where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

Correlated vs Uncorrelated Subqueries

```
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);
```

Uncorrelated subquery – the subquery makes no reference to the enclosing queries, and can be evaluated by itself

```
select dept_name,
        (select count(*)
          from instructor
          where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

* Correlated subquery – the subquery has a reference to the enclosing query
* For every tuple of department, the subquery returns a different result

Set Membership: ”IN” and “NOT IN”

```
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);
```

Can also be written using Set Intersection

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

Set Membership: ”IN” and “NOT IN”

Can do this with “tuples” as well:

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in (select course_id, sec_id, semester, year
                                              from teaches
                                              where teaches.ID= 10101);
```

Set Comparisons

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept_name = 'Biology');
```

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept_name = 'Biology');
```

Testing for Empty Results

```
select course_id
from section as S
where semester = 'Fall' and year= 2009 and
        exists (select *
                from section as T
                where semester = 'Spring' and year= 2010 and
                        S.course_id= T.course_id);
```

Also: “Not Exists”

Uniqueness

```
select T.course_id
from course as T
where unique (select R.course_id
                from section as R
                where T.course_id= R.course_id and
                      R.year = 2009);
```

There are usually alternatives to using these constructs (e.g., group by + having instead of “unique”), but these can often make queries more readable and more compact

"With" Clause

Used for creating "temporary" tables within the context of the query

```
with dept_total (dept_name, value) as
        (select dept_name, sum(salary)
        from instructor
        group by dept_name),
dept_total_avg(value) as
        (select avg(value)
        from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

(1) Find classrooms over capacity, i.e., with more students than capacity.

(2) Find students who haven’t taken the pre-requisite for a course they did take

(3) Instructors who taught 4 courses in a single day in a single semester, along with the day and the semester

(4) Create a pivot table, with rows being course_ids, and columns being Grades (let's assume "grade" takes values A, B, C, D, F)

So we want something like:

	A	B	C	D	F
CMSC424	120	90	100	20	20

(5) For each course, find the most lenient instructor, i.e., the instructor who appears to have the highest average grade

(6) Find pairs of students with identical courses taken throughout -- output should be a table with two studentid columns

CMSC424: Database Design

Module: Relation Model + SQL

SQL: NULLs

Instructor: Amol Deshpande
amol@cs.umd.edu

SQL: NULLs

- Book Chapters (6th Edition)
  - 3.6, 3.7.4

- Key Topics
  - Operating with NULLs
  - ”Unknown” as a new Boolean value
  - Operating with UNKNOWNs
  - Aggregates and NULLs

SQL: Nulls

Can cause headaches for query semantics as well as query processing and optimization)

Can be a value of any attribute

e.g: branch =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

What does this mean?

(unknown) We don't know Waltham's assets?

(inapplicable) Waltham has a special kind of account without assets

(withheld) We are not allowed to know

SQL: Nulls

Arithmetic Operations with `Null`

`n + NULL = NULL` (similarly for all *arithmetic ops*: `+, -, *, /, mod, …`)

e.g: branch =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

```
SELECT bname, assets * 2 as a2
FROM branch
```

=

bname	a2
Downtown	18M
Perry	3.4M
Mianus	.8M
Waltham	NULL

Counter-intuitive: NULL * 0 = NULL

SQL: Nulls

Boolean Operations with `Null`

`n < NULL = UNKNOWN` (similarly for all *boolean ops*: `>, <=, >=, <>, =, …`)

e.g: branch =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

assets < 10M will evaluate to UNKNOWN for the last tuple

But what about:

(assets < 10M) or (bcity = 'Boston') ?

(assets < 10M) and (bcity = 'Boston')?

SQL: Unknown

```
FALSE OR UNKNOWN = UNKNOWN
TRUE AND UNKNOWN = UNKNOWN
FALSE AND UNKNOWN = FALSE
TRUE OR UNKNOWN = TRUE
```

```
UNKNOWN OR UNKNOWN = UNKNOWN
UNKNOWN AND UNKNOWN = UNKNOWN
NOT (UNKNOWN) = UNKNOWN
```

Intuition: substitute each of TRUE, FALSE for unknown. If different answer results, results is unknown

Values	Expression	Result
x = NULL, y = 10	(x < 10) and (y = 20)	UNKNOWN and FALSE = FALSE
x = NULL, y = 10	(x is NULL) and (y = 20)	TRUE and FALSE = FALSE
x = NULL, y = 10	(x < 10) and (y = 10)	UNKNOWN and TRUE = UNKNOWN
x = NULL, y = 10	(x < 10) is UNKNOWN	TRUE
x = NULL, y = 10	((x < 10) is UNKNOWN) and (y = 10)	TRUE AND TRUE = TRUE

UNKNOWN tuples are not included in final result

Aggregates and NULLs

Given

branch =

bname	bcity	assets
Downtown	Boston	9M
Perry	Horseneck	1.7M
Mianus	Horseneck	.4M
Waltham	Boston	NULL

Aggregate Operations

```
SELECT SUM (assets) =
FROM branch
```

SUM
11.1 M

`NULL` *is ignored for SUM*

Same for `AVG` *(3.7M),* `MIN` *(0.4M),* `MAX` *(9M)*

Also for COUNT(assets) -- returns 3

But `COUNT (*)` *returns*

COUNT
4

Aggregates and NULLs

Given

branch =

bname	bcity	assets

```
SELECT SUM (assets) =
FROM branch
```

SUM
NULL

- *Same as* `AVG, MIN, MAX`
- *But* `COUNT (assets)` *returns*

COUNT
0

CMSC424: Database Design

Module: Relation Model + SQL

Keys

Instructor: Amol Deshpande
amol@cs.umd.edu

Relational Model: Keys

- Book Chapters (6th Edition)
  - 2.3

- Key Topics
  - Keys as a mechanism to uniquely identify tuples in a relation
  - Super key vs Candidate key vs Primary key
  - Foreign keys and Referential Integrity
  - How to identify keys of a relation

Keys

- Let $K \subseteq R$
- K is a **superkey** of R if values for K are sufficient to identify a unique tuple of any possible relation r(R)
  - *Example: {ID} and {ID,name} are both superkeys of instructor.*
- Superkey K is **a candidate key** if K is minimal (i.e., no subset of it is a superkey)
  - *Example: {ID} is a candidate key for Instructor*
- One of the candidate keys is selected to be the **primary key**
  - Typically one that is small and immutable (doesn't change often)
- Primary key typically highlighted (e.g., underlined)

Tables in a University Database

classroom(building, room_number, capacity)

department(dept_name, building, budget)

course(course_id, title, dept_name, credits)

instructor(ID, name, dept_name, salary)

Tables in a University Database

takes(ID, course_id, sec_id, semester, year, grade)

What about ID, course_id?

No. May repeat:

("1011049", "CMSC424", "101", "Spring", 2014, D)

("1011049", "CMSC424", "102", "Fall", 2015, null)

What about ID, course_id, sec_id?

May repeat:

("1011049", "CMSC424", "101", "Spring", 2014, D)

("1011049", "CMSC424", "101", "Fall", 2015, null)

What about ID, course_id, sec_id, semester?

Still no: ("1011049", "CMSC424", "101", "Spring", 2014, D)

("1011049", "CMSC424", "101", "Spring", 2015, null)

Tables in a University Database

classroom(building, room_number, capacity)

department(dept_name, building, budget)

course(course_id, title, dept_name, credits)

instructor(ID, name, dept_name, salary)

section(course_id, sec_id, semester, year, building,
room_number, time_slot_id)

teaches(ID, course_id, sec_id, semester, year)

student(ID, name, dept_name, tot_cred)

takes(ID, course_id, sec_id, semester, year, grade)

advisor(s_ID, i_ID)

time_slot(time_slot_id, day, start_time, end_time)

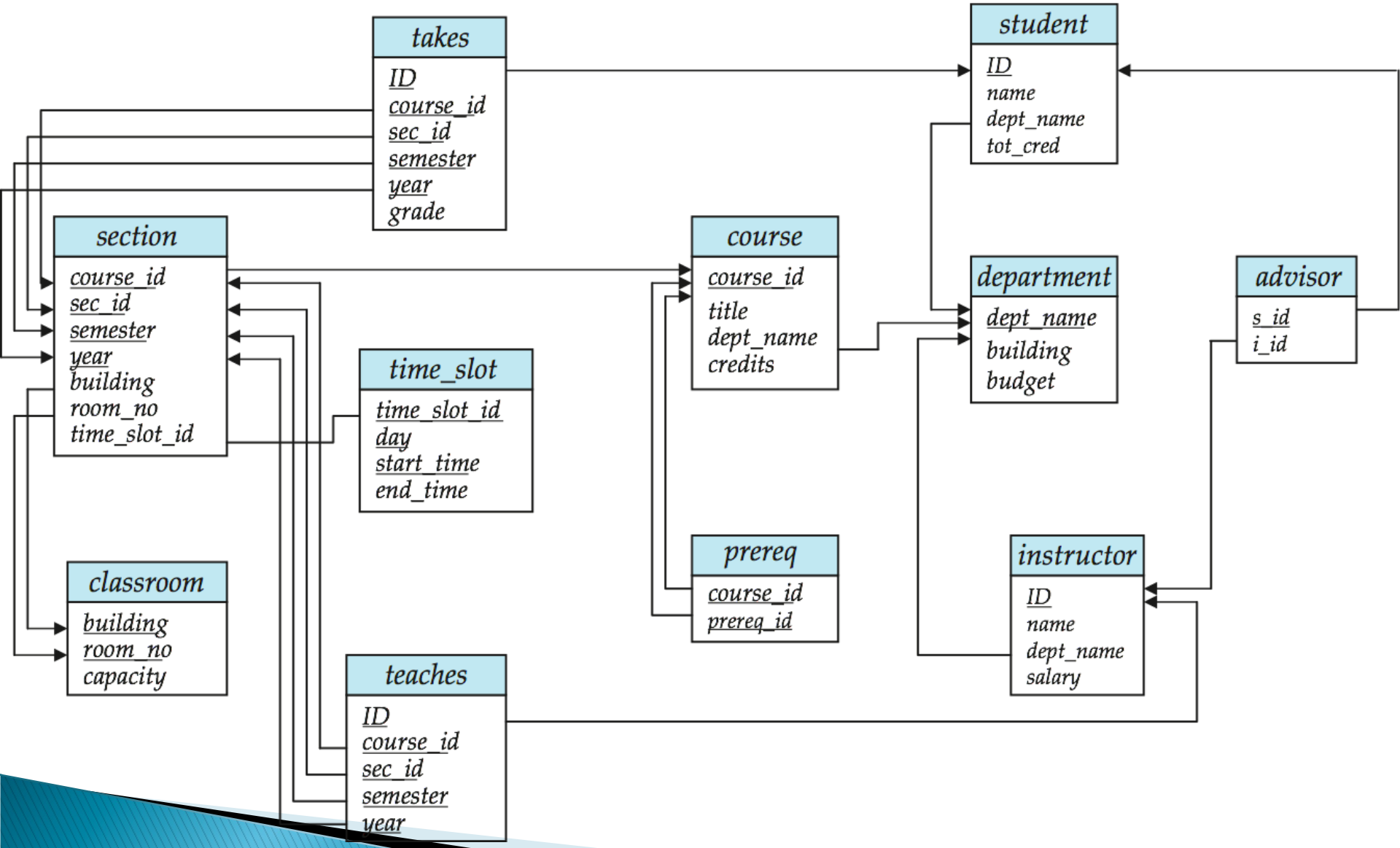
prereq(course_id, prereq_id)

Keys

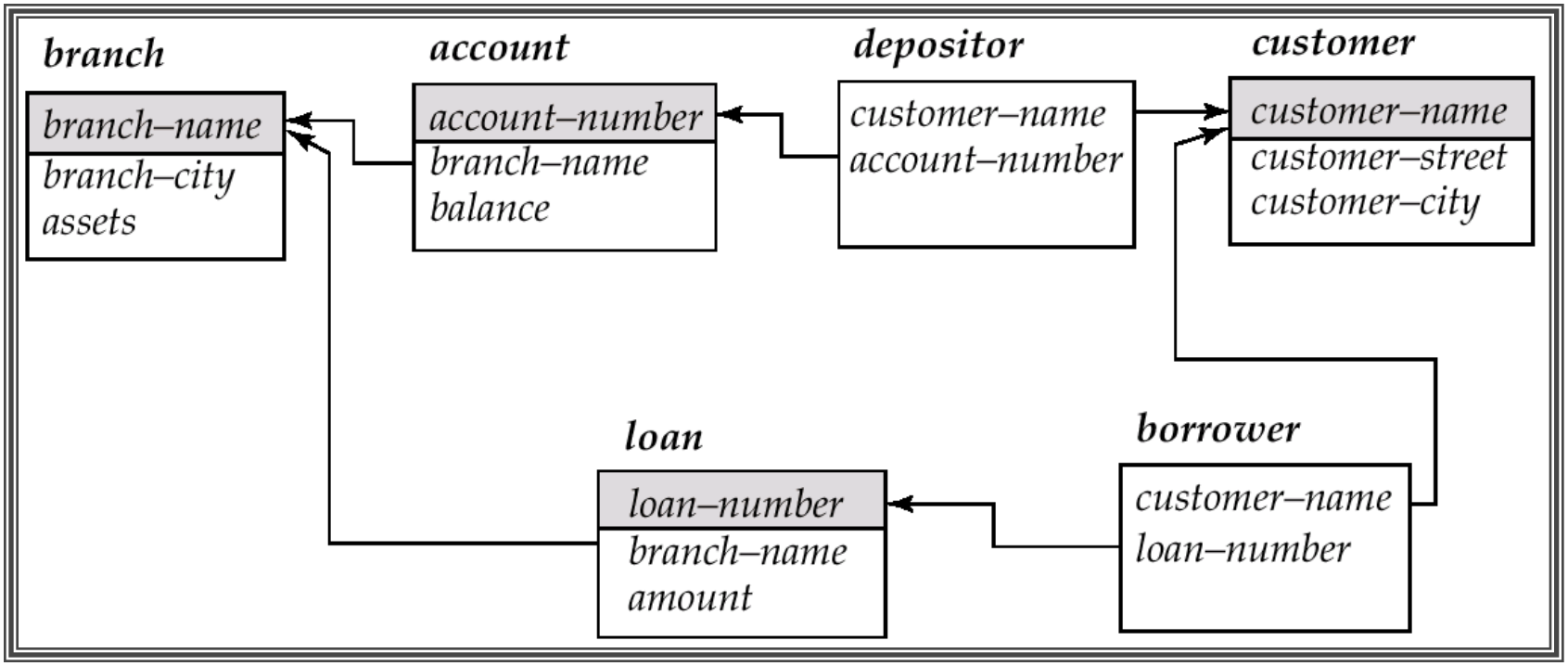
- **Foreign key:** Primary key of a relation that appears in another relation
  - {ID} from *student* appears in *takes, advisor*
  - *student* called ***referenced*** relation
  - *takes* is the ***referencing*** relation
  - Typically shown by an arrow from referencing to referenced

- **Foreign key constraint**: the tuple corresponding to that primary key must exist
  - Imagine:
    - Tuple: (‘student101’, ‘CMSC424’) in *takes*
    - But no tuple corresponding to ‘student101’ in *student*
  - Also called ***referential integrity constraint***

Schema Diagram for University Database



Schema Diagram for the Banking Enterprise



Examples

- Married(person1_ssn, person2_ssn, date_married, date_divorced)
- Account(cust_ssn, account_number, cust_name, balance, cust_address)
- RA(student_id, project_id, superviser_id, appt_time, appt_start_date, appt_end_date)
- Person(Name, DOB, Born, Education, Religion, …)
  - *Information typically found on Wikipedia Pages*

Examples

- Married(person1_ssn, person2_ssn, date_married, date_divorced)
- Account(cust_ssn, account_number, cust_name, balance, cust_address)
  - If a single account per customer, then: cust_ssn
  - Else: (cust_ssn, account_number)
    - In the latter case, this is not a good schema because it requires repeating information
- RA(student_id, project_id, superviser_id, appt_time, appt_start_date, appt_end_date)
  - Could be smaller if there are some restrictions – requires some domain knowledge of the data being stored
- Person(Name, DOB, Born, Education, Religion, …)
  - *Information typically found on Wikipedia Pages*
  - *Unclear what could be a primary key here: you could in theory have two people who match on all of those*

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Functions, Procedures, Recursive Queries, Triggers, Authorization, …

Instructor: Amol Deshpande
amol@cs.umd.edu

Miscellaneous SQL

- Book Chapters (7th Edition)
  - Sections 4.7, 5.2, 5.3, 5.4, 5.5.1
  - Mostly at a high level
  - See Assignment 2

- Key topics
  - Ranking over relations or results
  - Recursion in SQL (makes SQL Turing Complete)
  - Windows
  - Unnest and Lateral
  - Functions and Procedures
  - Triggers
  - Authorization

Ranking

- Goal: Rank tuples by some property

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

ID	Salary	Rank
15151	95000	1
83821	92000	2
12121	90000	3
33456	87000	4
98345	80000	5
76543	80000	5
45565	75000	7
76766	72000	8
10101	65000	9
58583	62000	10
32343	60000	11
15151	40000	12

In DENSE_RANK, we would not skip 6 -- so the last rank would be 11.

Ranking

- Ranking is done in conjunction with an order by specification.

- Basic SQL:

```
select ID, Salary, (1 + (select count(*)
                         from Instructor B
                         where B.Salary > A.Salary)) as s_rank
from Instructor A
order by s_rank;
```

Variation 1

```
select ID, Salary, ((select count(*)
                         from Instructor B
                         where B.Salary >= A.Salary)) as s_rank
from Instructor A
order by s_rank;
```

We would skip
5 instead of 6
i.e., after 4, the next rank
would be 6 (for two tuples)

Ranking

- Ranking is done in conjunction with an order by specification.

- Basic SQL:

```
select ID, Salary, (1 + (select count(*)
                         from Instructor B
                        where B.Salary > A.Salary)) as s_rank
from Instructor A
order by s_rank;
```

Variation 2

```
select ID, Salary, (1 + (select count(distinct Salary)
                         from Instructor B
                        where B.Salary > A.Salary)) as s_rank
from Instructor A
order by s_rank;
```

Equivalent to DENSE_RANK
We would not skip a rank
Two tuples with rank = 5
Next tuple with rank = 6

Ranking

- Ranking is done in conjunction with an order by specification.

- Basic SQL:

```
select ID, Salary, (1 + (select count(*)
                         from Instructor B
                        where B.Salary > A.Salary)) as s_rank
from Instructor A
order by s_rank;
```

Variation 3

```
select ID, Salary, (1 + (select count(Salary)
                         from Instructor B
                        where B.Salary < A.Salary)) as s_rank
from Instructor A
order by s_rank;
```

Opposite rank order
Tuple with lower salary has rank = 1

Ranking

- Ranking is done in conjunction with an order by specification.

- Basic SQL:

```
select ID, Salary, (1 + (select count(*)
                         from Instructor B
                         where B.Salary > A.Salary)) as s_rank
from Instructor A
order by s_rank;
```

SQL Support

```
select ID, Salary, rank() over (order by salary desc) as s_rank
        from instructor
        order by s_rank
```

PostgreSQL support dense_rank() as well

Ranking

- Ranking is done in conjunction with an order by specification.

- Basic SQL:

```
select ID, Salary, (1 + (select count(*)
                         from Instructor B
                         where B.Salary > A.Salary)) as s_rank
from Instructor A
order by s_rank;
```

SQL Support -- can do subqueries and rank on results

```
select ID, Salary, rank() over (order by
        (select count(*) from teaches where teaches.id = instructor.id)
                                    desc) as s_rank
       from instructor
       order by s_rank
```

Window Functions

- Similar to “Group By” – allows a calculation over “related” tuples
- Unlike aggregates, does not “group” them – rather rows remain separate from each other

Goal: Associate each instructor row with average salary across all instructors from that department

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

```
  id   |  salary   | dept_name  |         avg
-------+-----------+------------+--------------------
 10101 | 65000.00 | Comp. Sci. | 77333.333333333333
 12121 | 90000.00 | Finance    | 85000.000000000000
 15151 | 40000.00 | Music      | 40000.000000000000
 22222 | 95000.00 | Physics    | 91000.000000000000
 32343 | 60000.00 | History    | 61000.000000000000
 33456 | 87000.00 | Physics    | 91000.000000000000
 45565 | 75000.00 | Comp. Sci. | 77333.333333333333
 58583 | 62000.00 | History    | 61000.000000000000
 76543 | 80000.00 | Finance    | 85000.000000000000
 76766 | 72000.00 | Biology    | 72000.000000000000
 83821 | 92000.00 | Comp. Sci. | 77333.333333333333
 98345 | 80000.00 | Elec. Eng. | 80000.000000000000
(12 rows)
```

Window Functions

- Similar to “Group By” – allows a calculation over “related” tuples
- Unlike aggregates, does not “group” them – rather rows remain separate from each other

Goal: Associate each instructor row with average salary across all instructors from that department

```
select ID, Salary, Dept_name, (select avg(salary)
                                 from Instructor B
                                 where B.dept_name = A. dept_name)
    from Instructor A;
```

Common enough task to simplify the syntax:

```
select ID, Salary, Dept_name, avg(salary) over (partition by dept_name)
from Instructor;
```

Benefits: (1) Simpler syntax
(2) Easier to optimize

Window Functions

- Similar to “Group By” – allows a calculation over “related” tuples
- Unlike aggregates, does not “group” them – rather rows remain separate from each other

Goal: Associate each instructor row the “rank” for that instructor by salary within that department

select *ID, Salary*, Dept_name, **rank() over (partition by** dept_name order by salary desc**) from** *Instructor;*

id	name	salary	dept_name	rank
76766	Crick	72000.00	Biology	1
83821	Brandt	92000.00	Comp. Sci.	1
45565	Katz	75000.00	Comp. Sci.	2
10101	Srinivasan	65000.00	Comp. Sci.	3
98345	Kim	80000.00	Elec. Eng.	1
12121	Wu	90000.00	Finance	1
76543	Singh	80000.00	Finance	2
58583	Califieri	62000.00	History	1
32343	El Said	60000.00	History	2
15151	Mozart	40000.00	Music	1
22222	Einstein	95000.00	Physics	1
33456	Gold	87000.00	Physics	2

(12 rows)

Recursion in SQL

- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

Makes SQL Turing Complete (i.e., you can write any program in SQL)

But: Just because you can, doesn't mean you should

Recursion in SQL

- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

course_id	prereq_id
CMSC424	CMSC351
CMSC424	CMSC330
CMSC351	CMSC250
CMSC250	CMSC216
CMSC216	CMSC132
CMSC132	CMSC131

Iteration 1: rec_prereq =

course_id	prereq_id
CMSC424	CMSC351
CMSC424	CMSC330
CMSC351	CMSC250
CMSC250	CMSC216
CMSC216	CMSC132
CMSC132	CMSC131

Iteration 2: rec_prereq =

course_id	prereq_id
CMSC424	CMSC351
CMSC424	CMSC330
CMSC351	CMSC250
CMSC250	CMSC216
CMSC216	CMSC132
CMSC132	CMSC131
CMSC424	CMSC250
CMSC351	CMSC216
CMSC250	CMSC132
CMSC216	CMSC131

Recursion in SQL

- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

course_id	prereq_id
CMSC424	CMSC351
CMSC424	CMSC330
CMSC351	CMSC250
CMSC250	CMSC216
CMSC216	CMSC132
CMSC132	CMSC131

Iteration 1: rec_prereq =

course_id	prereq_id
CMSC424	CMSC351
CMSC424	CMSC330
CMSC351	CMSC250
CMSC250	CMSC216
CMSC216	CMSC132
CMSC132	CMSC131

Iteration 3: rec_prereq =

course_id	prereq_id
CMSC424	CMSC351
CMSC424	CMSC330
CMSC351	CMSC250
CMSC250	CMSC216
CMSC216	CMSC132
CMSC132	CMSC131
CMSC424	CMSC250
CMSC351	CMSC216
CMSC250	CMSC132
CMSC216	CMSC131
CMSC424	CMSC216
CMSC424	CMSC132
…	…
…	…

Arrays and Unnest

- Allows taking an array attribute and separate out its elements

```
CREATE TABLE sal_emp
        (name text,
          pay_by_quarter integer[],
          schedule text[][] );


INSERT INTO sal_emp VALUES ('Bill', '{10000, 10000, 10000, 10000}',
            '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp VALUES ('Carol', '{20000, 25000, 25000, 25000}',
            '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

```
university=# select * from sal_emp;
 name  |      pay_by_quarter       |                  schedule
-------+---------------------------+--------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

https://www.postgresql.org/docs/current/arrays.html

Arrays and Unnest

- Allows taking an array attribute and separate out its elements

```
university=# select * from sal_emp;
  name  |      pay_by_quarter       |                  schedule
-------+---------------------------+-------------------------------------
  Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
  Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

```
university=# SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];
 name
-------
 Carol
(1 row)
```

```
university=# SELECT pay_by_quarter[3] FROM sal_emp;
 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

https://www.postgresql.org/docs/current/arrays.html

Arrays and Unnest

- Allows taking an array attribute and separate out its elements

```
university=# select * from sal_emp;
 name  |      pay_by_quarter       |                  schedule
-------+---------------------------+-------------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

```
university=# select name, unnest(pay_by_quarter), schedule from sal_emp;
 name  | unnest |                  schedule
-------+--------+-------------------------------------------
 Bill  |  10000 | {{meeting,lunch},{training,presentation}}
 Bill  |  10000 | {{meeting,lunch},{training,presentation}}
 Bill  |  10000 | {{meeting,lunch},{training,presentation}}
 Bill  |  10000 | {{meeting,lunch},{training,presentation}}
 Carol |  20000 | {{breakfast,consulting},{meeting,lunch}}
 Carol |  25000 | {{breakfast,consulting},{meeting,lunch}}
 Carol |  25000 | {{breakfast,consulting},{meeting,lunch}}
 Carol |  25000 | {{breakfast,consulting},{meeting,lunch}}
(8 rows)
```

https://www.postgresql.org/docs/current/arrays.html

Arrays and Unnest

- Allows taking an array attribute and separate out its elements

```
university=# select * from sal_emp;
 name  |      pay_by_quarter       |                  schedule
-------+---------------------------+-------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

```
university=# select name, unnest(pay_by_quarter), unnest(schedule) from sal_emp;
 name  | unnest |    unnest
-------+--------+--------------
 Bill  |  10000 | meeting
 Bill  |  10000 | lunch
 Bill  |  10000 | training
 Bill  |  10000 | presentation
 Carol |  20000 | breakfast
 Carol |  25000 | consulting
 Carol |  25000 | meeting
 Carol |  25000 | lunch
(8 rows)
```

https://www.postgresql.org/docs/current/arrays.html

SQL Functions

- Function to count number of instructors in a department

```
create function dept_count (dept_name varchar(20))
    returns integer
        begin
        declare d_count  integer;
            select count (* ) into d_count
            from instructor
            where instructor.dept_name = dept_name
        return d_count;
    end
```

- Can use in queries

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

SQL Procedures

- Same function as a procedure

```
create procedure dept_count_proc (in dept_name varchar(20),
                                  out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name
end
```

- But use differently:

```
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```

- HOWEVER: Syntax can be wildly different across different systems
  - Was put in place by DBMS systems before standardization
  - Hard to change once customers are already using it

SQL Functions/Procedures

- Stored procedures widely used in practice
  - Many benefits including reusability, better performance (reduce back and forth to the DB)
- Most database systems support multiple langauges
  - Purely SQL → Fully procedural (e.g., C, etc)
- PostgreSQL supports SQL, C, PL/pgSQL
  - Note PostgreSQL 10 (that we use) does not support PROCEDURE, only FUNCTION

```
CREATE FUNCTION c_overpaid (EMP, INTEGER) RETURNS BOOLEAN AS '
    DECLARE
        emprec ALIAS FOR $1;
        sallim ALIAS FOR $2;
    BEGIN
        IF emprec.salary ISNULL THEN
            RETURN ''f'';
        END IF;
        RETURN emprec.salary > sallim;
    END;
' LANGUAGE 'plpgsql';
```

Pivots, Rollups, Cubes

item_name	*color*	*clothes_size*	*quantity*
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2

```
select *
from sales
pivot (
        sum(quantity)
        for color in ('dark', 'pastel', 'white')
)
```

item_name	*clothes_size*	*dark*	*pastel*	*white*
dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
pants	small	14	1	3
pants	medium	6	0	0
pants	large	0	1	2
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
skirt	small	2	11	2
skirt	medium	5	9	5
skirt	large	1	15	3

Not supported by PostgreSQL as is
Has something called “crosstab”

Pivots, Rollups, Cubes

item_name	*color*	*clothes_size*	*quantity*
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2

select *item_name*, *color*, **sum**(*quantity*)
from *sales*
group by rollup(*item_name*, *color*);

item_name	*color*	*quantity*
skirt	dark	8
skirt	pastel	35
skirt	white	10
dress	dark	20
dress	pastel	10
dress	white	5
shirt	dark	14
shirt	pastel	7
shirt	white	28
pants	dark	20
pants	pastel	2
pants	white	5
skirt	*null*	53
dress	*null*	35
shirt	*null*	49
pants	*null*	27
null	*null*	164

Not supported by PostgreSQL as is
Has something called “crosstab”

Triggers

- A ***trigger*** is a statement that is executed automatically by the system as a side effect of a modification to the database.

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  - 1. setting the account balance to zero
  - 2. creating a loan in the amount of the overdraft
  - 3. giving this loan a loan number identical to the account number of the overdrawn account

Trigger Example in SQL:1999

create trigger *overdraft-trigger* **after update on** *account*
referencing new row as *nrow*
for each row
when *nrow.balance* < 0
begin atomic
actions to be taken
end

Trigger Example in SQL:1999

```
create trigger overdraft-trigger after update on account
    referencing new row as nrow
    for each row
    when nrow.balance < 0
    begin atomic
            insert into borrower
                (select customer-name, account-number
                 from depositor
                 where nrow.account-number = depositor.account-number);
            insert into loan values
                (nrow.account-number, nrow.branch-name, nrow.balance);
            update account set balance = 0
             where account.account-number = nrow.account-number
    end
```

PostgreSQL Trigger Syntax

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

https://www.postgresql.org/docs/12/sql-createtrigger.html

NOTE: We use PostgreSQL 10, which does not support PROCEUDRE

Triggers…

- External World Actions
  - How does the DB *order* something if the inventory is low ?
- Syntax
  - Every system has its own syntax
- Careful with triggers
  - Cascading triggers, Infinite Sequences…
- More Info/Examples:
  - http://www.adp-gmbh.ch/ora/sql/create_trigger.html
  - Google: “create trigger” oracle download-uk

Authorization/Security

- GRANT and REVOKE keywords
  - **grant select on** *instructor* **to** U_1, U_2, U_3
  - **revoke select on** *branch* **from** U_1, U_2, U_3
- Can provide select, insert, update, delete privileges
- Can provide this for tables, schemas, "functions/procedures", etc.
  - Some databases support doing this at the level of individual "tuples"
    - MS SQL Server: https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security?view=sql-server-ver15
    - PostgreSQL: https://www.postgresql.org/docs/10/ddl-rowsecurity.html
- Can also create "Roles" and do security at the level of roles

CMSC424: Database Design

Aside

Anatomy of a Web Application

Instructor: Amol Deshpande
amol@umd.edu

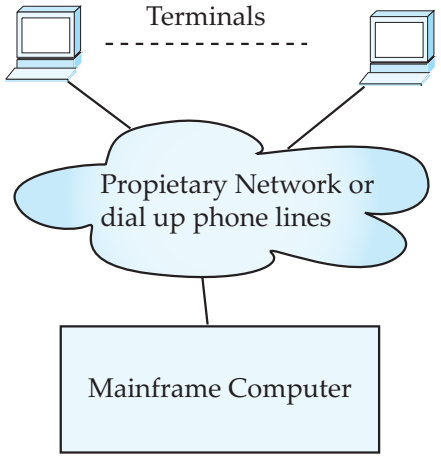
Anatomy of a Web Application

- Book Chapters (6th Edition)
  - Sections 9.1, 9.2, 9.3.5, 9.3.6, 9.4.3
  - Much not covered in depth in the book, but lot of good tutorials on the web

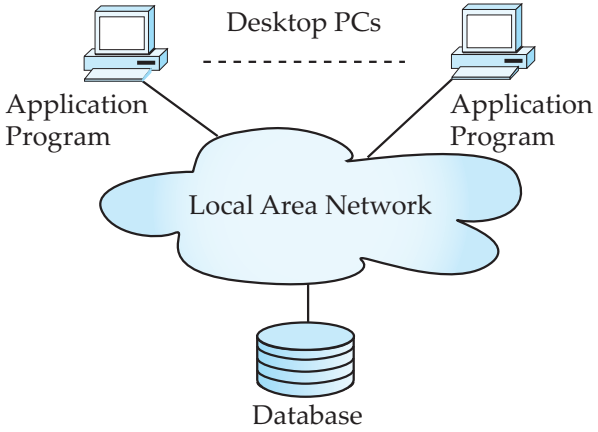
- Key Topics
  - How Web Applications Work
  - Some of the underlying technologies
  - REST

Application Architecture Evolution

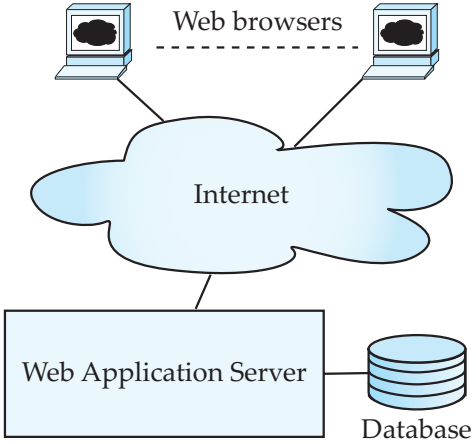
- Three distinct eras of application architecture
  - Mainframe (1960's and 70's)
  - Personal computer era (1980's)
  - Web era (mid 1990's onwards)
  - Web and Smartphone era (2010 onwards)



(a) Mainframe Era



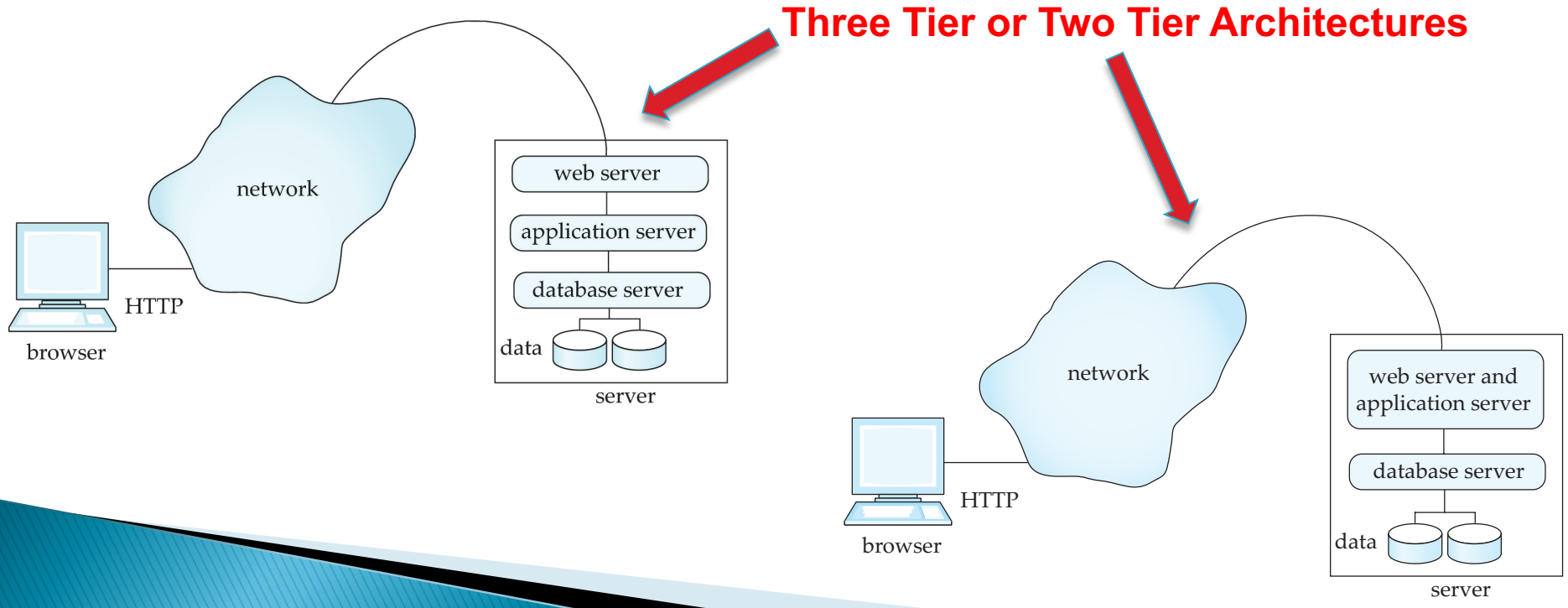
(b) Personal Computer Era



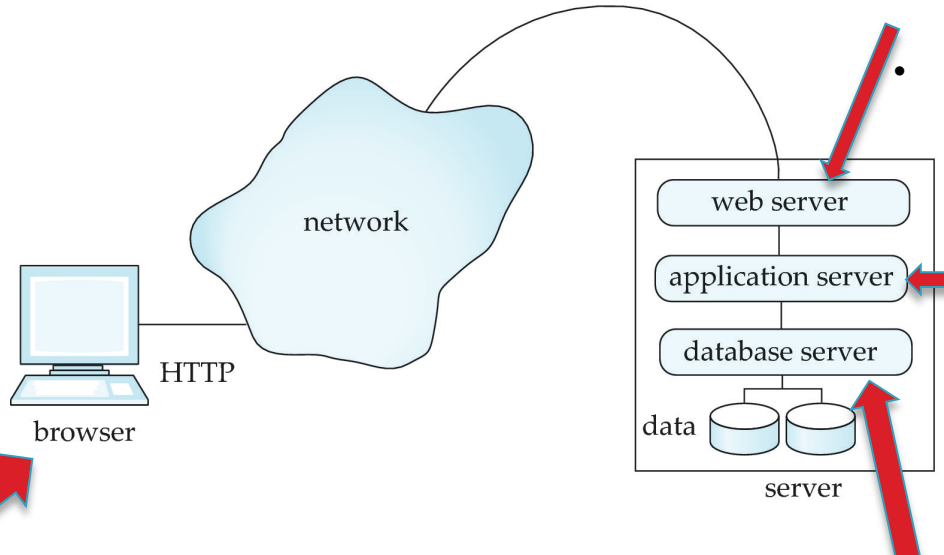
(c) Web era

Web or Mobile Applications

- Web browsers and mobile applications have become de facto standard user interface
  - Wide cross-platform accessibility
  - No need to download something



What runs where?



- Flask, Django, Tomcat, Node.js, and others
- Accept requests from the client and pass to the application server
- Pass application server response back to the client
- Support HTTP and HTTPS connections

- Encapsulates business logic
- Needs to support different user flows
- Needs to handle all of the rendering and visualization
- Ruby-on-rails, Django, Flask, Angular, React, PHP, and many others

1. Web Browser (Firefox, Chrome, Safari, Edge)
2. HTML to render webpages
3. Javascript for “client-side scripting” (running code in your browser without contacting the server)
4. Flash (not supported much – too much security risk)
5. Java “applets” – less common today

- PostgreSQL, Oracle, SQL Server, Amazon RDS (Relational Databases)
- MongoDB (Document/JSON databases)
- SQLite --- not typically for production environments
- Pretty much any database can be used…

Some Key Technologies

- HTML
  - Controls display of content on webpages
- HTTP/HTTPS, Sessions, Cookies
  - How “clients” connect to “servers”
- Server-side vs client-side scripting
  - Some processing happens on the server, but increasingly on the client (though Javascript)
- REST, SOAP, GraphQL
  - Protocols for “clients” to requests things from the “servers” (or for two web services to talk to each other)
- Web APIs (typically REST or GraphQL)
  - Some services available on the Web

REST

- **Representation State Transfer**: use standard HTTP requests to execute a request (against a web or application server) and return data
  - Technically REST is a software architectural style -- APIs that conform to it are called RESTful APIs

- How REST uses the five standard HTTP request types:
  - POST: Invoke the method that corresponds to the URL, typically with data that is sent with the request
  - GET: Retrieve the data (no data sent with the request)
  - PUT: Reverse of GET
  - PATCH: Update some data
  - DELETE: Delete the data

As someone on Stackoverflow put it: "**REST** is the way **HTTP** should be *used*."

- **Alternative: GraphQL** -- uses HTTP POST calls, where the body of the call tells the web server what needs to be done

REST – GET Calls

https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

REST Example: Twitter

API reference contents

GET /2/tweets (lookup by list of IDs)

GET /2/tweets/:id (lookup by single ID)

GET /2/tweets/:id (lookup by single ID)

Returns a variety of information about a single Tweet specified by the requested ID.

Run in Postman ›

Endpoint URL

`https://api.twitter.com/2/tweets/:id`

Authentication and rate limits

Authentication methods supported by this endpoint	OAuth 2.0 Bearer token OAuth 1.0a User context
Rate limit	300 requests per 15-minute window (app auth) 900 requests per 15-minute window (user auth)

CMSC424: Database Design

Module: Relational Model + SQL

SQL and Programming Languages

Instructor: Amol Deshpande
amol@umd.edu

SQL and Programming Languages

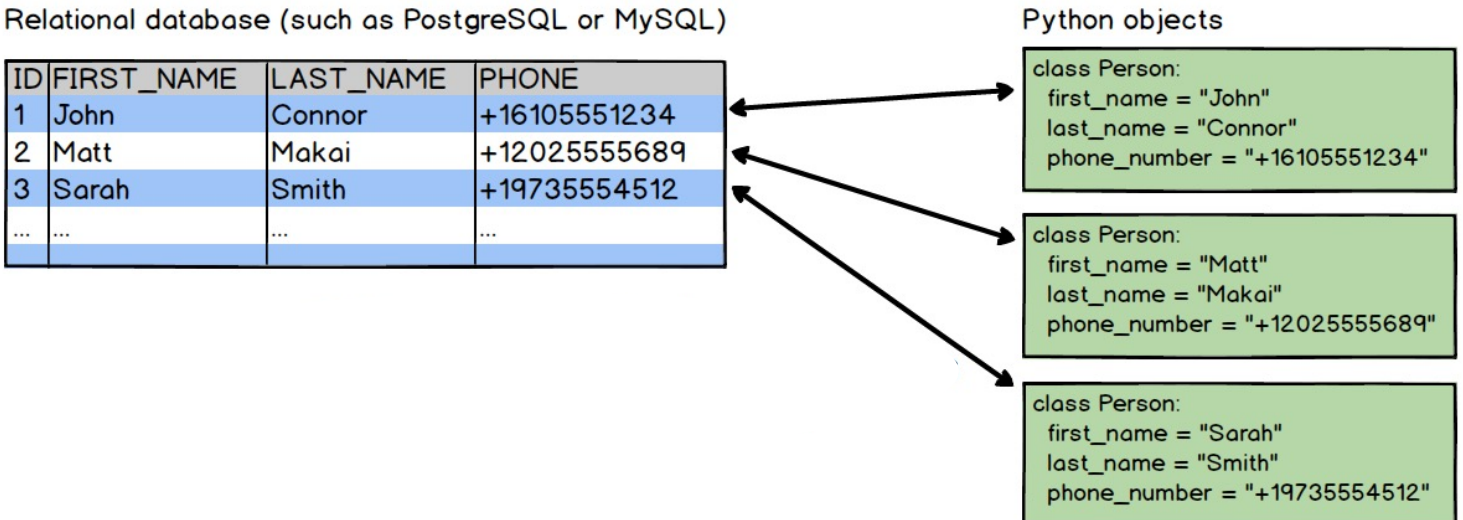
- Book Chapters (6th Edition)
  - Sections 5.1, 9.4.2

- Key Topics
  - Why use a programming language
  - Embedded SQL vs OBDC/JDBC
  - Object-relational impedance mismatch
  - Object-relational Mapping Frameworks

SQL and Programming Languages

- Programmers/developers more comfortable using a programming language like Java, Python, etc.
  - SQL not natural for many things
  - Performance issues in going back and forth to the database

- Need to deal with impedance mismatch between:
  - how data is represented in memory (typically as objects)
  - how it is stored (typically in a “normalized” relational schema)



Option 1: JDBC/ODBC

- Use a standard protocol like JDBC (Java Database Connectivity) to talk to the database from the programming language

```
>>> import jaydebeapi
>>> conn = jaydebeapi.connect("org.hsqldb.jdbcDriver",
...                           "jdbc:hsqldb:mem:.",
...                           ["SA", ""],
...                           "/path/to/hsqldb.jar",)
>>> curs = conn.cursor()
>>> curs.execute('create table CUSTOMER'
...              '("CUST_ID" INTEGER not null,'
...              ' "NAME" VARCHAR(50) not null,'
...              ' primary key ("CUST_ID"))'
...             )
>>> curs.execute("insert into CUSTOMER values (1, 'John')")
>>> curs.execute("select * from CUSTOMER")
>>> curs.fetchall()
[(1, u'John')]
>>> curs.close()
>>> conn.close()
```

- Doesn't solve impedance mismatch problem
  - Have to convert from the "result tuples" into "objects" and vice versa (when updating)

```java
import java.sql.*;

public class JDBCExample
{

    public static void main(String[] argv) {
        System.out.println("-------- PostgreSQL " + "JDBC Connection Testing ------------");
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            System.out.println("Where is your PostgreSQL JDBC Driver? " + "Include in your library path!");
            e.printStackTrace();
            return;
        }

        System.out.println("PostgreSQL JDBC Driver Registered!");
        Connection connection = null;
        try {
            connection = DriverManager.getConnection("jdbc:postgresql://localhost:5432/olympics","vagrant", "vagrant");
        } catch (SQLException e) {
            System.out.println("Connection Failed! Check output console");
            e.printStackTrace();
            return;
        }

        if (connection != null) {
            System.out.println("You made it, take control your database now!");
        } else {
            System.out.println("Failed to make connection!");
            return;
        }

        Statement stmt = null;
        String query = "select * from players;";
        try {
            stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String name = rs.getString("name");
                System.out.println(name + "\t");
            }
            stmt.close();
        } catch (SQLException e ) {
            System.out.println(e);
        }
    }
}
```

Option 1: JDBC/ODBC

- WARNING: always use prepared statements when taking an input from the user and adding it to a query (Related to the issue of SQL Injection attacks)
  - NEVER create a query by concatenating strings
  - "insert into instructor values(' " + ID + " ', ' " + name + " ', " + " ' + dept name + " ', " ' balance + ')“
  - What if name is “D'Souza”?

```
PreparedStatement pStmt = conn.prepareStatement("insert into instructor
values(?,?,?,?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();
```

- Python psycopg2 also has its own way of doing prepared statements

```
cur = conn.cursor()
for i, j in parameters:
        cur.execute( "select * from tables where i = %s and j = %s", (i, j))
         for record in cur: do_something_with(record)
```

Option 1: JDBC/ODBC

- JDBC Features
  - Getting schemas, columns, primary keys
    - DatabaseMetaData dbmd = conn.getMetaData()
  - Transaction control
    - conn.commit(), conn.rollback()
  - Calling functions and procedures

- ODBC: Open Database Connectivity Standard
  - Similar in many ways
  - Older – designed by Microsoft and typically used in C, C++, like languages
    - Java supports as well but slower

Option 2: Embedded SQL

- SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1
  - The language in which embedded is call “host” language

```
C++

int main() {
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL BEGIN DECLARE SECTION;
      int OrderID;          /* Employee ID (from user)         */
      int CustID;             /* Retrieved customer ID         */
      char SalesPerson[10]    /* Retrieved salesperson name      */
      char Status[6]          /* Retrieved order status        */
   EXEC SQL END DECLARE SECTION;

   /* Set up error processing */
   EXEC SQL WHENEVER SQLERROR GOTO query_error;
   EXEC SQL WHENEVER NOT FOUND GOTO bad_number;

   /* Prompt the user for order number */
   printf ("Enter order number: ");
   scanf_s("%d", &OrderID);

   /* Execute the SQL query */
   EXEC SQL SELECT CustID, SalesPerson, Status
      FROM Orders
      WHERE OrderID = :OrderID
      INTO :CustID, :SalesPerson, :Status;

   /* Display the results */
   printf ("Customer number:  %d\n", CustID);
   printf ("Salesperson: %s\n", SalesPerson);
   printf ("Status: %s\n", Status);
   exit();
```

Option 2: Embedded SQL

- SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1
  - The language in which embedded is call “host” language
- Needs compiler support for the host language
  - The compiler needs to know what to do with the EXEC SQL commands
  - Hard to port
- Doesn’t solve impedance mismatch problem
  - Have to convert from the “result tuples” into “objects” and vice versa (when updating)
- Not a preferred approach today

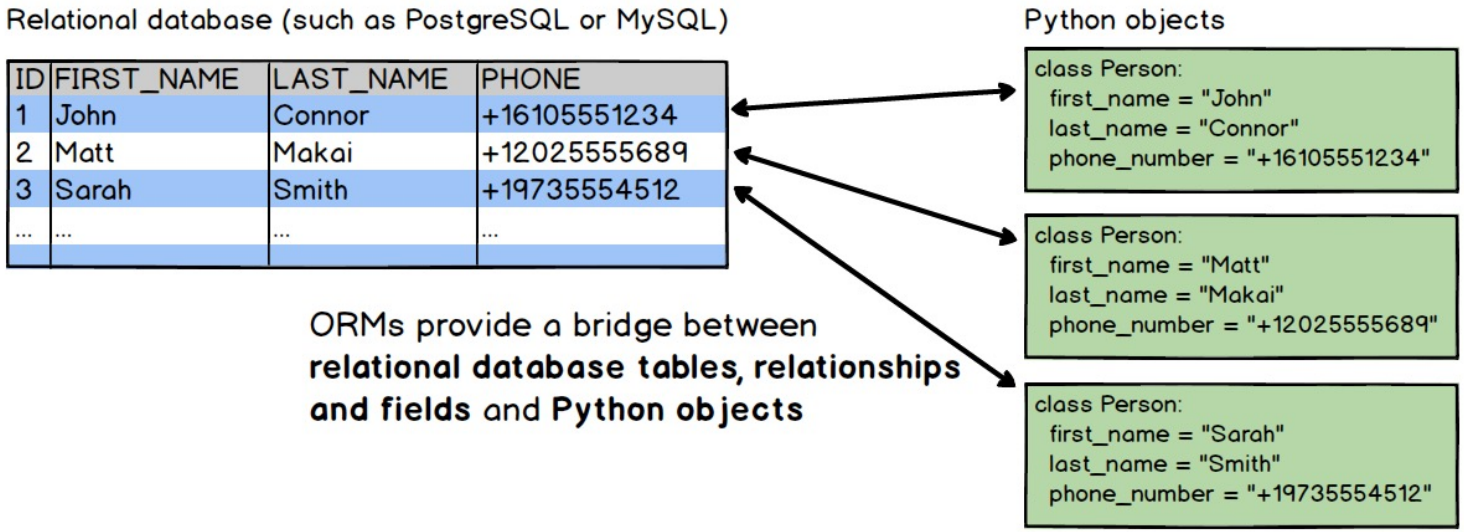
Option 3: Custom Libraries

- Often there are vendor-specific libraries that sometimes use internal protocols (and not JDBC/ODBC)
- e.g., python psycopg2 for PostgreSQL – although similar to JDBC calls, it uses the same proprietary protocol that 'psql' uses

```
conn = psycopg2.connect("dbname=olympics user=vagrant")
cur = conn.cursor()

totalscore = 0
for i in range(0, 14):
    # If a query is specified by -q option, only do that one
    if args.query is None or args.query == i:
        try:
            if interactive:
                os.system('clear')
            print("========== Executing Query {}".format(i))
            print(queries[i])
            cur.execute(queries[i])

            if i not in [5, 6, 8, 9]:
                ans = cur.fetchall()

                print("--------- Your Query Answer ---------")
                for t in ans:
                    print(t)
                print("")
```

Option 4: Object-relational Mappers

- Aimed at solving the impedance mismatch
  - Primarily for Web Application Development
- The ORM takes care of the mapping between objects and the database
  - Although largely designed around RDBMS, some ORMs support other databases as well
- The programmer works with objects, and never directly sees the SQL
  - Has pros (easier to use) and cons (performance and correctness issues)
- ORMs typically work with “Entities/Objects” and “Relationships”
  - Aligns well with the ER model that we will discuss next
  - We will cover Django constructs in more detail



Option 5: Other Mappers

- Many other “wrappers” on top of relational databases that offer different functionalities
  - In some cases, operations written in a higher-level language mapped to SQL
    - like what we saw for ORMs
    - Microsoft LINQ is also similar
    - Allows intermixing of code mapped to SQL and other code
  - In some cases, used to provide alternate data models to users
    - e.g., a thin layer that provides a graph data model, but stores data in a relational database
    - Most “RDF” databases built on top of SQL databases

- In today’s big data ecosystem, we see many many permutations how different tools (including databases) are combined together

CMSC424: Database Design

Module: Relation Model + SQL

SQL: Integrity Constraints

Instructor: Amol Deshpande
amol@umd.edu

SQL Integrity Constraints

- Book Chapters (6th Edition)
  - 4.4

- Key Topics
  - Why Constraints
  - Different Types of Integrity Constraints
  - Referential Integrity
  - How to specify in SQL

IC's

- Goal: Avoid Semantic Inconsistencies in the Data
- An IC is a predicate on the database
- Must always be true (checked whenever DB gets updated)

- There are the following 4 types of IC's:
  - Key constraints (1 table)

    e.g., *2 accts can't share the same acct_no*
  - Attribute constraints (1 table)

    e.g., *accts must have nonnegative balance*
  - Referential Integrity constraints (2 tables)

    E.g. *bnames* associated w/ *loans* must be names of real branches
  - Global Constraints (*n* tables)

    E.g., all *loans* must be carried by at least 1 *customer* with a savings acct

Key Constraints

Idea: specifies that a relation is a set, not a bag

SQL examples:

1. Primary Key:

```
CREATE TABLE branch(
        bname  CHAR(15)  PRIMARY KEY,
        bcity      CHAR(20),
        assets     INT);
```

or

```
CREATE TABLE depositor(
        cname   CHAR(15),
        acct_no  CHAR(5),
         PRIMARY KEY(cname, acct_no));
```

2. Candidate Keys:

```
CREATE TABLE customer (
        ssn     CHAR(9)    PRIMARY KEY,
        cname  CHAR(15),
        address CHAR(30),
        city          CHAR(10),
         UNIQUE (cname, address, city));
```

Key Constraints

Effect of SQL Key declarations

PRIMARY (A1, A2, .., An) or
UNIQUE (A1, A2, ..., An)

Insertions: check if any tuple has same values for A1, A2, .., An as any inserted tuple. If found, **reject insertion**

Updates to any of A1, A2, ..., An: treat as insertion of entire tuple

Primary vs Unique (candidate)

1. 1 primary key per table, several unique keys allowed.
2. Only primary key can be referenced by “foreign key” (ref integrity)
3. DBMS may treat primary key differently
   (e.g.: create an index on PK)

Attribute Constraints

- Idea:
  - Attach constraints to values of attributes
  - Enhances types system (e.g.: >= 0 rather than integer)
- In SQL:

1. NOT NULL

```
e.g.:   CREATE TABLE branch(
                bname   CHAR(15)  NOT NULL,
                ....
                )
```

Note: declaring bname as primary key also prevents null values

2. CHECK

```
e.g.:   CREATE TABLE depositor(
                ....
                balance int NOT NULL,
                CHECK(  balance >= 0),
                ....
                )
```

affect insertions, update in affected columns

Attribute Constraints

Domains: can associate constraints with DOMAINS rather than attributes

e.g: instead of:

```
CREATE TABLE depositor(
    ....
    balance INT NOT NULL,
    CHECK  (balance >= 0)
     )
```

One can write:

```
CREATE DOMAIN  bank-balance INT (
    CONSTRAINT not-overdrawn CHECK (value >= 0),
    CONSTRAINT not-null-value CHECK( value NOT NULL));
```

```
CREATE TABLE depositor (
    .....
    balance    bank-balance,
    )
```

Advantages?

Attribute Constraints

Advantage of associating constraints with domains:

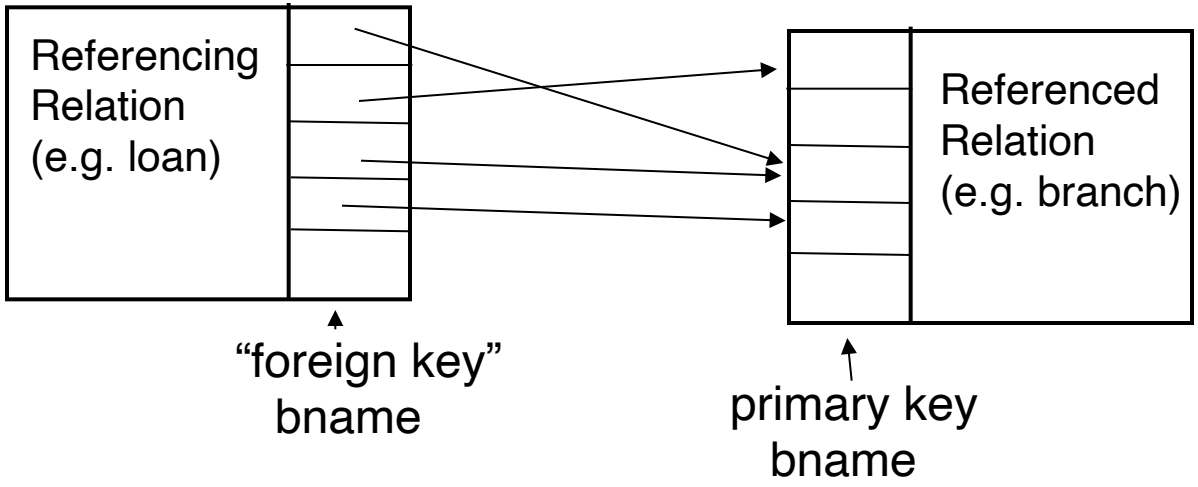
1. can avoid repeating specification of same constraint
   for multiple columns

2. can name constraints
   e.g.: CREATE DOMAIN bank-balance INT (
   CONSTRAINT not-overdrawn
   CHECK (value >= 0),
   CONSTRAINT not-null-value
   CHECK(value NOT NULL));

   allows one to:
   1. add or remove:
      ALTER DOMAIN bank-balance
      ADD CONSTRAINT capped
      CHECK(value <= 10000)
   2. report better errors (know which constraint violated)

Referential Integrity Constraints

Idea: prevent "dangling tuples" (e.g.: a loan with a bname, *Kenmore*, when no *Kenmore* tuple in branch)



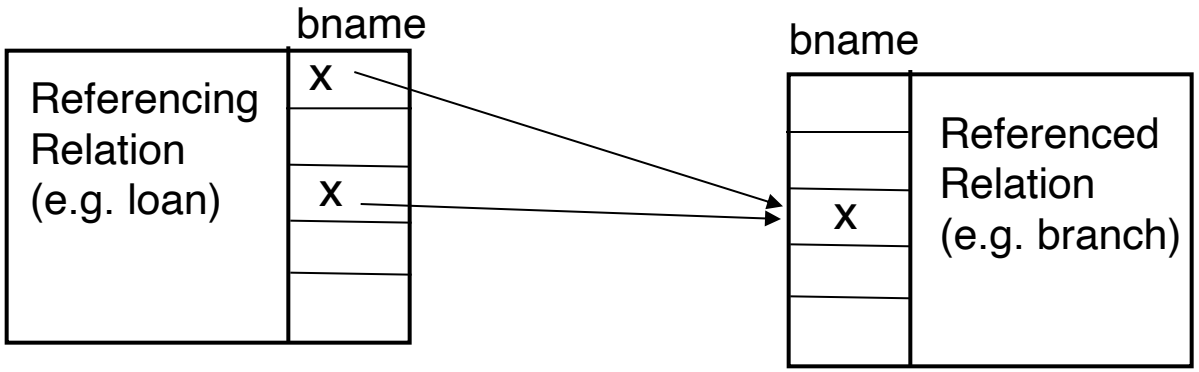
Ref Integrity:

ensure that:

foreign key value → primary key value

(note: don't need to ensure ←, i.e., not all branches have to have loans)

Referential Integrity Constraints



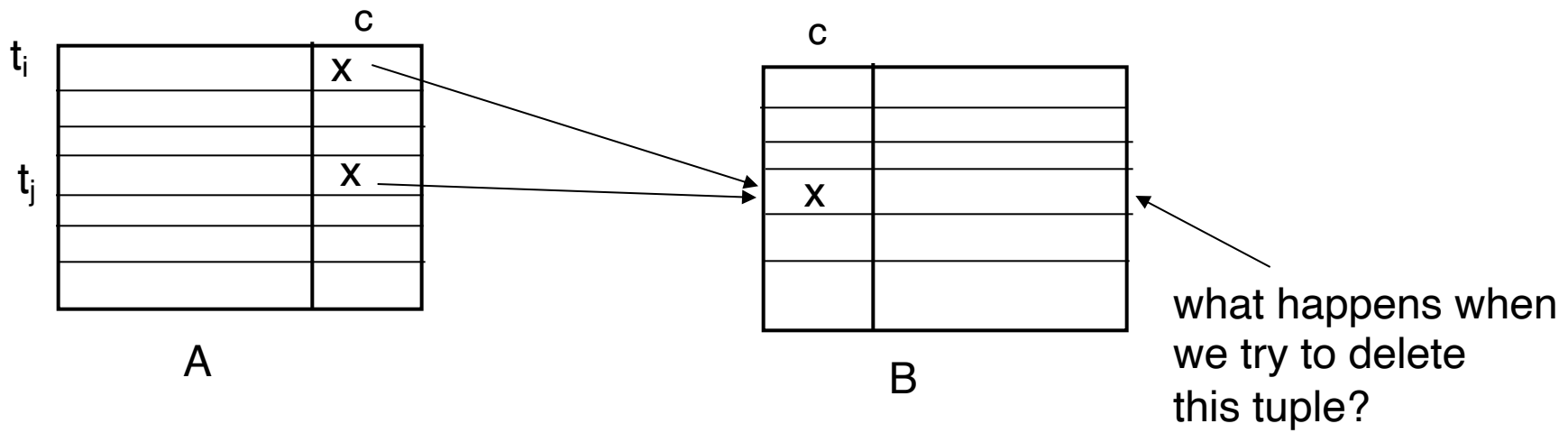
In SQL:

```
CREATE TABLE  branch(
      bname   CHAR(15)   PRIMARY KEY
      ....)

CREATE TABLE loan (
        .........
        FOREIGN KEY bname REFERENCES branch);
```

Affects:

1) Insertions, updates of referencing relation
2) Deletions, updates of referenced relation

Referential Integrity Constraints



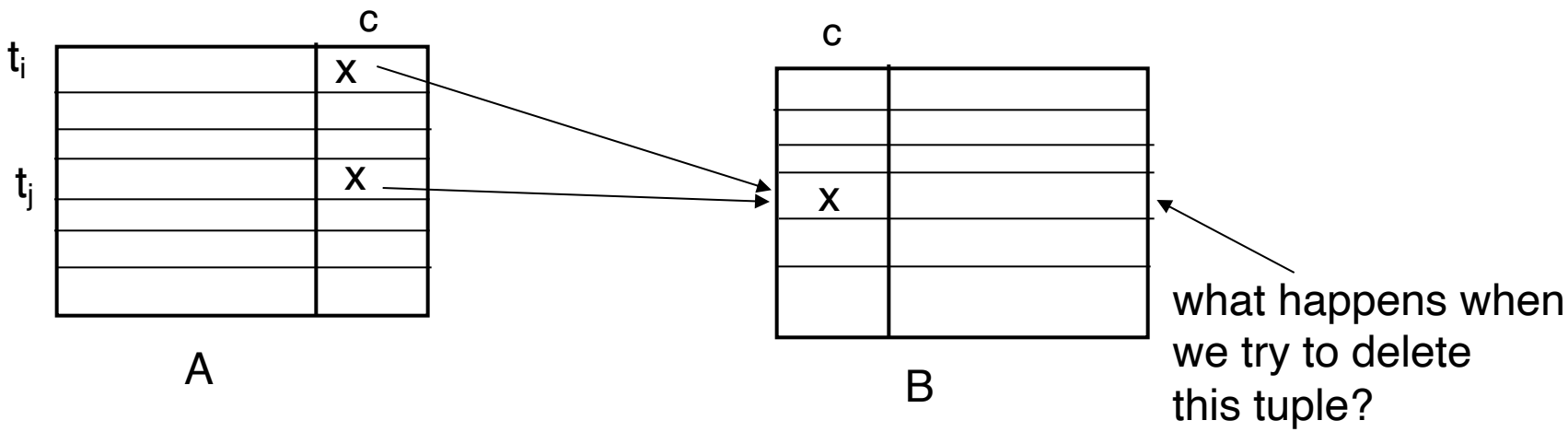
Ans: 3 possibilities

1) reject deletion/ update

2) set $t_i[c], t_j[c]$ = NULL

3) propagate deletion/update
   DELETE: delete ti, tj
   UPDATE: set ti[c], tj[c] to updated values

Referential Integrity Constraints



CREATE TABLE A (.....
FOREIGN KEY c REFERENCES B action
.....)

Action: 1) left blank (deletion/update rejected)

2) ON DELETE SET NULL/ ON UPDATE SET NULL
sets ti[c] = NULL, tj[c] = NULL

3) ON DELETE CASCADE
deletes ti, tj
ON UPDATE CASCADE
sets ti[c], tj[c] to new key values

Global Constraints

Idea: two kinds

1) single relation (constraints spans multiple columns)
   - E.g.: CHECK (total = svngs + check) declared in the CREATE TABLE
2) multiple relations: CREATE ASSERTION

SQL examples:

1) single relation: All Bkln branches must have assets > 5M

```
CREATE TABLE branch (
        .....
        bcity  CHAR(15),
        assets INT,
        CHECK (NOT(bcity = 'Bkln') OR assets > 5M))
```

Affects:

insertions into branch
updates of bcity or assets in branch

Global Constraints

SQL example:

2) Multiple relations: every loan has a borrower with a savings account

```
CHECK (NOT EXISTS (
                SELECT   *
                FROM loan AS L
                WHERE  NOT EXISTS(
                        SELECT   *
                        FROM borrower B, depositor D, account A
                        WHERE B.cname = D.cname  AND
                                D.acct_no = A.acct_no  AND
                                L.lno  = B.lno)))
```

Problem: Where to put this constraint? At depositor? Loan? ....

Ans: None of the above:

```
        CREATE ASSERTION loan-constraint
                CHECK(  ..... )
```

Checked with EVERY DB update!
very expensive.....

Summary: Integrity Constraints

Constraint Type	Where declared	Affects...	Expense
Key Constraints	CREATE TABLE (PRIMARY KEY, UNIQUE)	Insertions, Updates	Moderate
Attribute Constraints	CREATE TABLE CREATE DOMAIN (Not NULL, CHECK)	Insertions, Updates	Cheap
Referential Integrity	Table Tag (FOREIGN KEY .... REFERENCES ....)	1.Insertions into referencing rel'n 2. Updates of referencing rel'n of relevant attrs 3. Deletions from referenced rel'n 4. Update of referenced rel'n	1,2: like key constraints. Another reason to index/sort on the primary keys 3,4: depends on a. update/delete policy chosen b. existence of indexes on foreign key
Global Constraints	Table Tag (CHECK) or outside table (CREATE ASSERTION)	1. For single rel'n constraint, with insertion, deletion of relevant attrs 2. For assesrtions w/ every db modification	1. cheap 2. very expensive

CMSC424: Database Design

Module: Relational Model + SQL

SQL: Views

Instructor: Amol Deshpande
amol@umd.edu

SQL Views

- Book Chapters (6th Edition)
  - 3.8

- Key Topics
  - Defining Views and Use Cases
  - Difference between a view and a table
  - Updating a view

Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:

create view *v* **as** <query expression>

where:

<query expression> is any legal expression

The view name is represented by *v*

- Can be used in any place a normal table can be used
- For users, there is no distinction in terms of using it

Example Queries

- A view consisting of courses and sections for Physics in Fall 2009

```
create view physics_fall_2009 as
    select course.course_id, sec_id, building, room_number
    from course, section
    where course.course_id = section.course_id
                and course.dept_name = 'Physics'
                and section.semester = 'Fall'
                and section.year = '2009';
```

Find all physics fall 2009 courses in a building.

```
select course_id
from physics_fall_2009
where building= 'Watson';
```

Views

- Is it different from DBMS's side ?
  - Yes; a view may or may not be *materialized*
  - Pros/Cons ?

- Updates into views have to be treated differently
  - In most cases, disallowed.

Views vs Tables

Creating	Create view V as (select * from A, B where …)	Create table T as (select * from A, B where …)
Can be used	In any select query. Only some update queries.	It's a new table. You can do what you want.
Maintained as	1. Evaluate the query and store it on disk as if a table. 2. Don't store. Substitute in queries when referenced.	It's a new table. Stored on disk.
What if a tuple inserted in A ?	1. If stored on disk, the stored table is automatically updated to be accurate. 2. If we are just substituting, there is no need to do anything.	T is a separate table; there is no reason why DBMS should keep it updated. If you want that, you must define a trigger.

Views vs Tables

- Views strictly supercede “create a table and define a trigger to keep it updated”

- Two main reasons for using them:
  - Security/authorization
    - Can provide a user with ”read” access to only the view
  - Ease of writing queries
    - E.g. *PresidentStateReturns* , or a view listing who won which state

- Perhaps the only reason to create a table is to force the DBMS to choose the option of “materializing”
  - That has efficiency advantages in some cases
  - Especially if the underlying tables don’t change

Update of a View

- Create a view of all instructors while hiding the salary

  create view *faculty* **as**
  select *ID*, *name*, *dept_name*
  from *instructor*;

- Add a new tuple to the view

  insert into *faculty* **values** ('30765', 'Green', 'Music');

- Options:
  - Reject because we don't "salary" information, or
  - Insert into "instructors": ('30765', 'Green', 'Music', NULL);

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.
- Many SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

CMSC424: Database Design

Module: Relational Model + SQL

Transactions: Overview

Instructor: Amol Deshpande
amol@umd.edu

Transactions: Overview

- Book Chapters (7th Edition)
  - 4.3, 17

- Key topics:
  - Transactions and ACID Properties
  - Different states a transaction goes through
  - Transactions in PostgreSQL

Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
  - Transactions are started implicitly and terminated by one of
    - commit work: makes all updates of the transaction permanent in the database
    - rollback work: undoes all updates performed by the transaction.
- Motivating example
  - Transfer of money from one account to another involves two steps:
    - deduct from one account and credit to another
  - If one steps succeeds and the other fails, database is in an inconsistent state
  - Therefore, either both steps should succeed or neither should
- If any step of a transaction fails, all work done by the transaction can be undone by rollback work.
- Rollback of incomplete transactions is done automatically, in case of system failures

Transactions (Cont.)

- In most database systems, each SQL statement that executes successfully is automatically committed.
  - Each transaction would then consist of only a single statement
  - Automatic commit can usually be turned off, allowing multi-statement transactions, but how to do so depends on the database system
  - Another option in SQL:1999: enclose statements within

```
begin atomic
  …
end
```

Examples of Transactions

insert into students values (…)

update instructor
set salary = salary * 1.03

enrolled = select count(*)
from takes
where (course_info) = (CMSC 424, 201, Fall 2022)
if enrolled < capacity for the room:
insert new student into takes for that course

(Modify prerequisites)
delete (CMSC422, CMSC351)
insert (CMSC422, CMSC320)

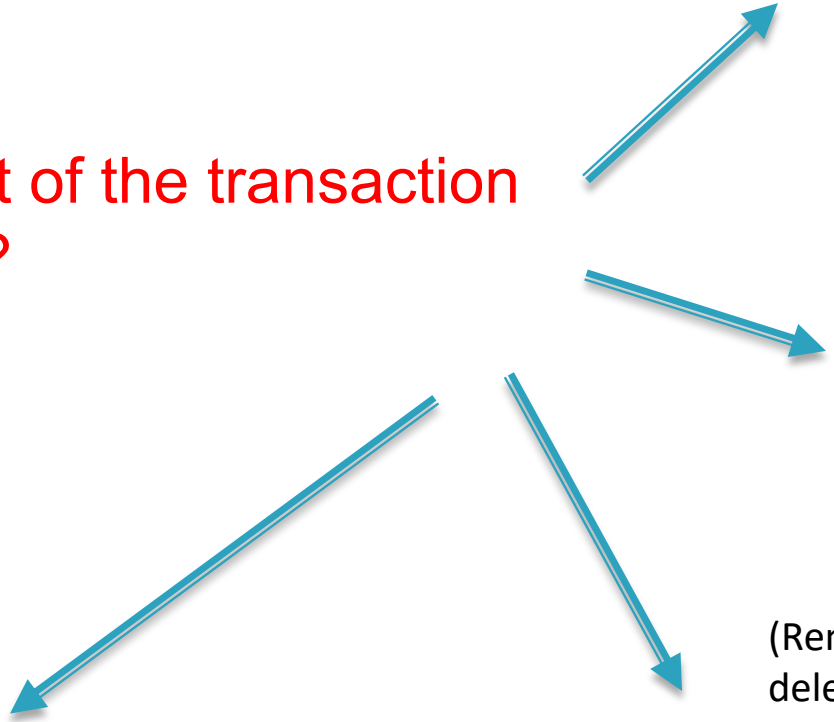
(Add a new section for a course for a given room and instructor)
if no section currently in that room:
insert a tuple into “sections” with that room
insert a tuple into “teaches”

(Remove a section)
delete from takes for that section
delete from teaches for that section
delete tuple from section

(Switch the advisor for a student)
delete old tuple with that s_id
add new tuple with that s_id and new advisor

Examples of Transactions

What if only part of the transaction was completed?



update instructor
set salary = salary * 1.03

(Modify prerequisites)
delete (CMSC422, CMSC351)
insert (CMSC422, CMSC320)

(Remove a section)
delete from takes for that section
delete from teaches for that section
delete tuple from section

(Switch the advisor for a student)
delete old tuple with that s_id
add new tuple with that s_id and new advisor

Examples of Transactions

```
enrolled = select count(*)
           from takes
           where (course_info) = (CMSC 424, 201, Fall 2022)
if enrolled < capacity for the room:
    insert student A into takes for that course
```

```
enrolled = select count(*)
           from takes
           where (course_info) = (CMSC 424, 201, Fall 2022
if enrolled < capacity for the room:
    insert student B into takes for that course
```

What if two different students tried to enroll at the same time?

Overview

- *Transaction*: A sequence of database actions enclosed within special tags
- Properties:
  - ***Atomicity***: Entire transaction or nothing
  - ***Consistency***: Transaction, executed completely, takes database from one consistent state to another
  - ***Isolation***: Concurrent transactions *appear* to run in isolation
  - ***Durability***: Effects of committed transactions are not lost

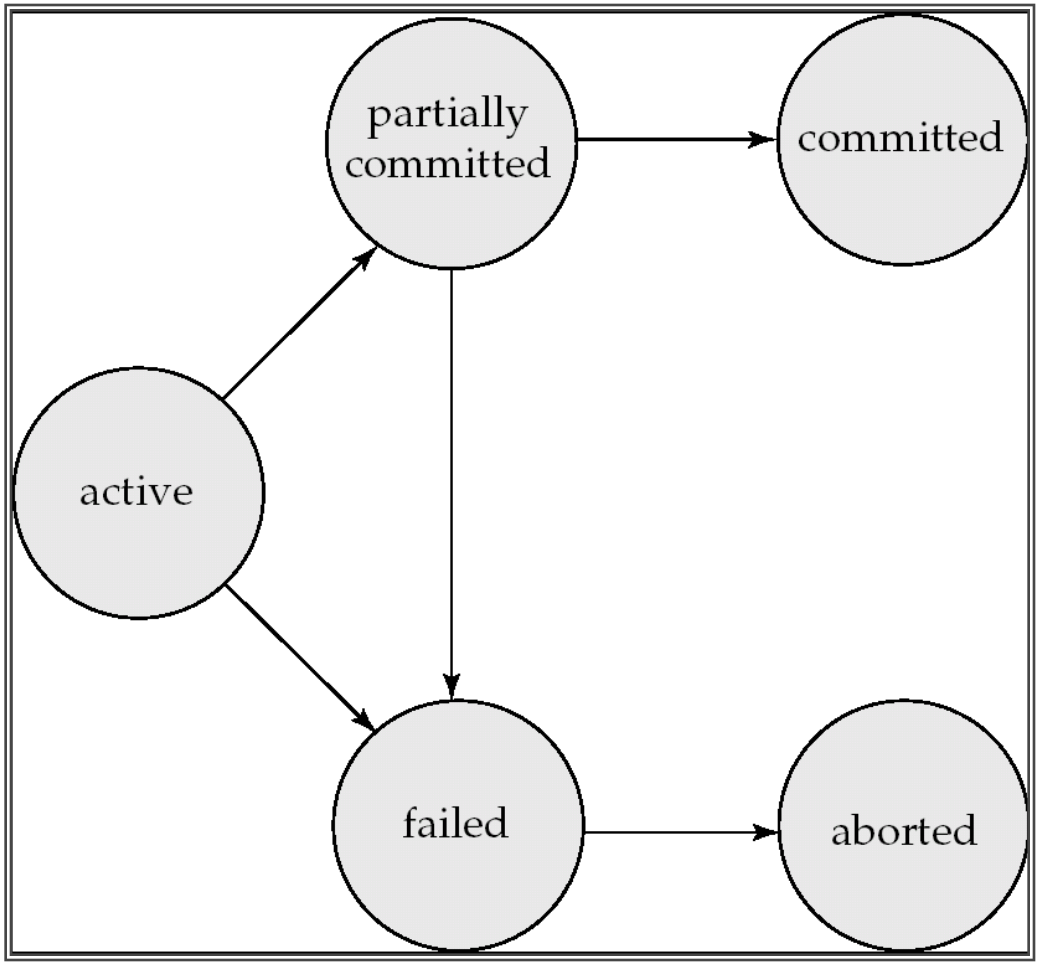
- Consistency: Transaction programmer needs to guarantee that
  - DBMS can do a few things, e.g., enforce constraints on the data

- Rest: DBMS guarantees

How does..

- .. this relate to *queries* that we discussed ?
  - Queries don't update data, so *durability* and *consistency* not relevant
  - Would want *concurrency*
    - Consider a query computing total balance at the end of the day
  - Would want *isolation*
    - What if somebody makes a *transfer* while we are computing the balance
    - Typically not guaranteed for such long-running queries
- TPC-C vs TPC-H

Transaction states



Initial State – stays in this during execution

Successful Completion

Any changes have been rolled back

Transactions in PostgreSQL

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- etc etc
COMMIT;
```

PostgreSQL actually treats every SQL statement as being executed within a transaction. If you do not issue a BEGIN command, then each individual statement has an implicit BEGIN and (if successful) COMMIT wrapped around it. A group of statements surrounded by BEGIN and COMMIT is sometimes called a transaction block.

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

https://www.postgresql.org/docs/current/tutorial-transactions.html

Summary

- Transactions is how we update data in databases
- ACID properties: foundations on which high-performance transaction processing systems are built
  - From the beginning, consistency has been a key requirement
  - Although “relaxed” consistency is acceptable in many cases (originally laid out in 1975)
- NoSQL systems originally eschewed ACID properties
  - MongoDB was famously bad at guaranteeing any of the properties
  - Lot of focus on what’s called “eventual consistency”
- Recognition today that strict ACID is more important than that
  - Hard to build any business logic if you have no idea if your transactions are consistent

CMSC424: Database Design

Module: Relational Model + SQL

Transactions: Isolation

Instructor: Amol Deshpande
amol@umd.edu

Isolation and Concurrency Levels

- Book Chapters
  - 14.5
- Key topics:
  - Why Concurrency
  - Notion of a ”Schedule”
  - Serializability
  - Isolation Levels

A Schedule

Transactions:

T1: transfers $50 from A to B

T2: transfers 10% of A to B

Database constraint: A + B is constant (*checking+saving accts)*

T1	T2
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

Each transaction obeys the constraint.

This schedule does too.

Schedules

- A *schedule* is simply a (possibly interleaved) execution sequence of transaction instructions
- *Serial Schedule:* A schedule in which transaction appear one after the other
  - ie., No interleaving
- Serial schedules satisfy isolation and consistency
  - Since each transaction by itself does not introduce inconsistency

Example Schedule

- Another “serial” schedule:

T1	T2
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	

Effect:	Before	After
A	100	40
B	50	110

Consistent ?

Constraint is satisfied.

Since each Xion is consistent, any serial schedule must be consistent

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B + tmp
	write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Consistent.
So this schedule is okay too.

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Further, the effect same as the serial schedule 1.

Called *serializable*

Example Schedules (Cont.)

A "bad" schedule

T1	T2
read(A)	
A = A -50	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
write(A)	
read(B)	
B=B+50	
write(B)	
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	50
B	50	60

Not consistent

Desired Property 1: Serializability

- A schedule is called *serializable* if its final effect is the same as that of a *serial schedule*
- Serializability → schedule is fine and doesn't cause inconsistencies
  - Since serial schedules are fine
- Non-serializable schedules unlikely to result in consistent databases
- Not possible to look at all *n!* serial schedules to check if the effect is the same
  - Instead we ensure serializability by allowing or not allowing certain schedules

Desired Property 1: Serializability

- However, serializability is considered too expensive to enforce
- SQL Standard allows “relaxed” consistency levels
- Defined in terms of three desirable properties (or in terms of “anomalies” to avoid)
  - No phantom phenomenon
  - Repeatable read
  - No dirty reads

Desired Property 2: No Phantom Phenomenon

- Schema: *accounts(acct_no, balance, zipcode, …)*
- **Transaction 1:** Find the number of accounts in *zipcode = 20742,* and divide $1,000,000 between them
- **Transaction 2:** Insert *<acctX, …, 20742, …>*

1. Transaction 1 counts the number of accounts with zipcode = 20742 (say 100)

2. Transaction 1 computes:
   share = 1000000/count
   = 10,000

3. Transaction 2 inserts a new account with the same zipcode

4. Transaction 1 goes through and adds "10,000" to every account with zipcode = 20742

Including the new account
So the total amount was higher

Desired Property 3: Repeatable Read

- Schema: *accounts(acct_no, balance, zipcode, …)*
- **Transaction 1:** Reads account #10549 twice
- **Transaction 2:** Updates account #10549

1. Transaction 1 reads account #10549 information to show user the balance

2. Transaction 2 updates account #10549

3. Transaction 1 reads/write account #10549 information based on user action

Transaction 1 reads a different value the second time (Note: Transaction 1 could "remember" it -- this is assuming that it doesn't

Desired Property 4: No Dirty Read

- Schema: *accounts(acct_no, balance, zipcode, …)*
- **Transaction 1:** Reads info for #10549
- **Transaction 2:** Updates several accounts over a period of time

1. Transaction 2 starts
2. Transaction 2 updates #10549

3. Transaction 1 reads info for #10549

4. Transaction 2 updates #10342
5. Transaction 2 finishes

Transaction 1 read "uncommitted" ("dirty") data
What if Transaction 2 never finishes, and is rolled back?

Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
- Unfortunately slightly different interpretations of the different properties

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

- In PostgreSQL, can set the level per transaction

```
SET TRANSACTION transaction_mode [, ...]
```

https://www.postgresql.org/docs/current/transaction-iso.html

Weak Levels of Consistency in SQL

- In many database systems, read committed is default
  - has to be explicitly changed to serializable when required
    - **set isolation level serializable**
- MySQL InnoDB defaults to repeatable read
- Oracle has no repeatable read
  - But does support a “read_only” mode
  - Let’s the transaction see all data as of the time it started
  - Part of what’s called “snapshot isolation”
- IBM DB2
  - Repeatable read is actually “serializable”
  - Has a different mode called “read stability” -- equivalent to “repeatable read”

CMSC424: Database Design

NOT IN SYLLABUS

SQLMan: Wielding the Superpower of SQL

Instructor: Amol Deshpande
amol@umd.edu

Fun with SQL

- https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/
  - Long slide-deck linked off of this page
  - Complex SQL queries showing how to do things like: do Mandelbrot, solve subset sum problem etc.

- **The MADlib Analytics Library or MAD Skills, the SQL;** https://arxiv.org/abs/1208.4165

- https://www.red-gate.com/simple-talk/blogs/statistics-sql-simple-linear-regressions/

1. Everything is a Table

```sql
SELECT *
FROM (
  SELECT *
  FROM person
) t
```

```sql
SELECT *
FROM (
  VALUES(1),(2),(3)
) t(a)
```

Everything is a table. In PostgreSQL, even functions are tables:

```sql
SELECT *
FROM substring('abcde', 2, 3)
```

https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/

2. Recursion can be very powerful

```
1  WITH RECURSIVE t(v) AS (
2    SELECT 1      -- Seed Row
3    UNION ALL
4    SELECT v + 1 -- Recursion
5    FROM t
6  )
7  SELECT v
8  FROM t
9  LIMIT 5
```

Makes SQL
Turing-Complete

It yields

```
 v
---
 1
 2
 3
 4
 5
```

https://blog.jooq.org/2016/04/25/10-sql-tricks-that-you-didnt-think-were-possible/

3. Window Functions

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
depname   | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

https://www.postgresql.org/docs/9.3/tutorial-window.html

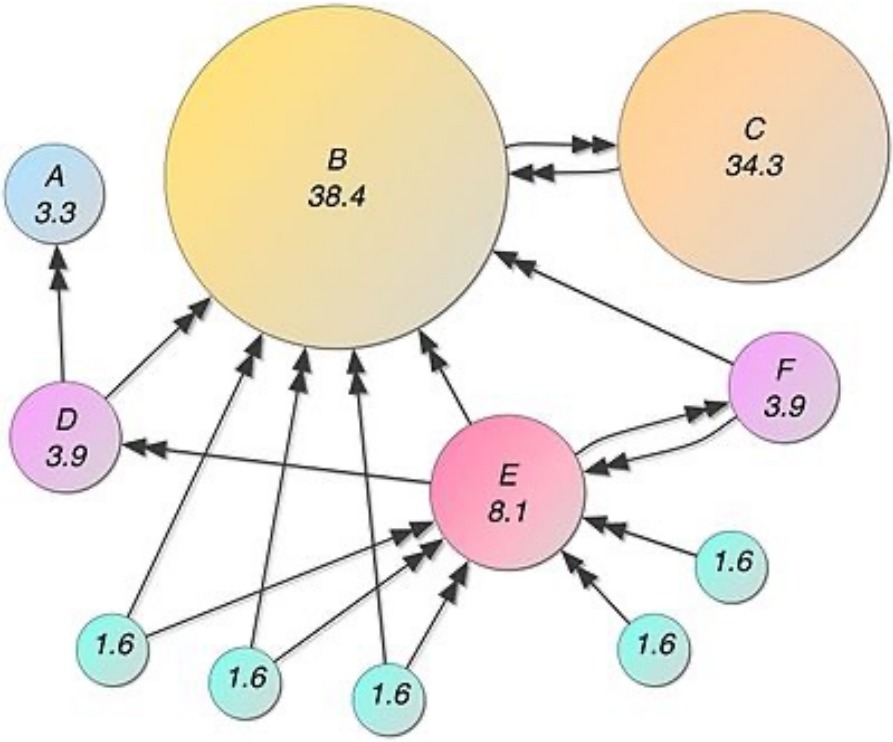
https://www.red-gate.com/simple-talk/blogs/statistics-sql-simple-linear-regressions/

4. Correlation Coefficient

```
SET ARITHABORT ON;

DECLARE @OurData TABLE
    (
    x NUMERIC(18,6) NOT NULL,
    y NUMERIC(18,6) NOT NULL
    );
  INSERT INTO @OurData
    (x, y)
  SELECT
   x,y
   FROM (VALUES
  (1,32),(1,23),(3,50),(11,37),(-2,39),(10,44),(27,32),(25,16),(20,23),
  (4,5),(30,41),(28,2),(31,52),(29,12),(50,40),(43,18),(10,65),(44,26),
  (35,15),(24,37),(52,66),(59,46),(64,95),(79,36),(24,66),(69,58),(88,56),
  (61,21),(100,60),(62,54),(10,14),(22,40),(52,97),(81,26),(37,58),(93,71)
  (64,82),(24,33),(112,49),(64,90),(53,90),(132,61),(104,35),(60,52),
  (29,50),(85,116),(95,104),(131,37),(139,38),(8,124)
  )f(x,y)
  SELECT
    ((Sy * Sxx) - (Sx * Sxy))
    / ((N * (Sxx)) - (Sx * Sx)) AS a,
    ((N * Sxy) - (Sx * Sy))
    / ((N * Sxx) - (Sx * Sx)) AS b,
    ((N * Sxy) - (Sx * Sy))
    / SQRT(
        (((N * Sxx) - (Sx * Sx))
         * ((N * Syy - (Sy * Sy))))) AS r
    FROM
      (
      SELECT SUM([@OurData].x) AS Sx, SUM([@OurData].y) AS Sy,
        SUM([@OurData].x * [@OurData].x) AS Sxx,
        SUM([@OurData].x * [@OurData].y) AS Sxy,
        SUM([@OurData].y * [@OurData].y) AS Syy,
        COUNT(*) AS N
        FROM @OurData
      ) sums;
```

5. Page Rank

- Recursive algorithm to assign weights to the nodes of a graph (Web Link Graph)
- Weight for a node depends on the weights of the nodes that point to it
- Typically done in iterations till "convergence"
- Not obvious that you can do it in SQL, but:
  - Each iteration is just a LEFT OUTERJOIN
  - Stopping condition is trickier
- Other ways to do it as well



https://devnambi.com/2013/pagerank.html

```
declare @DampingFactor decimal(3,2) = 0.85 --set the damping factor
        ,@MarginOfError decimal(10,5) = 0.001 --set the stable weight
        ,@TotalNodeCount int
        ,@IterationCount int = 1

-- we need to know the total number of nodes in the system
set @TotalNodeCount = (select count(*) from Nodes)

-- iterate!
WHILE EXISTS
(
        -- stop as soon as all nodes have converged
        SELECT *
        FROM dbo.Nodes
        WHERE HasConverged = 0
)
BEGIN

        UPDATE n SET
        NodeWeight = 1.0 - @DampingFactor + isnull(x.TransferWeight, 0.0)

        -- a node has converged when its existing weight is the same as the weight it would be given
        -- (plus or minus the stable weight margin of error)
        ,HasConverged = case when abs(n.NodeWeight - (1.0 - @DampingFactor + isnull(x.TransferWeight, 0.0))) < @MarginOfError then 1
else 0 end
        FROM Nodes n
        LEFT OUTER JOIN
        (
                -- Here's the weight calculation in place
                SELECT
                        e.TargetNodeId
                        ,TransferWeight = sum(n.NodeWeight / n.NodeCount) * @DampingFactor
                FROM Nodes n
                INNER JOIN Edges e
                  ON n.NodeId = e.SourceNodeId
                GROUP BY e.TargetNodeId
        ) as x
        ON x.TargetNodeId = n.NodeId

        -- for demonstration purposes, return the value of the nodes after each iteration
        SELECT
                @IterationCount as IterationCount
                ,*
        FROM Nodes

        set @IterationCount += 1
END
```