RAWDATA
Section 1

SQL part 2

Henrik Bulskov & Troels Andreasen

Intermediate SQL

- Join Expressions
- Views
- Integrity Constraints
- Indexing
- Authorization

Joined Relations

- **Join operations**
  - take two relations and return as a result another relation
  - basically a Cartesian product, however, only a subset where tuples from the two relations match (under some condition)

- A simple example of a **Join** with two relations in **from** is

```
select *
from course, prereq
where course.course_id = prereq.course_id
```

- **Join** operations can also be specified more explicitly
  - often as subquery expressions in the **from** clause

Join operations – Examples

- **Lets consider ways to combine info from the following two tables**

Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

Relation *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

  prereq information is missing for CS-315 and

  course information is missing for CS-437

Natural Join

- **select** *
  from *course* **natural join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101

- **Loss of information?**

course

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

prereq

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other.
- Uses *null* values.

- Variations
  - Left Outer Join
    - add tuples from the left argument relation
  - Right Outer Join
    - add tuples from the right argument relation
  - Full Outer Join
    - add tuples from both argument relations

Left Outer Join

- **select** *
  from *course* **natural left outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

course

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

prereq

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Right Outer Join

- **select** *
  from *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

course

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

prereq

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Full Outer Join

- **select** *
  from *course* **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

natural full outer join is not supported in MySQL (use UNION)

course

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

prereq

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Joined Relations

- Join operations
  - take two relations and return as a result another relation.
  - typically used as subquery expressions in the **from** clause
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

Joined Relations – Examples

- **select** *
  from *course* **inner join** *prereq* **on**
  course.course_id = prereq.course_id

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- What is the difference between the above, and a natural join of *course* and *prereq*?

Joined Relations – Examples

- **select** *
  from *course* **left outer join** *prereq* **on**
  course.course_id = prereq.course_id

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	*null*

Joined Relations – Examples

- **select** *
  from *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

- **select** *
  from *course* **right outer join** *prereq* **using** (*course_id*)

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

How to specify the latter with … join … on …?

4.1(a,b,d)

4.1 Write the following queries in SQL:

a. Display a list of all instructors, showing their ID, name, and the number of sections that they have taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outerjoin, and should not use scalar subqueries.

b. Write the same query as above, but using a scalar subquery, without outerjoin.

d. Display the list of all departments, with the total number of instructors in each department, without using scalar subqueries. Make sure to correctly handle departments with no instructors.

- d. Try to add a department to the database without instructors and check

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view**
  - provides a mechanism to hide certain data from the view of certain users.
  - is a "virtual relation" that is not part of the conceptual model

View Definition

- A view is defined using the **create view** statement which has the form

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

  where <query expression> is any legal SQL expression. The view name is represented by *v.*

- A view of instructors without their salary

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

- View definition
  - Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
  - a view definition causes the saving of an expression; the expression is substituted into queries using the view

Example Views

- A view of instructors without their salary

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

- Find all instructors in the Biology department

  select *name*
  from *faculty*
  where *dept_name* = 'Biology'

- Create a view of department salary totals

  create view *departments_total_salary*(*dept_name*, *total_salary*) **as**
  select *dept_name*, **sum**(*salary*)
  from *instructor*
  group by *dept_name*;

Views Defined Using Other Views

- **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’ ;

- **create view** *physics_fall_2009_watson* **as**
  select *course_id*, *room_number*
  from *physics_fall_2009*
  where *building*= ’ Watson’ ;

View Expansion

- Expand use of a view in a query

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

```
select course_id, room_number
from (select course.course_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' )
where building= 'Watson';
```

Update of a View

- A view of instructors

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

- A view of department salary totals

  create view *departments_total_salary*(*dept_name*, *total_salary*) **as**
  select *dept_name*, **sum**(*salary*)
  from *instructor*
  group by *dept_name*;

- Does update of these make sense?

Update of a View

- The faculty view

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

- Add a new tuple to *faculty* view

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

  This insertion can be represented by the insertion of the tuple

  (’30765’, ’Green’, ’Music’, null)

  into the *instructor* relation

Some Updates cannot be Translated Uniquely

- A view of department salary totals

  create view *departments_total_salary*(*dept_name*, *total_salary*) **as**
  select *dept_name*, **sum**(*salary*)
  from *instructor*
  group by *dept_name*;

- **insert into** *departments_total_salary* **values** ('Math', 75000);
  - which department rows to insert??

Some Updates cannot be Translated Uniquely

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - The query does not have a **group** by or **having** clause.

  - Any attribute not listed in the **select** clause can be set to null

Integrity Constraints

- Integrity constraints guard against accidental damage to the database and loss of data consistency.
  - A certain account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $10.00 an hour
  - A customer must have a (non-null) phone number

- DBMS' varies on the support for specification of constraints
  - MySQL is fairly limited,
  - However, by use of triggers and stored procedures quite many of the constraints we can think of can be implemented,

- Triggers and stored procedures will be covered later

Integrity Constraints

- **not null**
- **primary key**
- **Unique**
- **check** (P), where P is a predicate
- **Referencial integrity**

- **check** (P) is not supported in MySQL (simply ignored)

From university_database.sql

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

Not Null and Unique Constraints

- **not null**
  - Declare *building* and *budget* to be **not null**

    building **varchar**(15) **not null**
    budget **numeric**(12,2) **not null**

- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes A_1, A_2, …, A_m form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

Referential Integrity

- Referential Integrity:
  If you refer to something, this something must exists.

- Referential Integrity ensures that a value that appears in one relation also appears in another relation.
  - Example: If "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology".

instructor

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

department

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

instructor

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

department

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

From university_database.sql

```
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                     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 (dept_name)
                on delete set null
        );
```

Cascading Actions in Referential Integrity

- **create table** *course (*
  course_id **char**(5) **primary key**,
  title **varchar**(20),
  dept_name **varchar**(20) **references** *department (dept_name)*
)

- **create table** *course* (
  …
  dept_name **varchar**(20),
  foreign key (*dept_name*) **references** *department (dept_name)*
  on delete cascade
  on update cascade,
  . . .
)
- alternative actions to cascade: **set null**, **set default**

MySQL requirement

Integrity Constraint Violation During Transactions

- E.g.

  create table *person* (
  ID **char**(10),
  name **char**(40),
  father **char**(10),
  mother **char**(10),
  primary key *ID,*
  foreign key *father* **references** *person,*
  foreign key *mother* **references** *person*)

- How to insert the first person?
- How to insert a tuple without causing constraint violation ?

A note on Index Creation

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0)

- **create index** *studentID_index* **on** *student*(*ID*)

- Indices are data structures used to speed up access to records with specified values for index attributes
  - e.g. **select ***
    from *student*
    where *ID* = '12345'

  can be executed by using the index to find the required record, without looking at all records of *student*

  More on indexing in later

A note on Index Creation

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))

With this …

This is redundant (An index is always created on primary key)

- **create index** *studentID_index* **on** *student*(*ID*)

but this will imply improved query performance

- **create index** *student_name_index* **on** *student*(*name*)

what would be implications on update performance?

More on indexing in later

Authorization

Forms of authorization on the **database content**:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the **database schema**

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization
  grant <privilege list>
  on <relation name or view name> **to** <user list>

- <privilege list> items:
  - **select:** allows read access to relation,or the ability to query using the view
  - **insert**: the ability to insert tuples
  - **update**: the ability to update using the SQL update statement
  - **delete**: the ability to delete tuples.
  - **all privileges**: used as a short form for all the allowable privileges

- <user list> is:
  - a list of user-id's
  - **public**, which allows all valid users the privilege granted

- Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:
  grant select on *instructor* **to** U_1, U_2, U_3

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list>

  on <relation name or view name> **from** <user list>

  - <privilege-list> may be **all** to revoke all privileges the revokee may hold.
  - If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.

- Examples:

  revoke select on *instructor* **from** U_1, U_2, U_3

  revoke all on *instructor* **from** U_1, U_2, U_3

  revoke select on *instructor* **from public**

- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Transfer of privileges

- Grant with grant option
  - **grant select on** *department* **to** Amit **with grant option**;
  - Amit can now
    - **grant select on** *department* **to** Satoshi;

- Trying
  - **revoke select on** *department* **from** Amit **restrict**;
  - would fail
- while
  - **revoke select on** *department* **from** Amit **cascade**;
  - would revoke the privileges from Amit as well as Satoshi

Authorization-grant graph

- To visualize grants of privileges
  - Users are nodes, grants are directed edges
  - One graph per privilege