Database Systems

Lecture 9: Intermediate SQL (part 1)

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Outline

- **Join Expressions**
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause

Join operations – Example

- 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

Joined Relations – Examples

- *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

Join operations – Example

- 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

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 relation to the result of the join.
- Uses *null* values.

Left Outer Join

- *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*

Right Outer Join

- *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

Full Outer Join

- *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

Inner Join

- *course* **natural inner join** *prereq*

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

- The default join type, when the join clause is used without the outer prefix is the inner join.

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **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 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

- *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?

- Alternative:

course, prereq **where**

course.course_id = prereq.course_id

Joined Relations – Examples

- *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

- *course* **full 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-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Outline

- Join Expressions
- **Views**
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization

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.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

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.*

View Definition

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.

- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, 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*

- Using views in SQL queries:
- Find all instructors in the Biology department

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

Example Views

- 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/another view

create view *physics_fall_2009_watson* **as**
(**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’;

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be *recursive* if it depends on itself.

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  repeat
    Find any view relation v_i in e_1
    Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate

Update of a View

- Views present serious problems if we express updates, insertions, or deletions with them.
- The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.

- Add a new tuple to *faculty* view which we defined earlier

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

This insertion must be represented by the insertion of the tuple ('30765', 'Green', 'Music', null) into the *instructor* relation

Some Updates cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor*.*dept_name*= *department*.*dept_name*;

- **insert into** *instructor_info* **values** ('69987', 'White', 'Taylor');
  - which department, if multiple departments in Taylor?
  - what if no department is in Taylor?
  - what happen if we add the following tuples to the *instructor* and *department* relations?

  ('69987', 'White', null, null) into *instructor*
  (null, 'Taylor', null) into *department*

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.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.

And Some Not at All

- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= 'History';

- What happens if we insert ('25566', 'Brown', 'Biology', 100000) into *history_instructors?*

NOTE:

By default, SQL would allow the above update to proceed. However, views can be defined with a check option clause at the end of the view definition; then, if a tuple inserted into the view does not satisfy the view's where clause condition, the insertion is rejected by the database system. Updates are similarly rejected if the new value does not satisfy the where clause conditions

Materialized Views

- **Materializing a view**: create a physical table containing all the tuples in the result of the query defining the view
  - Such views called **Materialized view**
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.
  - The process of keeping the materialized view up-to-date is called **materialized view maintenance** (or just **view maintenance**)

Materialized Views Maintenance

- Maintaining a view can be done in different ways
  - View maintenance can be done immediately when any of the relations on which the view is defined is updated.
  - View maintenance can be performed lazily, when the view is accessed.
  - Some systems update materialized views only periodically

Outline

- Join Expressions
- Views
- **Transactions**
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization

Transactions

- Consists of a sequence of query and/or update statements.
- Atomic transaction
- Either fully executed or rolled back as if it never occurred

- Transactions begin implicitly and ended by one of the following
  - **Commit work** commits the current transaction
    - Making the updates performed by the transaction become permanent in the database.
    - After the transaction is committed, a new transaction is automatically started.
  - **Rollback work** causes the current transaction to be rolled back
    - It undoes all the updates performed by the SQL statements in the transaction.
    - Thus, the database state is restored to what it was before the first statement of the transaction was executed.

Transactions

- By default most databases commit each SQL statement automatically as a transaction
  - Can turn off auto commit for a session (e.g. using API)
  - In SQL:1999
    - **begin atomic** …. **end**
    - But not supported on most databases

- Further reading for transactions: Chapter 14

Outline

- Join Expressions
- Views
- Transactions
- **Integrity Constraints**
- SQL Data Types and Schemas
- Authorization

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

Integrity Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

Not Null and Unique Constraints

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

    name **varchar**(20) **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).

The check clause

- **check** (P)

  where P is a predicate

  Example: ensure that semester is one of fall, winter, spring or summer:

  create table *section* (
  course_id **varchar** (8),
  sec_id **varchar** (8),
  semester **varchar** (6),
  year **numeric** (4,0),
  building **varchar** (15),
  room_number **varchar** (7),
  time slot id **varchar** (4),
  primary key (*course_id*, *sec_id*, *semester*, *year*),
  check (*semester* **in** ('Fall', 'Winter', 'Spring', 'Summer'))
  );

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes 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".

- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

Cascading Actions in Referential Integrity

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

- **create table** *course* (
  …
  dept_name **varchar**(20),
  foreign key (*dept_name*) **references** *department*
  on delete cascade
  on update cascade,
  . . .)

- alternative actions to cascade: **set null**, **set default**

Integrity Constraint Violation During Transactions

- E.g.

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

- How to insert a tuple without causing constraint violation ?
  - insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking

Complex Check Clauses

- **check** (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))
  - should be check by any changes in *time_slot* table as well
- Every section has at least one instructor teaching the section.
  - how to write this?
- **create assertion** <assertion-name> **check** <predicate>;
  - introduce complex overhead
- Unfortunately: subquery in check clause not supported by pretty much any database
  - Alternative: triggers (later)

The picture can't be displayed.

Questions?