Intermediate SQL

CS3043 - Database Systems

Department of Computer Science and Engineering
University of Moratuwa

Overview

- Joins
  - Outer joins
  - Inner joins
- Views
- Transactions
- Integrity Constraints
- Referential Integrity
- Built-in Data Types
- User Defined Data Types
- Domains
- Large Object Data Types
- Indexing

Joined Relations

- **Join operations** take two relations and return another relation as a result.
- 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.

Examples:

```
SELECT *
FROM course LEFT OUTER JOIN prereq
ON course.course_id = prereq.prereq_id
```

```
SELECT *
FROM course INNER JOIN prereq
USING (course_id)
```

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.

Join operations - Example

- course relation

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 relation

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

- Observe that,
  - information of CS-315 is missing on prereq
  - information of CS-347 is missing on course

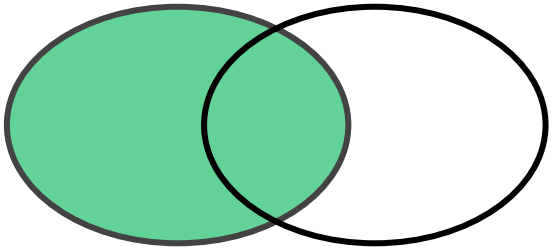
LEFT OUTER JOIN

- Returns **all the rows from the left table** with matching rows from right table. If there is no match in the right table, those values will be null.

Examples:

```
SELECT *
FROM course LEFT OUTER JOIN prereq
ON course.course_id = prereq.course_id
```

```
SELECT *
FROM course LEFT OUTER JOIN prereq
USING (course_id)
```

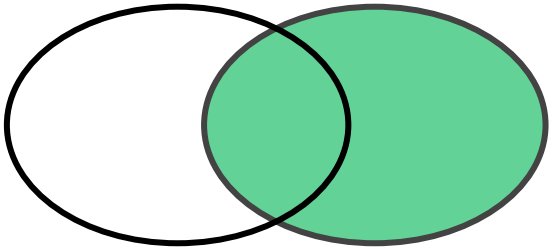


course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

RIGHT OUTER JOIN

- Returns **all the rows from the right table** with matching rows from left table. If there is no match in the left table, those values will be null.

Examples:



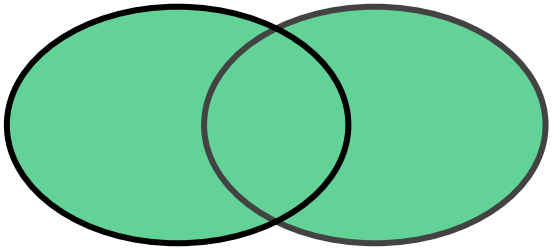
```
SELECT *
FROM course RIGHT OUTER JOIN prereq
ON course.course_id = prereq.course_id
```

```
SELECT *
FROM course RIGHT OUTER JOIN prereq
USING (course_id)
```

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

FULL OUTER JOIN

- Returns **all the rows from the right table and all the rows from the left table**. Missing values will be null.



Examples:

```
SELECT *
FROM course FULL OUTER JOIN prereq
ON course.course_id = prereq.course_id
```

- Not supported in MySQL

How do you emulate FULL OUTER JOIN in MySQL using LEFT and RIGHT JOIN?

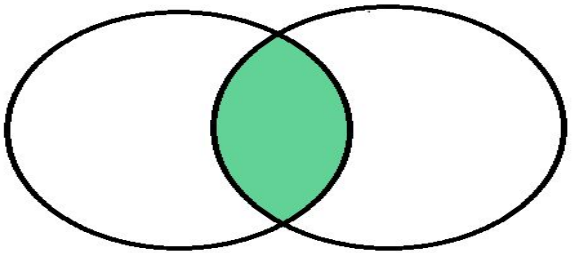
INNER JOIN

- Returns the rows when there is **a match in both tables.**

Examples:

```
SELECT *
FROM course INNER JOIN prereq
ON course.course_id = prereq.course_id

SELECT *
FROM course INNER JOIN prereq
USING (course_id)
```



Writing some queries

Instructor(ID, name, dept_name, salary)

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

Teaches(ID, course_id, sec_id, semester, year

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

Using JOINS only,

1. Find the names of instructors who do not teach any courses

```
select name
from instructor left outer join teaches using(ID)
where course_id is null;
```

2. Find the names of instructors who teach at least one course (no duplicates)

```
select distinct name
from instructor inner join teaches using(ID);
```

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 instructor's 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 >

  v - name of the view
  <query expression> - any legal SQL expression

- 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 _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*;

View dependencies

- One view may be used in the expression defining another view.
- A view relation v_2 is said to depend directly on a view relation v_1, if v_1 is used in the expression defining v_2

$$V_1 \leftarrow V_2$$

- A view relation v_2 is said to depend on view relation v_1 if either v_2 depends directly on v_1 or there is a path of dependencies from v_2 to v_1

$V_1 \leftarrow V_2$ or $V_1 \leftarrow \ldots \leftarrow V_2$

- A view relation v is said to be recursive if it depends on itself.

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

- A way to define the meaning of views defined in terms of other views.
- Let the view v_1 be defined by an expression e_1 that may itself contains 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.

Updating a View

- For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table.
- 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.

Example:

INSERT INTO faculty **VALUES** (30765, ‘Green’, ‘Music’);
⇒ Query OK

INSERT INTO dept_total_salary **VALUES** ('Nuclear', 299000.00);
⇒ ERROR : The target table dept_total_salary of the INSERT is not insertable-into

Materialized Views

- A logical view of the data driven by a SELECT query. But creates a physical relation/table containing all the tuples in the result of the query defining the view.
- If relations used in the query are updated, the materialized view (MV) result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.
- **Performance is higher** than Views
  - Instead of running the expanded query against the database every time, MV acquire the results from a physical table

- Not supported in MySQL

Transactions

- Transaction is a **Unit of work** that is performed against a database.
- Have following four properties
  - **Atomicity** - the whole sequence of actions within a transaction is either fully executed or, in case of any exception, rolled back as if it never occurred.
  - **Consistency** - ensures that the database properly changes states upon a successfully committed transaction.
  - **Isolation** - enables transactions to occur independently and transparent of each other.
  - **Durability** - ensures that the result or effect of a committed transaction persists in case of a system failure.
- By default on most databases each SQL statement commits automatically
  - Can turn off auto commit for a session

```
START TRANSACTION;
    <sequence of sql actions>
COMMIT;
```

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.
- Example constraints:
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $40 an hour
  - A customer must have a (non-null) phone number

Integrity Constraints

- **NOT NULL**
  - declare certain attributes to be not null
    Example:
    name varchar(20) **NOT NULL**,
    budget numeric(12,2) **NOT NULL**,
- **PRIMARY KEY**
  - declare the primary key of the relation
    Example:
    PRIMARY KEY (ID)
  - primary key cannot be null
- **UNIQUE**
  - 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).
    Example:
    UNIQUE (course_id, sec_id)

Integrity Constraints

- **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 _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'))
);
```

Integrity Constraints - 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 *course* 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.

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

Cascading Actions in Referential Integrity

- A foreign key with cascade delete/update in a child table means that **if a record in the parent table is deleted/updated, then the corresponding records in the child table will automatically be deleted/updated**.

```
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

- Example:

  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 and use transactions*

Built-in Data Types in SQL

- **date:** Dates, containing a year, month and date
  Example: **date** ‘2005-7-27’
- **time:** Time of day in hours minutes and seconds , in hours, minutes and seconds.
  Example: **time** ‘09:00:30’
  time ‘09:00:30.75’
- **timestamp**: date plus time of day
  Example: **timestamp** ‘2005-7-27 09:00:30.75’
- **interval:** period of time
  Example: interval ‘1’ day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

User Defined Types

- **create type** construct in SQL creates user-defined type

  create type *Dollars* **as numeric (12,2) final**

  Using the custom data type:

  create table *department(*
  dept_name **varchar** (20),
  building **varchar** (15),
  budget Dollars
)

Specify **FINAL** if no subtypes can be created for this type. (default)

Specify **NOT FINAL** if further subtypes can be defined for this type.

Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.

  create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** (’Bachelors’, ’Masters’, ’Doctorate’));

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object - object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object - object is a large collection of character data
  - When a query returns a large object, a pointer is returned rather than the large object itself.

Indexing

- Indices are data structures used to speed up access to records with specified values for index attributes. (Indices are used to find rows with specific column values quickly.)

  CREATE INDEX *student_dept_name_index* **ON** *student*(*dept_name*);

  Example scenario:

  select *
  from *student*
  where *dept_name = “Physics”*

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

- The users cannot see the indexes, they are just used to speed up searches/queries.
- Primary key is indexed by default in many implementations (MySQL)
- More on indices in Chapter 11

Indexing

- Index is a data structure which is implemented using
  - B-Trees
  - Hash tables
  - R-Trees
- Advantages of using an index
  - Faster search time by not having to scan entire table
  - Specially when running SELECT queries or JOINS
- Disadvantages
  - Takes up space - larger the number of rows, larger the index size
  - Need to update the index, when rows are added, deleted or updated
- Best practices
  - Indices should only be used if the data in the indexed column is queried frequently
  - Index columns that are being used as foreign keys in other tables
  - Add additional indices based on performance requirements
  - Do not index every column

Sample entry in an index
(“BIO-319”, 0x562189)

Thank you!

Truncate and Delete

- Truncate
  - classified as a DDL statement
  - empties the table completely
  - drop and re-create the table, which is much faster than deleting rows one by one
  - does not invoke ON DELETE triggers
  - cannot be rolled back on MySQL
- Delete
  - classified as a DML statement
  - deletes row by row (can specify conditions with where clause)
  - operations are logged individually
  - invokes on delete triggers
  - can be rolled back