SQL
Part II: Advanced Queries

Aggregation

sid	sname	rating	age
28	yuppy	7	15
31	debby	7	10
22	conny	5	10
58	lilly	10	13

- Significant extension of relational algebra

- *"Count the number of tuples in Skaters"*

```
SELECT COUNT(*)
FROM skaters
```

count
4

- *"Count how many different ratings?"*

```
SELECT COUNT(DISTINCT rating)
FROM skaters
```

count
3

Result is a relation with only one tuple

Aggregation

sid	sname	rating	age
28	yuppy	7	15
31	debby	7	10
22	conny	5	10
58	lilly	10	13

- Syntax: **`COUNT, SUM, AVG, MAX, MIN`** apply to single attribute/column.
- Additionally, **`COUNT(*)`**

- *"What is the average age of skaters with rating 7?"*

```
SELECT AVG(age)
FROM skaters
WHERE rating = 7
```

avg
12.50

- *What is the average age of skaters with rating 7, and how many are there?"*

```
SELECT AVG(age), COUNT(*)
FROM skaters
WHERE rating = 7
```

avg	count
12.50	2

Aggregation

sid	sname	rating	age
28	yuppy	7	15
31	debby	7	10
22	conny	5	10
58	lilly	10	13

- *"Give the names of the skaters with the highest rankings"*

```
SELECT sname
 FROM skaters
 WHERE rating = (SELECT MAX(rating)
                         FROM skaters)
```

sname
lilly

(Note also the = in the **where** clause. We can use direct comparison (=, <, …) when it is assured that the relation resulting from the subquery has only one tuple.)

- *"Give the name of the skater that is the first in the alphabet"*

```
SELECT min(sname)
FROM Skaters S
```

min
conny

Wrong Aggregations

- *"Give the names of the skaters with the highest rankings"*

```
SELECT sname, MAX(rating)
 FROM skaters
```

- Does not work!
- `Max` is one value for ALL tuples, sname is one value for each tuple

Grouping

- So far, we have applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several <u>groups</u> of tuples.
- Example: *"Find the average age of the skaters in each rating level"*
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are.
  - Suppose, we know that the rating levels go from 1 to 10; then we can write 10 queries that look like this:

<u>sid</u>	sname	rating	age
28	yuppy	7	15
31	debby	7	10
22	conny	10	11
58	lilly	10	13

For i = 1, 2, … 10

```
SELECT AVG(age)
FROM skaters
WHERE rating = i
```

avg
12.0

Grouping

- Grouping does this with one query

```
SELECT AVG(age), MIN(age)
FROM skaters
GROUP BY rating
```

sid	sname	rating	age
28	yuppy	7	15
31	debby	7	10
22	conny	10	10
58	lilly	10	13

avg	min
12.5	10
11.5	10

Queries with GROUP BY

```
SELECT target-list
FROM   relation list
WHERE  qualification
GROUP BY grouping list
```

- A group is defined as a set of tuples that have the same value for all attributes in the grouping list
- One answer tuple is generated per group.
- The target-list contains aggregation terms and/or *attributes*
- Allowed attributes:
  - Subset of the grouping list
  - Since each answer tuple corresponds to one group, we can only depict attributes, for which all tuples in the group have the same value

- Example:

```
SELECT rating, MIN(age)
FROM skaters
GROUP BY rating
```

Queries with GROUP BY

```
SELECT    target-list
FROM   relation list
WHERE     qualification
GROUP BY grouping list
(ORDER BY…)
```

```
  SELECT rating, MIN(age)
  FROM Skaters
  WHERE rating > 5
  GROUP BY rating
  ORDER BY rating
```

sid	sname	rating	age
28	yuppy	7	15
31	debby	7	10
22	conny	10	10
58	lilly	10	13
25	Conny	5	5

rating	min
7	10
10	10

Evaluation

```
SELECT target-list
FROM   relation list
(WHERE    qualification)
GROUP BY grouping list
(ORDER BY attributes from target-list)
```

- Conversion to Relational Algebra
  - Compute the cross-product of relations in **FROM** clause, consider only tuples that fulfill the qualification in **WHERE** clause, project on fields that are needed (in **SELECT** or **GROUP BY**)
  - Partition the filtered tuples into groups by the value of attributes in grouping-list
  - Return all attributes in the **SELECT** clause (must also be in the group list) plus the calculated aggregation terms per group.
  - Return in order if requested

SELECT lists with aggregation

- If any aggregation is used, then each element in the attribute list of the **SELECT** clause must either be aggregated or appear in a group-by clause

```
SELECT rating, MIN(age)
FROM Skater
GROUP BY rating
```

- Wrong way to find the name of the oldest skaters

```
SELECT sname, MAX(age)
FROM Skaters
```

- Correct way to find the names of the oldest skaters

```
SELECT sname, age
FROM skaters
WHERE age = (SELECT MAX(age)
             FROM skaters)
```

HAVING CLAUSE

- **HAVING** clauses are selections on groups, just as **WHERE** clauses are selections on tuples
- Example: "*For each rating, find the minimum age of the skaters with this rating. Only consider rating levels with at least two skaters*

```
 SELECT rating, MIN(age)
 FROM Skaters
GROUP BY rating
HAVING COUNT(*) >= 2
```

- Example 2: "For each rating > 5, find the average age of the skaters with this rating. Only consider rating levels where there are at least two skaters

```
SELECT rating, AVG(age)
 FROM Skaters
WHERE rating > 5
GROUP BY rating
HAVING COUNT(*) >= 2
```

sid	sname	rating	age
1	A	9	18
2	B	1	20
3	C	6	12
4	D	9	18
5	E	1	10
6	F	8	16
7	G	8	8

```
SELECT rating, avg(age)
   FROM Skaters
   WHERE rating > 5
   GROUP BY rating
   HAVING COUNT(*) >= 2
```

- Select upon **WHERE** and project to necessary attributes
- Partition by **GROUP** and check whether they fulfill **HAVING**

rating	age
9	18
6	12
9	18
8	16
8	10

rating	age
9	18
8	13

sid	sname	rating	age
1	A	9	18
2	B	1	20
3	C	6	12
4	D	9	18
5	E	1	10
6	F	8	16
7	G	8	8

```
SELECT rating, age, count(*)
   FROM Skaters
  WHERE rating > 5
  GROUP BY rating, age
  HAVING COUNT(*) >= 2
```

rating	age	count
9	18	2

sid	location	rating	age
1	Montreal	9	18
2	Toronto	1	20
3	Ottawa	6	12
4	Montreal	9	18
5	Montreal	1	10
6	Laval	8	16
7	Laval	8	8

```
SELECT rating, age, count(*)
   FROM Skaters
   WHERE location = 'Montreal'
   GROUP BY rating, age
   HAVING COUNT(*) >= 2
```

rating	age	count
9	18	2

Evaluation

```
SELECT rating, avg(age)        SELECT target-list
FROM Skaters                   FROM relation list
WHERE rating > 5               WHERE qualification
GROUP BY rating                GROUP BY grouping list
HAVING COUNT(*) >= 2           HAVING group-qualification
```

- Conversion to Relational Algebra
  - Compute the partial query result considering the relations in **FROM** clause, the conditions specified in **WHERE** clause and only considering the attributes that are needed (in **SELECT** or **GROUP BY**)
  - NOTE: the WHERE clause can contain any attributes of the relations in the relation list (it is considered before any grouping is executed

    ```
    SELECT rating, MIN(age)
    FROM Skaters
    WHERE sname LIKE 'A%'
    GROUP BY rating
    HAVING COUNT(*) >= 2
    ```

  - Partition the remaining tuples into groups by the value of attributes in grouping-list
  - For each group, the group qualification in the **HAVING** clause are then checked and only those groups are kept that fulfill the conditions Expressions in group-qualification must have a single value per group. Hence, for each attribute that appears in the group qualification, either
    - the attribute also appears in the grouping list
    - or it is argument of an aggregation

Example II

- Grouping over a join of two relations.

- *For each local competition, find the number of participants*

```
SELECT  C.cid,  COUNT (*) AS scount
FROM  Competition C, Participates P
WHERE  C.cid=P.cid AND c.type='local'
GROUP BY  C.cid
```

Example III

- *Find those ratings for which the average age is the minimum over all rating, i.e., the minimum average age*

- Aggregate operations cannot be nested! WRONG

```
SELECT S.rating
FROM   Skaters S
WHERE  S.age =(SELECT  MIN (AVG (S2.age))
                FROM Skaters S2)
```

- First approach: complex having clause

```
SELECT S.rating, AVG(S.age)
FROM   Skaters S
GROUP BY S.rating
HAVING AVG(S.age) <= ALL (SELECT AVG(S2.age)
                           FROM Skaters S2
                           GROUP BY S2.rating)
```

- Second approach: views

Views

- A *view* is just an unmaterialized relation: we store a *definition* rather than a set of tuples.

```
CREATE VIEW ActiveSkaters (sid,sname)
   AS SELECT DISTINCT S.sid, S.sname
   FROM  Skaters S, Participates P
   WHERE S.sid = P.sid
```

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
  - Given ActiveSkaters, we know the names of the skaters who have participated in competition but not the *age* of the skaters (may be uninteresting for the users of ActiveSkaters).

Views

- Use a view as intermediate relation (rename in rel.algebra)
- *Find those ratings for which the average age is the minimum over all ratings*

```
CREATE VIEW Temp (rating, avgage)
 AS SELECT  rating, AVG (age) AS avgage
    FROM  skaters
    GROUP BY  rating)

SELECT rating, avgage
FROM Temp
 WHERE  avgage = (SELECT  MIN (avgage)
                 FROM  Temp)
```

Views (contd)

- Views can be treated in a SELECT statement as if they were just any other table (appear in the FROM clause)
- The system rewrites SELECTS statements that use a view in the FROM clause to SELECT statements that only use the underlying tables.
- Note that the DBS stores the view definition internally but does not execute the view query when the view is defined.
- Performing an update on a view is problematic as the data is not materialized
- Views can be dropped using the **`DROP VIEW`** command
  - How to handle **`DROP TABLE`** if there is a view on the table?
    - DROP TABLE command has options to let the user specifiy this.

WITH CLAUSE

Functions like the concept of a view for the scope of this SQL statement

```
WITH partinfo(sid, sname, age, rank, cid) AS
(
  SELECT S.sid, S.sname, S.age, P.rank, P.cid
  FROM skaters S
      INNER JOIN participates P
    ON S.sid = P.sid
)
SELECT sid, sname, cid
FROM partinfo
WHERE age > 7;
```

Can be any complex SQL Select query, with joins, aggregations, etc.

```
WITH A(...) AS
 (SELECT ...)
, B(...) AS
 (SELECT ...)
SELECT ...;
```

Can be any complex SQL Select query, with joins, aggregations, etc., including with other tables.

Possible to have multiple aliases defined

Derived Tables

```
SELECT sid, sname, cid
FROM
(
  SELECT S.sid, S.sname, S.age, P.rank, P.cid
  FROM skaters S
      INNER JOIN participates P
    ON S.sid = P.sid
)partinfo
WHERE age > 7;
```

Can be any complex SQL Select query, with joins, aggregations, etc.

Functions like the concept of a view for the scope of this SQL statement

The outer query can be any complex SQL Select query, with joins, aggregations, etc., including with other tables.

```
SELECT ...
FROM (SELECT ...)A
    ,(SELECT ...)B
WHERE ...
```

Possible to have multiple aliases defined

Can we use WITH clause and/or derived tables for our minimum average age rating groups?

Find those ratings for which the average age is the minimum over all ratings

```
With temp (rating, avgage) AS
  ( SELECT rating, AVG(age)
    FROM Skaters
    GROUP BY rating)
SELECT rating, avgage
FROM temp
WHERE avgage = (SELECT MIN(avgage) FROM temp);
```

Does it work?

```
SELECT rating, avgage
FROM (
   SELECT rating, AVG (age) AS avgage
   FROM Skaters
   GROUP BY rating) Temp
   WHERE  avgage = (SELECT  MIN(avgage)
                    FROM  Temp);
```

Does it work?

Inner Join (Default)

Dangling tuples: No match in the other relation → no output

```
SELECT *
FROM skaters s INNER JOIN participates p
ON s.sid = p.sid
```

Optional Keyword

skaters

sid	sname	rating	age
28	yuppy	9	15
31	debby	7	10
22	conny	5	10
58	lilly	10	13

participates

sid	cid	rank
31	101	2
58	103	7
58	101	7

S.Sid	sname	rating	age	p.sid	cid	rank
31	debby	7	10	31	101	2
58	lilly	10	13	58	103	7
58	lilly	10	13	58	101	7

Outer Join

Dangling tuples: No match in the other relation → One dummy record

```
SELECT *
FROM skaters s LEFT OUTER JOIN participates p
ON s.sid = p.sid
```

sid	sname	rating	age
28	yuppy	9	15
31	debby	7	10
22	conny	5	10
58	lilly	10	13

sid	cid	rank
31	101	2
58	103	7
58	101	7

S.Sid	sname	rating	age	p.sid	cid	rank
28	yuppy	9	15	NULL	NULL	NULL
31	debby	7	10	31	101	2
22	conny	5	10	NULL	NULL	NULL
58	lilly	10	13	58	103	7
58	lilly	10	13	58	101	7

Outer Join Types

```
SELECT *
FROM A LEFT OUTER JOIN B
ON A.att1 = B.att2
```

Pad dangling tuples from A

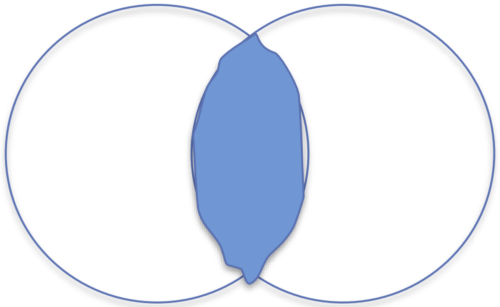
```
SELECT *
FROM A RIGHT OUTER JOIN B
ON A.att1 = B.att2
```

Pad dangling tuples from B

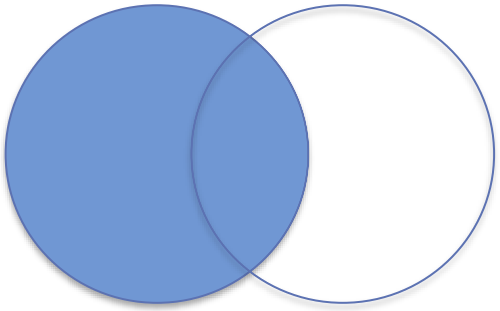
```
SELECT *
FROM A FULL OUTER JOIN B
ON A.att1 = B.att2
```

Pad dangling tuples from A and B

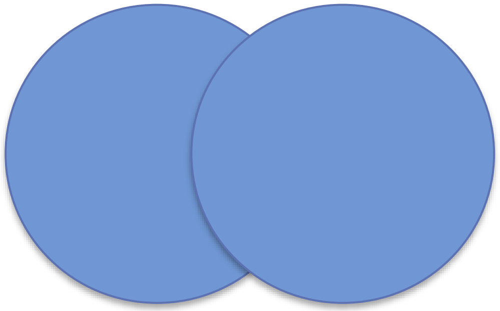
Visualization



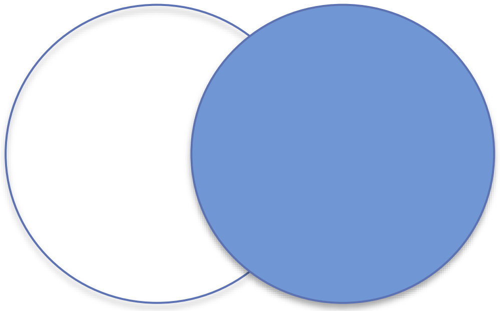
A INNER JOIN B



A LEFT OUTER JOIN



A FULL OUTER JOIN



A RIGHT OUTER JOIN

Foreign Key Constraints

- If B has NOT NULL foreign key referencing A
  - Every tuple in B joins with exactly one tuple in A
  - A INNER JOIN B = A RIGHT OUTER JOIN B
    - As no dangling tuples

NULL Values

- Meaning of a NULL value
  - Unknown/missing
  - Inapplicable (e.g., no spouse's name)
- Comparing NULLs to values
  - E.g., how to evaluate condition rating>7 if tuple has a NULL in rating?
  - When we compare a NULL value and any other value (including NULL) using a comparison operator like > or =, the result is "unknown".
  - If we want to check whether a value is NULL, SQL provides the special comparison operator **`IS NULL`**
- Arithmetic Operations (*, +, etc):
  - When at least one operand has a NULL value (the other operands can have any value including NULL) then the result is NULL (consequence 0*NULL=NULL !)
  - We cannot use NULL as an operand (e.g., rating < NULL).

NULL Values (contd.)

- 3-valued logic necessary: `true, false, unknown`

- NOT A

A	NOT A
true	false
false	true
unknown	unknown

- A AND B

A \ B	true	false	unknown
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

- A OR B

A \ B	true	false	unknown
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

Query evaluation considering NULL values

- Evaluation in SQL
  - The qualification in the **`WHERE`** clause eliminates rows for which the qualification does not evaluate **`true`** (i.e., rows that evaluate to **`false`** or **`unknown`** are eliminated)
  - SQL defines that rows are duplicates if corresponding columns are either equal or both contain NULL (in contrast to the usual on previous slide where the comparison of the NULLs results in **`unknown`**)
  - All other aggregate operations simply discard NULL values

DB Modifications: insert/delete/update

- Insert values for all attributes in the order attributes were declared or values for only some attributes
  - `INSERT INTO Skaters VALUES (68,'Jacky',10, 10)`
  - `INSERT INTO Skaters (sid,name) VALUES (68, 'Jacky')`
- Insert the result of a query
  - `ActiveSkaters(sid,name)`
  -
    ```
    INSERT INTO ActiveSkaters (
      SELECT Skaters.sid Skaters.name
      FROM Skaters, Participates
      WHERE Skaters.sid =  Participates.sid)
    ```

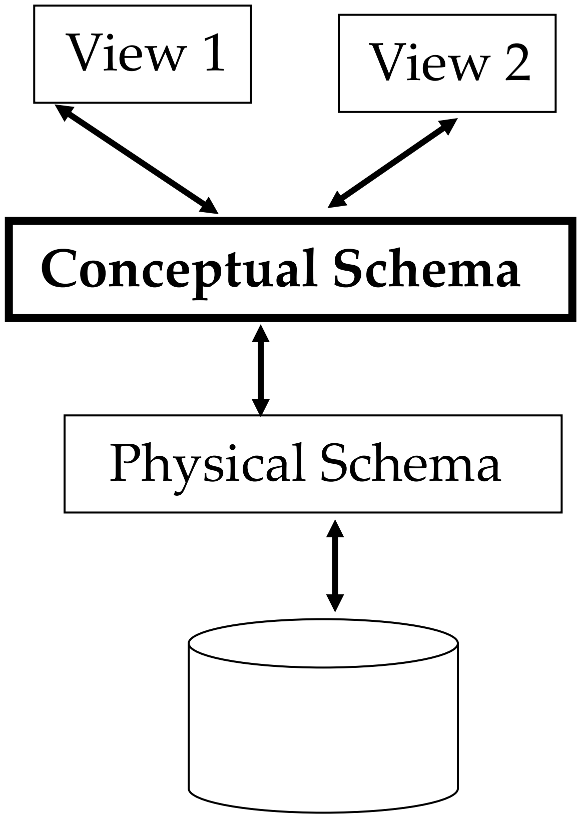
DB Modifications: insert/delete/update

- Delete some or all tuples of a relation
  - `DELETE FROM Competitions WHERE cid = 103`
  - `DELETE FROM Competitions`
- Update some of the attributes of some of the tuples

  -
    ```
    UPDATE Skaters
    SET rating = 10, age = age + 1
    WHERE name = 'debby' OR name = 'lilly'
    ```
- SQL2 semantics: all conditions in a modification statement must be evaluated by the system *BEFORE* any modifications occur.

Levels of Abstraction

- Single **conceptual (logical) schema** defines logical structure
  - Conceptual database design
- **Physical schema** describes the files and indexes used
  - Physical database design
- Different **views** describe how users see the data (also referred to as external schema)
  - generated on demand from the real data
- **Physical data independence**: the conceptual schema protects from changes in the physical structure of data
- **Logical data independence**: external schema protects from changes in conceptual schema of data


View 1
View 2
Conceptual Schema
Physical Schema