No Slides.

Joins

The basic form of queries is actually:

```
SELECT A1, A2, ..., An
FROM R1, R2, ..., Rn
WHERE P;
```

- $A_i$ represents an attribute
- $R_i$ represents a relation
- $P$ is a predicate

**CROSS PRODUCT**

If you list two Relations, R1 and R2, the DB performs a cross product of the two relations, including all attributes from both relations.

```
SELECT * FROM College, Tryout;
```

product (a.k.a. Cartesian Product) is generally only useful when combined with a WHERE.

**JOIN Operations**

JOIN operations take two relations and return another relation as the result. JOINs are typically used as subquery expressions in the FROM clause.

* JOIN condition - determines which tuples match, and with which attributes
NATURAL, ON <predicate>, USING (A1, A2, ...

JOIN type - specifies what to do with the unmatched tuples

INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN (note FULL is not supported by MySQL)

Outer is used to avoid the loss of any information. It adds in tuples from one relation that doesn’t match tuples in the other, which one is decided by LEFT or RIGHT.

**SIMPLE JOIN**

Same as Cross Product but with a WHERE clause that matches equal keys in each relation.

```sql
SELECT * FROM Player, Tryout WHERE Player.pID=Tryout.pID;
SELECT * FROM Player, Tryout WHERE Player.pID=Tryout.pID AND decision="yes";
```

that the pID’s from both relations are included in the resulting relation.

**NATURAL JOIN**

This join matches tuples with the same values for all common attributes, keeping only one copy of each common attribute.

This is the same as SIMPLE JOIN except that only one copy of the attribute that was used for the JOIN is included in the resulting relation.

```sql
SELECT * FROM Player NATURAL JOIN Tryout;
SELECT * FROM Player NATURAL JOIN Tryout WHERE decision="yes";
```

that the NATURAL JOIN has the same result as the REGULAR JOIN with the explicit pID=pID condition.

```sql
SELECT * FROM Player, Tryout WHERE Player.pID=Tryout.pID;
SELECT * FROM Player NATURAL JOIN Tryout;
```

**OUTER JOINS**

```sql
SELECT * FROM course natural left outer join prereq;
SELECT * FROM course natural right outer join prereq;
```

doesn’t have FULL OUTER JOIN, so to get that you just UNION a LEFT and a RIGHT.

```sql
### UN-NATURAL JOINs

Compare this to the `LEFT OUTER JOIN`:
**Rename using AS**

Attributes of relations may be renamed to avoid conflicts

```sql
SELECT pID, pName, HS/1000 FROM Player;
SELECT pID, pName, HS/1000 AS HSScaled FROM Player;

/* Rename is especially useful with Self-Joins */
/* Find the names of all the players who have yCards AND came from a HS smaller than some other Player*/
SELECT * FROM Player;
SELECT DISTINCT A. pName
FROM Player as A , Player as B
WHERE A.yCard = "yes" AND A.HS < B.HS;
```

**String functions**

LIKE: ‘%’ matches 0 or more characters, ‘_’ matches exactly 1 character
and these patterns are case-sensitive

```sql
SELECT DISTINCT pName FROM Player WHERE pName LIKE '%a%';
SELECT DISTINCT pName FROM Player WHERE pName LIKE '__a%';
```

supports a variety of other string operations such as:
* concatenation (using “||”)
* converting from upper to lower case (and vice versa)
* finding string length, extracting substrings, etc.

**BETWEEN operator**

SELECT pName FROM Player WHERE HS between 900 AND 1500;

**Comparison of tuples (including subsets of tuples)**

SELECT pName, pPos FROM Player, Tryout
WHERE (Player.pID, yCard) = (Tryout.pID, "yes");

**Set operations**

For these operations the two “sets” must be **UNION Compatible**

**UNION**

```sql
SELECT cName, enr, state FROM College WHERE enr > 12000 AND cName LIKE "_SU";
SELECT cName, enr, state FROM College WHERE enr > 10000 AND state="LA";
(SELECT cName, enr, state FROM College WHERE enr > 12000 AND cName LIKE "_SU")
MySQL doesn't support INTERSECT, so we do it this way:

```sql
SELECT cName FROM Tryout WHERE pPos="goalie" AND (cName) IN
(SELECT cName FROM Tryout WHERE decision="no");
```

that this example returns two results when we expected one ... why?

MySQL doesn't support EXCEPT or MINUS, so we do it this way:

```sql
SELECT cName FROM Tryout WHERE pPos="goalie" AND (cName) NOT IN
(SELECT cName FROM Tryout WHERE decision="yes");
```

that we're “subtracting” more than we have ... it's ok

MOVE TO UNIVERSITY.SQL (From Silberschatz, et.al.)

Explain with some `SELECT *`'s

**SubQuerries in FROM**

Find the average instructors’ salaries of those departments where the average salary is greater than $42,000

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

**NOT EXISTS**

Find all students who have taken all courses offered in the Biology department.

*Note*: this will not work in MySQL due to the `EXCEPT`.

```sql
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
from course
where dept_name = 'Biology')
except
```
WITH

This clause provides a way of defining a temporary view whose definition is available only to the query in which the with clause occurs.

Note: this will not work in MySQL due to lack of WITH support.

```sql
/* Find all departments with the maximum allowed budget */
with max_budget (value) as
    (select max(budget) from department)
select budget from department, max_budget
where department.budget = max_budget.value;
```

Scalar subquery

These may be used when a single value is expected.

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

Case Statement for conditional updates to the DB

```sql
update instructor
    set salary = case
        when salary <= 100000 then salary * 1.05
        else salary * 1.03
    end
```

VIEWS and updatable VIEWS

There may be times when it is inappropriate for some users to see every attribute in every table.

- consider a student who needs to know an instructor’s name, department, and office number, but not the salary. They need to see something like
SELECT ID, name, dept_name, office
FROM instructor;

SQL view provides a mechanism for limiting the attributes that certain users may see.

The view can be based on any valid SQL expression.

Once defined, the view can be used to refer to the virtual relation it generates.

This does not create a new relation/table, it essentially just saves an expression.

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;

Can be defined in terms of 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';

Views are created as actual physical tables.
- this is done primarily for performance reasons
- it is potentially a maintenance problem since the view must be updated whenever the underlying relations are updated ... and you will forget about the materialized view!

Authorization in SQL

SELECT, INSERT, UPDATE, and DELETE authorizations.
Schema mods include INDEX, RESOURCES (new relations), ALTER, and DROP
Use **grant** to give authorization

```
GRANT SELECT ON Tryout TO user1, user2, ...;
GRANT SELECT ON Student TO PUBLIC;
GRANT SELECT ON cs_instructor TO user1, user2, ...; /* where cs_instructor is a view */
GRANT SELECT ON Tryout to "ROLE";
GRANT ALL PRIVILEGES to CCPalmer;
```

**revoke** undoes **grant**.

In other SQL’s, there is **create role**
Then you grant privileges to that **role**
and then you can **grant role to** a person or a relation.

and even grant the right to grant, as in
**grant select on Player to ccpalmer with grant option**;

## TRIGGERS

A Trigger is a SQL statement that is automatically executed by the DBMS as a side effect of a modification to the db.

It can be set to run before OR after the mod.

```
CREATE TABLE players_audit (  
    id      int(11) NOT NULL AUTO_INCREMENT,  
    pID     int(11) NOT NULL,  
    pName   varchar(50) NOT NULL,  
    changedon datetime DEFAULT NULL,  
    action  varchar(50) DEFAULT NULL,  
    PRIMARY KEY (id) 
);  
DELIMITER $$
CREATE TRIGGER before_player_update  
BEFORE UPDATE ON Player  
FOR EACH ROW BEGIN  
    INSERT INTO players_audit  
    SET action = 'update',  
    pID = OLD.pID,  
    pName = OLD.pName,  
    changedon = NOW();  
END$$
DELIMITER ;
```

MariaDB [ccptestdb]> **SELECT * FROM Player;**

<table>
<thead>
<tr>
<th>pID</th>
<th>pName</th>
<th>yCard</th>
<th>HS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>Andrew</td>
<td>no</td>
<td>1200</td>
</tr>
<tr>
<td>20002</td>
<td>Blake</td>
<td>no</td>
<td>1600</td>
</tr>
<tr>
<td>30003</td>
<td>Charles</td>
<td>no</td>
<td>600</td>
</tr>
<tr>
<td>40004</td>
<td>David</td>
<td>yes</td>
<td>1600</td>
</tr>
<tr>
<td>40004</td>
<td>David</td>
<td>yes</td>
<td>1600</td>
</tr>
<tr>
<td>50005</td>
<td>Eddie</td>
<td>yes</td>
<td>300</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
</tbody>
</table>
6 rows in set (0.01 sec)

MariaDB [ccptestdb]> **UPDATE** Player
-> **SET** pName = 'Bryan'
-> **WHERE** pID = 20002;
Query OK, 1 row affected (0.03 sec)
Rows matched: 1 Changed: 1 Warnings: 0

MariaDB [ccptestdb]> **SELECT** * FROM Player;

<table>
<thead>
<tr>
<th>pID</th>
<th>pName</th>
<th>yCard</th>
<th>HS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10001</td>
<td>Andrew</td>
<td>no</td>
<td>1200</td>
</tr>
<tr>
<td>20002</td>
<td>Bryan</td>
<td>no</td>
<td>1600</td>
</tr>
<tr>
<td>30003</td>
<td>Charles</td>
<td>no</td>
<td>600</td>
</tr>
<tr>
<td>40004</td>
<td>David</td>
<td>yes</td>
<td>1600</td>
</tr>
<tr>
<td>40004</td>
<td>David</td>
<td>yes</td>
<td>1600</td>
</tr>
<tr>
<td>50005</td>
<td>Eddie</td>
<td>yes</td>
<td>300</td>
</tr>
</tbody>
</table>
6 rows in set (0.00 sec)

MariaDB [ccptestdb]> **select** * from players_audit;

<table>
<thead>
<tr>
<th>id</th>
<th>pID</th>
<th>pName</th>
<th>changedon</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20002</td>
<td>Blake</td>
<td>2015-04-27 00:25:34</td>
<td>update</td>
</tr>
</tbody>
</table>
1 row in set (0.00 sec)

You can drop Triggers, but you can’t ALTER them.

You can list them, but it’s tough to understand the output:

```
SELECT * from Information_Schema.Triggers where Trigger_schema = 'ccptestdb' AND Trigger_name = 'before_player_update'
```

You can also create them to run automatically on some schedule.

**STORED PROCEDURES**

A stored procedure is a set of SQL statements stored inside the database. It can be invoked at the command line, by triggers, by other procedures, or by application code (e.g., C, Java, etc.).

Procedures and Functions are all about readability and usability!

**Advantages**
- performance
- reduces traffic since table entries don’t have to be sent to the calling application, just a few parameters come in and all the table
entries are worked on in the database
- they are reusable, making it easier for appl's to use the db
- they are secured by the DBA

Disadvantages
- memory utilization of the DB may increase
- CPU utilization of the DB may increase
No surprise here - the DB is doing the appl's work!
- They are tough to debug

You have to use the DELIMITER command since you will need semi-colons inside!

```sql
DELIMITER //
CREATE PROCEDURE GetAllPlayers()
BEGIN
    SELECT * 
    FROM Player;
END//
DELIMITER ;
```

later, perhaps at the command line:

```sql
call GetAllPlayers();
```

you get the results from the SELECT.

You can declare and use variables inside the PROCEDURE:

```sql
DECLARE total_count INT DEFAULT 0
SET total_count = 10;
```

you can SELECT INTO a variable:

```sql
DECLARE total_count INT DEFAULT 0
SELECT COUNT(*) INTO total_count 
FROM inventory;
```

You can pass parameters to them:

```sql
DELIMITER $$
CREATE PROCEDURE CountPlayersByStatus(
    IN pStatus VARCHAR(25),
    OUT total INT)
BEGIN
    SELECT count(pStatus)
```
STORED PROCEDURES

```
INTO total
FROM Tryout
WHERE decision = pStatus;
END$
DELIMITER ;
```

```
CALL CountPlayersByStatus("no",@totalno);
SELECT @totalno;
+----------+
| @totalno |
+----------+
| 4        |
+----------+
```

**Stored functions**

These are simpler Procedures that return a single value.

Specify DETERMINISTIC if the function always returns the same value for the same input. This enables optimization by the DBMS. NOT DETERMINISTIC is the non-default alternative.

```
DELIMITER $$
CREATE FUNCTION StudentClass(s_credits integer) RETURNS VARCHAR(10)
DETERMINISTIC
BEGIN
    DECLARE lvl varchar(10);
    IF s_credits > 100 THEN
        SET lvl = 'SENIOR';
    ELSEIF (s_credits <= 100 AND s_credits > 60) THEN
        SET lvl = 'JUNIOR';
    ELSEIF (s_credits <= 60 AND s_credits > 30) THEN
        SET lvl = 'SOPHOMORE';
    ELSEIF p_creditLimit <= 30 THEN
        SET lvl = 'FRESHPERSON';
    END IF;
    RETURN (lvl);
END
```

it's used like this:

```
SELECT name, StudentClass(tot_cred)
FROM student;
```

**Embedded SQL**

See the dev_guide.pdf in our Resources directory.
1. Lecture notes based on texts by Coronel, Widom, Ullman, Jukic, and Silberschatz.