

SQL: Basic concepts

- SQL operates with tables, so the first thing to do is create tables.
- Syntax:

```
CREATE TABLE <Name> (<attr1> type, ..., <attrN> type)
```

- For example:

```
CREATE TABLE Movies (title char(20),  
                      director char(10),  
                      actor char(10))  
CREATE TABLE Schedule (theater char(10),  
                        title char(20))
```

Types

- `char(n)` – fixed length string of exactly *n* characters.
Example: 'Polanski'
- `varchar(n)` – variable length string of up to *n* characters.
Example: 'Polanski'. What's the difference? We'll see soon.
Note: `varchar` is actually an abbreviation for `char varying`.
- `bit(n)` – fixed length bit string of exactly *n* bits.
Example: B'0101', X'C1'
- `bit varying(n)` – variable length bit string of up to *n* bits.

Types cont'd

- `int` – signed integer (4 bytes)
- `smallint` – signed integer (2 bytes)
- `real` – real numbers.
- In fact, there is a general float type `float(s)`, and `real` is `float(s)` where s is implementation defined.
- SQL has many more types, such as `date`, `time`, `timestamp`, character sets in different alphabets, etc.

Types cont'd: Dates and Times

- `date` type: keyword `DATE` followed by a date in an appropriate form, e.g. `DATE '2001-12-14'`
- `time` type: keyword `TIME` followed by a string representing time; SQL uses a 24-hour clock. For example, most of you will have left this room by `TIME '18:00:01'`
- `timestamp` type: combines date and time. For example, `TIMESTAMP '2001-12-14 11:28:00'` is 11:28am on December 14, 2001.
- Operations on these types: they can be compared for equality, and for order. If for two dates d_1 and d_2 we have $d_1 < d_2$, then d_1 is earlier than d_2 .

Populating tables

- General syntax:

```
INSERT INTO <name> VALUES (...)
```

- Examples:

```
INSERT INTO Movies VALUES  
('Chinatown', 'Polanski', 'Nicholson')
```

```
INSERT INTO Schedule VALUES ('Odeon', 'Chinatown')
```

- More generally, one can use other queries for insertion:

```
INSERT INTO Name  
(SELECT ... FROM ... WHERE ...)
```

as long as the attributes in the result of the query are the same as those of Name.

Dropping tables

- DROP TABLE Name
removes the table from the database.

Changing tables

- Adding attributes:

```
ALTER TABLE Name ADD COLUMN  
newcolumn type
```

Example:

```
ALTER TABLE Schedule ADD COLUMN  
screen# smallint
```

Dropping columns

- ALTER TABLE Name DROP COLUMN columnname
- Example:

```
ALTER TABLE Schedule DROP COLUMN screen#
```

Default values

- Could be specified for some attributes:

```
CREATE TABLE Name (... <attribute> <type> DEFAULT <value> ...)
```

Example:

```
CREATE TABLE F (A1 INT DEFAULT 0, A2 INT)
```

```
INSERT INTO F VALUES (1,1)
```

```
select * from f
```

```
A1          A2
-----
          1          1
```

Default values cont'd

```
INSERT INTO F (A2) VALUES (3)
```

```
SELECT * FROM f
```

A1	A2
1	1
0	3

Fixed and variable length

```
CREATE TABLE foo1 (AA CHAR(10))
INSERT INTO foo1 VALUES ('xx')
SELECT LENGTH(AA) AS X FROM foo1
```

```
X
-----
      10
```

```
CREATE TABLE foo2 (AA VARCHAR(10))
INSERT INTO foo2 VALUES ('xx')
SELECT LENGTH(AA) AS X FROM foo2
```

```
X
-----
      2
```

SQL and constraints

- Keys are the most common type of constraints
- One should declare them in CREATE TABLE
- Example:

```
CREATE TABLE Employee
  (EmpId int not null primary key,
   FirstName char(20),
   LastName char(20),
   Dept char(10),
   Salary int default 0)
```

- not null means that the value of the attribute must always be present.

Primary keys

- CREATE TABLE specifies that certain constraints must be satisfied
- SQL then checks if each update preserves constraints
- Declare a table:
create table r (a1 int primary key not null, a2 int)
- Insertions:

```
db2 => insert into r values (1,2)
DB20000I The SQL command completed successfully.
db2 => insert into r values (1,3)
```

```
DB21034E The command was processed as an SQL statement because it was
not a valid Command Line Processor command. During SQL processing it
returned: SQL0803N One or more values in the INSERT statement, UPDATE
statement, or foreign key update caused by a DELETE statement are not
valid because they would produce duplicate rows for a table with a
primary key, unique constraint, or unique index. SQLSTATE=23505
```

Another way to declare primary keys

```
CREATE TABLE Employee
(EmpId int not null primary key,
 FirstName char(20),
 LastName char(20),
 Dept char(10),
 Salary int default 0)
```

```
CREATE TABLE Employee
(EmpId int not null,
 FirstName char(20),
 LastName char(20),
 Dept char(10),
 Salary int default 0,
 primary key (EmpId))
```

These are equivalent.

More than one key

- *Primary* in primary keys refers to primary means of accessing a relation.
- What if we have another key, e.g., (FirstName, LastName)
- We cannot declare it as another primary key.
- What does it mean that attributes K form a key for R ?
- It means that for any tuple t of values for K , there exists a *unique* tuple t' in R with $\pi_K(t') = t$.
- Hence we have unique declaration is SQL.

UNIQUE in SQL

- Revised example:

```
CREATE TABLE Employee
  (EmpId int not null,
   FirstName char(20) not null,
   LastName char(20) not null,
   Dept char(10),
   Salary int default 0,
   primary key (EmpId),
   unique (FirstName,LastName))
```

- Unique specifications are verified in the same way as primary key.

```
create table R (A not null, unique (A))
insert into R values 1
```

works fine but the following

```
insert into R values 1 gives an error message.
```

Inclusion constraints: reminder

- *Referential* integrity constraints: they talk about attributes of one relation but refer to values in another one.
- There is an inclusion dependency $R[A_1, \dots, A_n] \subseteq S[B_1, \dots, B_n]$ when

$$\pi_{A_1, \dots, A_n}(R) \subseteq \pi_{B_1, \dots, B_n}(S)$$

- Most often inclusion constraints occur as a part of a *foreign key*
- Foreign key is a conjunction of a key and an ID:

$$R[A_1, \dots, A_n] \subseteq S[B_1, \dots, B_n] \quad \text{and}$$

$$\{B_1, \dots, B_n\} \rightarrow \text{all attributes of } S$$

- Meaning: we find a key for relation S in relation R .

Inclusion dependencies in SQL

```
CREATE TABLE Movies
  (Title char(20), Director char(10), Actor char(10))
```

```
CREATE TABLE Schedule
  (Title char(20) references Movies(Title),
   Theater char(20))
```

Semantics:

$$\text{Schedule[Title]} \subseteq \text{Movies[Title]}$$

Warning: it is in the standard but doesn't seem to be working in the current version of db2.

Foreign keys in SQL

General definition:

```
CREATE TABLE Person
  (FirstName char(20) not null,
   LastName char(20) not null,
   ...
   primary key (FirstName, LastName))
```

Foreign keys in SQL cont'd

```
CREATE TABLE Employee
  (FirstName char(20) not null,
   LastName char(20) not null,
   ....
   foreign key (FirstName, LastName)
     references Person(FirstName, LastName))
```

```
CREATE TABLE Student
  (FName char(20) not null,
   LName char(20) not null,
   ....
   foreign key (FName, LName)
     references Person(FirstName, LastName))
```

Foreign keys in SQL cont'd

In some systems, you can only use a restricted form of this definition:

```
CREATE TABLE Employee
  (FirstName char(20) not null,
   LastName char(20) not null,
   ....
   foreign key (FirstName, LastName)
     references Person)
```

In general:

```
CREATE TABLE T1 (...
  ... foreign key <attr1,...,attrN>
     references T2)
```

In T2, <attr1,...,attrN> must be present and form a primary key.

Duplicates

```
SELECT * FROM T1
```

```
A1  A2
----  ----
1    2
2    1
1    1
2    2
```

```
SELECT A1 FROM T1
```

```
A1
--
1
2
1
2
```

Duplicates cont'd

- SELECT is not exactly the projection of relational algebra.
- Projection returns the set $\{1, 2\}$
- SELECT keeps duplicates.
- How to remove duplicates? Use SELECT DISTINCT

```
SELECT DISTINCT A1 FROM T1
```

```
A1
--
1
2
```

Dealing with duplicates

- So far, in relational algebra and calculus, we operated with sets. SQL, on the other hand, deals with bags, that is, sets with duplicates.
- This requires small changes to the operations of the relational algebra.
- Projection π no longer removes duplicates:

$$\pi_A \left(\begin{array}{c|c} A & B \\ \hline a_1 & b_1 \\ a_2 & b_2 \\ a_1 & b_2 \end{array} \right) = \{a_1, a_2, a_1\}$$

Notice that a_1 occurs twice.

- There is a special duplicate elimination operation:

$$\text{duplicate_elimination}(\{a_1, a_2, a_1\}) = \{a_1, a_2\}$$

Dealing with duplicates: union

- The union operation just puts two bags together:

$$\begin{aligned} S &= \{1, 1, 2, 2, 3, 3\} \\ T &= \{1, 2, 2, 2, 3\} \\ S \cup T &= \{1, 1, 1, 2, 2, 2, 2, 3, 3, 3\} \end{aligned}$$

That is, if a occurs k times in S , and m times in T , then it occurs $k + m$ times in $S \cup T$.

- This is, however, *not* the UNION operation of SQL. SQL's UNION does eliminate duplicates.
- If you want to keep duplicates, use UNION ALL:

```
SELECT * FROM S
UNION ALL
SELECT * FROM T
```

Dealing with duplicates: intersection

- The intersection operation keeps the minimum number of occurrences of an element:

$$\begin{aligned}S &= \{1, 1, 2, 2, 3, 3\} \\T &= \{1, 2, 2, 2, 3\} \\S \cap T &= \{1, 2, 2, 3\}\end{aligned}$$

That is, if a occurs k times in S , and m times in T , then it occurs $\min(k, m)$ times in $S \cap T$.

- This is, again, *not* the INTERSECT operation of SQL. SQL's INTERSECT, just as UNION, eliminates duplicates.
- If you want to keep duplicates, use INTERSECT ALL:

```
SELECT * FROM S
  INTERSECT ALL
SELECT * FROM T
```

Dealing with duplicates: difference

- The difference operation works as follows:

$$\begin{aligned}S &= \{1, 1, 2, 2, 3, 3\} \\T &= \{1, 2, 2, 2, 3\} \\S - T &= \{1, 3\}\end{aligned}$$

That is, if a occurs k times in S , and m times in T , then it occurs $k - m$ times in $S - T$, if $k > m$, and does not occur at all in $S - T$ if $k \leq m$.

- This is, again, *not* the EXCEPT operation of SQL. SQL's EXCEPT, just as UNION and INTERSECT, eliminates duplicates.
- If you want to keep duplicates, use EXCEPT ALL:

```
SELECT * FROM S
  EXCEPT ALL
SELECT * FROM T
```

SQL is NOT a programming language

- Calculate $2 + 2$ in SQL
- Step 1: there must be a table to operate with:

```
create table foo (a int)
```

- $2 + 2$ itself must go into selection. We also have to give it a name (attribute).
- Try:

```
db2 => select 2+2 as X from foo
```

```
X
-----
0 record(s) selected.
```

SQL is NOT a programming language cont'd

- Problem: there were no tuples in `foo`.
- Let's put in some:

```
insert into foo values 1
insert into foo values 5
```

```
select 2+2 as X from foo
```

```
X
-----
4
4

2 record(s) selected.
```

SQL is NOT a programming language cont'd

- It is also important to eliminate duplicates.
- So finally:

```
db2 => select distinct 2+2 as X from foo
```

```
X  
-----  
4
```

1 record(s) selected.

Empty set traps

- Assume there are three relations, S, T, R , with the same attribute A .
- Query: compute $Q = R \cap (S \cup T)$.
- A seemingly correct way to write it:

```
SELECT R.A  
FROM R, S, T  
WHERE R.A=S.A OR R.A=T.A
```

- Let $R = S = \{1\}, T = \emptyset$. Then $Q = \{1\}$, but the SQL query produces the empty table.
- Why?

More on the WHERE clause

- Once we have types such as strings, numbers, we have type-specific operations, and hence type-specific selection conditions

- ```
create table finance (title char(20),
 budget int,
 gross int)
```

```
insert into finance values ('Shining', 19, 100)
insert into finance values ('Star wars', 11, 513)
insert into finance values ('Wild wild west', 170, 80)
```

## More on the WHERE clause

- Find movies that lost money:

```
select title
from finance
where gross < budget
```

- Find movies that made at least 10 times as much as they cost:

```
select title
from finance
where gross > 10 * budget
```

- Find profit each movie made:

```
select title, gross - budget as profit
from finance
where gross - budget > 0
```

## More on the WHERE clause cont'd

- Is Kubrick spelled with a “k” or “ck” at the end?
- No need to remember.

```
SELECT Title, Director
FROM Movies
WHERE director LIKE 'Kubr%'
```

- Is Polanski spelled with a “y” or with an “i”?

```
SELECT Title, Director
FROM Movies
WHERE director LIKE 'Polansk_'
```

## LIKE comparisons

- attribute LIKE pattern
- Patterns are built from:
  - letters
  - \_ – stands for any letter
  - % – stands for any substring, including empty
- Examples:
  - address LIKE '%Toronto%'
  - pattern '\_a\_b\_' matches cacbc, aabba, etc
  - pattern '%a%b\_' matches ccaccbc, aaaabcbcbdbd, aba, etc

## LIKE comparisons: telling the truth

- ```
SELECT Title, Director
FROM Movies
WHERE director LIKE 'Polansk_'
```

 returns the empty set
- Because sometimes $x = y$ is true, but x LIKE y is false!
- The reason: trailing spaces
- `'Polanski '` = `'Polanski '` is true, but `'Polanski '` LIKE `'Polanski '` is false.
- Director was defined as `char(10)`, so `'Polanski '` is really `'Polanski '` and thus doesn't match `'Polansk_'`.

LIKE and trailing spaces

- Solution 1: use `varchar` (or `char varying`) declarations.
- Solution 2: use `'Polansk%'` as a pattern
- Solution 3: use the `TRIM` function:

```
SELECT Title, Director
FROM Movies
WHERE TRIM(TRAILING FROM Director) LIKE 'Polansk_'
```
- `TRIM TRAILING` eliminates trailing spaces (`LEADING` eliminates leading spaces, `BOTH` eliminates both leading and trailing spaces)
- Warning: db2 doesn't seem to like it ...

Adding attributes ... towards aggregate queries

```
ALTER TABLE Movies ADD COLUMN Length int DEFAULT 0
```

```
UPDATE Movies  
SET Length = 131  
WHERE title='Chinatown'
```

```
UPDATE Movies  
SET Length = 146  
WHERE title='Shining'
```

adds attribute Length, and puts in values of that attribute.

Adding attributes cont'd

```
ALTER TABLE Schedule ADD COLUMN Time int DEFAULT 0
```

```
UPDATE Schedule  
SET Time = 18  
WHERE Theater='Le Champo' AND Title='Chinatown'
```

```
INSERT INTO Movies VALUES ('Le Champo', 'Chinatown', 21)
```

adds attribute Time and puts in values.

Note that there could be more than one showing of a movie, hence we use both UPDATE and INSERT.

More examples that use arithmetic

Query: Suppose I want to see a movie by Lucas. I can't get to a theater before 8pm (20:00), and I must be out by 11pm (23:00). I want to see: theater, and the exact time I'll be out, if my conditions are satisfied.

```
SELECT S.Theater, S.Time + (M.Length/60.0) AS Outtime
FROM Schedule S, Movies M
WHERE M.Title=S.Title
      AND M.Director='Lucas'
      AND S.Time >= 20
      AND S.Time + (M.Length/60.0) < 23
```

Simple aggregate queries

Count the number of tuples in Movies

```
SELECT COUNT(*)
FROM Movies
```

Add up all movie lengths

```
SELECT SUM(Length)
FROM Movies
```

Duplicates and aggregation

Find the number of directors.

Naive approach:

```
SELECT COUNT(Director)
FROM Movies
```

returns the number of tuples in Movies.
Because: SELECT does not remove duplicates.

Correct query:

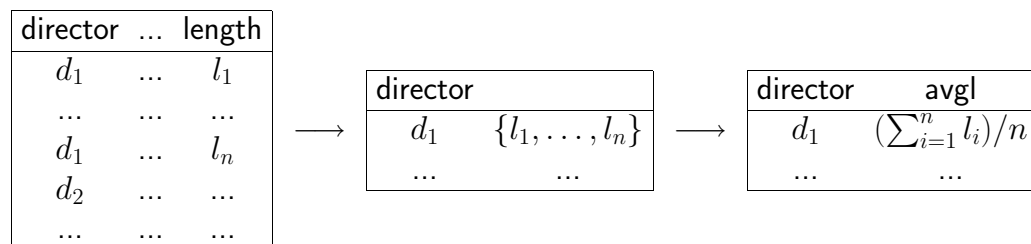
```
SELECT COUNT(DISTINCT Director)
FROM Movies
```

Aggregation and grouping

For each director, return the average running time of his/her movies.

```
SELECT Director, AVG(Length) AS Avg1
FROM Movies
GROUP BY Director
```

How does grouping work?



Aggregation and duplicates

Foo1	A1	A2	A3
	a	1	5
	a	1	2
	a	2	2
	a	2	3

```
SELECT A1, AVG(A3)
FROM Foo1
GROUP BY A1
```

A1	2
a	?

Aggregation and duplicates cont'd

One approach: take all the values of A3 and compute their average:

$$\frac{5 + 2 + 2 + 3}{4} = 3$$

Another approach: only attributes A1 and A3 are relevant for the query.

$$\pi_{A1,A3} \left(\begin{array}{c|cc} A1 & A2 & A3 \\ \hline a & 1 & 5 \\ a & 1 & 2 \\ a & 2 & 2 \\ a & 2 & 3 \end{array} \right) = \left(\begin{array}{c|c} A1 & A3 \\ \hline a & 5 \\ a & 2 \\ a & 3 \end{array} \right)$$

$$\text{Average} = \frac{5 + 2 + 3}{3} = \frac{10}{3}$$

Aggregation and duplicates cont'd

- SQL approach: always keep duplicates.
- The right answer is thus 3.
- However, one has to be careful:

```
SELECT AVG(A2) FROM Foo1
```

returns 1
- The reason: rounding
- Solution: cast as real numbers:

```
SELECT AVG(CAST (A2 AS REAL)) FROM Foo1
```

returns 1.5
- Syntax for CAST

```
CAST (<attribute> AS <type>)
```

More on duplicates

- What if we want to eliminate duplicates before computing aggregates?
- Use DISTINCT
- ```
SELECT AVG(DISTINCT A3) FROM Foo1
```

produces 3, due to rounding, but
- ```
SELECT AVG(DISTINCT CAST (A3 AS REAL)) FROM Foo1
```

produces, as expected, 3.3333...

More on rounding

- A dirty trick to cast integers as reals:

```
SELECT AVG(A3 + 0.0) FROM Foo1
```

Other aggregates

- MIN – minimum value of a column
- MAX – maximum value of a column
- SUM – adds up all elements in a column
- COUNT – counts the the number of values in a column

- MIN and MAX produce the same result regardless of duplicates
- SUM adds up all elements in a given column;
SUM DISTINCT adds up all distinct elements in a given column
- COUNT counts elements in a given column;
COUNT DISTINCT counts distinct elements in a given column

SUM, COUNT, and duplicates

- SELECT COUNT(A3) FROM Foo1
produces 4
- SELECT COUNT(DISTINCT A3) FROM Foo1
produces 3
- SELECT SUM(A3) FROM Foo1
produces 12
- SELECT SUM(DISTINCT A3) FROM Foo1
produces 10
- SELECT MIN(A3) FROM Foo1 and
SELECT MIN(DISTINCT A3) FROM Foo1
give the same result.
- The same holds for MAX.

Selection based on aggregation results

- Find directors and average length of their movies, provided they made at least one movie that is longer than 2 hours.
- Idea: calculate two aggregates: $\text{AVG}(\text{Length})$ and $\text{MAX}(\text{Length})$ and only choose directors for whom $\text{MAX}(\text{Length}) > 120$.
- SQL has a special syntax for it: HAVING.
- ```
SELECT Director, AVG(Length+0.0)
FROM Movies
GROUP BY Director
HAVING MAX(Length) > 120
```

## Aggregation and join

- Aggregate queries may use more than one relation.
- For each theater showing at least one movie that is longer than 2 hours, find the average length of movies playing there.
- ```
SELECT S.Theater, AVG(CAST (M.Length AS REAL))
FROM Schedule S, Movies M
WHERE S.Title=M.Title
GROUP BY S.Theater
HAVING MAX(M.Length) > 120
```
- What it says: produce the join $\text{Movies} \bowtie \text{Schedule}$, and over that join run the aggregate query that computes the average.

Aggregation, join and duplicates

- One could have unexpected results due to duplicates.
- Two tables:

R	A1	A2	S	A1	A3
	'a'	1		'a'	5
	'b'	2		'a'	7
				'b'	3

- Query:

```
SELECT R.A1, SUM(R.A2)
FROM R, S
WHERE R.A1=S.A1 AND R.A1='a'
GROUP BY R.A1
HAVING MIN(S.A3) > 0
```

- What is the result?

Aggregation, join and duplicates cont'd

- It appears that table S is irrelevant, and the result should be the same as that of:

```
SELECT A1, SUM(A2)
FROM R
WHERE A1='a'
GROUP BY A1
```

- This returns ('a', 1).
- However, the original query returns ('a', 2).

Aggregation, join and duplicates cont'd

- Why is this happening?
- Because the query first constructs the join $R \bowtie S$:

$R \bowtie S$	A1	A2	A3
	'a'	1	5
	'a'	1	7
	'b'	2	3

- and then runs the aggregate part against it, that is:

```
SELECT A1, SUM(A2)
FROM R  $\bowtie$  S
WHERE A1='a'
GROUP BY A1
HAVING MIN(A3) > 0
```

- Of course this returns ('a',2)

Aggregation, join and duplicates cont'd

- One has to be careful about duplicates even if it appears that there aren't any.
- The correct way to write this query is with DISTINCT:

```
SELECT R.A1, SUM(DISTINCT R.A2)
FROM R, S
WHERE R.A1=S.A1 AND R.A1='a'
GROUP BY R.A1
HAVING MIN(S.A3) > 0
```

- and it returns ('a',1), as expected.

Aggregates in WHERE

- Results of aggregates can be used for comparisons not only in the HAVING clause.
- Find movies that run longer than the longest currently playing movie:

```
SELECT M.Title
FROM Movies M
WHERE M.length > (SELECT MAX(M1.length)
                  FROM Movies M1, Schedule S
                  WHERE M1.title=S.title)
```

Aggregates in WHERE cont'd

- Be careful **not** to write:

```
SELECT M.Title
FROM Movies M
WHERE M.length > MAX(SELECT M1.length
                    FROM Movies M1, Schedule S
                    WHERE M1.title=S.title)
```

which is incorrect.

- Instead, you can write in SQL:

```
SELECT M.Title
FROM Movies M
WHERE M.length > ALL(SELECT M1.length
                    FROM Movies M1, Schedule S
                    WHERE M1.title=S.title)
```

Aggregates in WHERE cont'd

- A similar query:
- Find movies that are shorter than some currently playing movie:

```
SELECT M.Title
FROM Movies M
WHERE M.length < (SELECT MAX(M1.length)
                  FROM Movies M1, Schedule S
                  WHERE M1.title=S.title)
```

or

```
SELECT M.Title
FROM Movies M
WHERE M.length < ANY(SELECT M1.length
                    FROM Movies M1, Schedule S
                    WHERE M1.title=S.title)
```

- Note that it's ANY but not ALL in this case.

ALL vs ANY

- $\langle \text{value} \rangle \langle \text{condition} \rangle \text{ ALL } (\langle \text{query} \rangle)$
is true if either:
 - $\langle \text{query} \rangle$ evaluates to the empty set, or
 - for every $\langle \text{value1} \rangle$ in the result of $\langle \text{query} \rangle$,
 $\langle \text{value} \rangle \langle \text{condition} \rangle \langle \text{value1} \rangle$ is true.
- For example,
 - $5 > \text{ALL}(\emptyset)$ is true;
 - $5 > \text{ALL}(\{1, 2, 3\})$ is true;
 - $5 > \text{ALL}(\{1, 2, 3, 4, 5, 6\})$ is false.

ALL vs ANY cont'd

- `<value> <condition> ANY (<query>)`
is true if for some `<value1>` in the result of `<query>`,
`<value> <condition> <value1>` is true.
- For example,
 - 5 < ANY(\emptyset) is false;
 - 5 < ANY({1, 2, 3, 4}) is false;
 - 5 < ANY({1, 2, 3, 4, 5, 6}) is true.

Aggregates in WHERE cont'd

- Nor all comparisons with aggregate results can be replaced by ANY and ALL comparisons.
- Is there a movie whose length is at least 10% of the total lengths of all other movies combined?

```
SELECT M.Title
FROM Movies M
WHERE M.length >= 0.1 * (SELECT SUM(M1.length)
                        FROM Movies M1
                        WHERE M1.title <> M.title)
```

Joins in queries

- When we explained the semantics of aggregate queries, we used the following “query”:

```
SELECT A1, SUM(A2)
FROM R ⋈ S
WHERE A1='a'
GROUP BY A1
HAVING MIN(A3) > 0
```

- This isn't an SQL query – it uses \bowtie from relational algebra.
- But we can write this in SQL:

- ```
SELECT A1, SUM(A2)
FROM R NATURAL JOIN S
WHERE A1='a'
GROUP BY A1
HAVING MIN(A3) > 0
```

## Joins in queries cont'd

- Not all systems understand NATURAL JOIN, certainly not db2.
- There is a more general syntax for joins:

```
SELECT A1, SUM(A2)
FROM R JOIN S ON R.A1=S.A1
WHERE A1='a'
GROUP BY A1
HAVING MIN(A3) > 0
```

- $R \text{ JOIN } S \text{ ON } c$  computes

$$\sigma_c(R \times S)$$

- Condition  $c$  could be more complicated than simple attribute equality, e.g.  $R.A2 > S.A3 - 4$ .

## Theta joins

- Expressions like  $R \text{ JOIN } S \text{ ON } c$  are usually called *theta-joins* and are often included in relational algebra:

$$R \bowtie_{\theta} S$$

- This is not a new operation of the relational algebra but simply an abbreviation for  $\sigma_{\theta}(R \times S)$ .
- Reason for the name: traditionally, conditions were denoted by  $\theta$ .

## Joins in queries cont'd

- Caveat: it is no longer clear which relation a given attribute comes from:

```
SELECT A1, SUM(A2)
FROM R JOIN S ON R.A1=S.A1
GROUP BY R.A1
```

- SQL complains: A reference to column "A1" is ambiguous.
- db2 => `select * from r join s on r.a1=s.a1`

| A1 | A2 | A1  | A3 |
|----|----|-----|----|
| a  |    | 1 a | 5  |
| a  |    | 1 a | 7  |
| b  |    | 2 b | 3  |

## Joins in queries cont'd

- To use aggregation, one has to specify which relation attributes come from:

```
SELECT S.Theater, MAX(M.Length)
FROM Movies M JOIN Schedule S ON M.Title=S.Title
GROUP BY S.Theater
```

finds theaters and the lengths of the longest movies playing there.

- Note aliasing used inside the JOIN expression.
- Joins can also be given different names:

```
SELECT JT.theater, MAX(JT.Length)
FROM (Movies NATURAL JOIN Schedule) AS JT
```

## Joins in queries cont'd

- Join expressions could be quite complicated:

```
((R JOIN S ON <cond1>) AS Table1
 JOIN
 (U JOIN V ON <cond2>) AS Table2
 ON <cond3>)
```

- One has to be careful with referencing tables in conditions, e.g.:
  - <cond1> can refer to R, S, but not U, V, Table1, Table2
  - <cond2> can refer to U, V, but not R, S, Table1, Table2
  - <cond3> can refer to Table1, Table2 but not R, S, U, V

## More on subqueries

- So far we saw subqueries only in the WHERE clause, and in a limited way, in the FROM clause.
- But they can occur anywhere!
- Example: avoiding GROUP BY.

```
SELECT DISTINCT S.theater,
 (SELECT MAX(M.Length)
 FROM Movies M
 WHERE M.Title=S.Title)
FROM Schedule S
```

## More on subqueries cont'd

- Avoiding HAVING: subqueries in WHERE.

```
SELECT DISTINCT S.theater,
 (SELECT MAX(M.Length)
 FROM Movies M
 WHERE M.Title=S.Title)
FROM Schedule S
WHERE (SELECT COUNT(DISTINCT Title)
 FROM Movies M1
 WHERE M1.title IN (SELECT S1.title
 FROM Schedule S1
 WHERE S1.theater=S.theater)) > 5
```

restricts the previous query to theaters showing 6 or more movies.

- In general, the new standard is very liberal about where one can use a subquery, but not all systems fully comply yet.

## A useful feature: ordering the output

```
db2 => SELECT * FROM S
```

```
A1 A3
-- -----
a 5
a 7
b 3
```

```
db2 => SELECT * FROM S ORDER BY A3
```

```
A1 A3
-- -----
b 3
a 5
a 7
```

## A useful feature: ordering the output cont'd

- Decreasing order:

```
db2 => SELECT * FROM S ORDER BY A3 DESC
```

```
A1 A3
-- -----
a 7
a 5
b 3
```

- Ordering involving multiple attributes:

```
db2 => SELECT * FROM S ORDER BY A1, A3
```

```
A1 A3
-- -----
a 5
a 7
b 3
```

## Intermediate results

- There is a way to save intermediate results, for future reference
- Such intermediate results are called *views*
- Usually it is done when the result of a certain query is needed often
- Syntax: CREATE VIEW <name> (<attributes>) AS <query>
- Example: suppose we need theaters, directors whose movies are playing there, and lengths of those movies:

```
CREATE VIEW TDL (th, dir, len) AS
 SELECT S.theater, M.director, M.length
 FROM Movies M, Schedule S
 WHERE S.title=M.title
```

## Using views

- Once a view is created, it can be used in queries.
- Find theaters showing long (> 2 hours) movies by a director whose name starts with "K"
- SELECT th  
FROM TDL  
WHERE len > 120 and dir LIKE 'K%'
- Advantage: if the view is already created, one no longer has to perform a join.
- Thus views are useful if many queries are asked against them.

## Using views cont'd

- Views are also useful for making queries more readable, e.g., by creating intermediate values.
- However, it is not a good idea to use views for those purposes (e.g., one would need to say `DROP VIEW` afterwards, when the view is no longer needed).
- Instead, one can use keyword `WITH`:

```
WITH TDL (th, dir, len) AS
 (SELECT S.theater, M.director, M.length
 FROM Movies M, Schedule S
 WHERE S.title=M.title)
SELECT th
FROM TDL
WHERE len > 120 and dir LIKE 'K%'
```

## Database modifications

- We have seen how to insert tuples in relations:

```
INSERT INTO Table VALUES (...)
```

- One can also insert results of queries, as long as attributes match.
- Example: We want to ensure that every movie in table `Schedule` is listed in table `Movies`. This is done by

```
INSERT INTO Movies(title)
 SELECT DISTINCT S.Title
 FROM Schedule S
 WHERE S.Title NOT IN (SELECT title
 FROM Movies)
```

- What are the values of `director` and `actor` attributes when a new title is inserted? Answer – default values (most often nulls). We'll see them later.

## Database modification: deletions

- Suppose we want to delete movies which are not currently playing in theaters, unless they are directed by Kubrick:

```
DELETE FROM Movies
 WHERE title NOT IN (Select title FROM Schedule) AND
 director <> 'Kubrick'
```

- General form:

```
DELETE FROM <relation name>
 WHERE <condition>
```

- Conditions apply to individual tuples; all tuples satisfying the condition are deleted.

## Database modification: updates

- Suppose we have a table Personnel with two of its attributes being name and gender.
- Now, we want to replace, in table Movies, each name X of a male director by 'Mr. X':

```
UPDATE Movies
SET director = 'Mr. ' || director
WHERE director in
 (SELECT name FROM Personnel WHERE gender='male')
```

- Here || is the SQL notation for string concatenation.
- General form of updates:

```
UPDATE <table> SET <value-assignments> WHERE <conditions>
```

- Tables are updated one tuple at a time.

## Referential integrity and updates

- Updates can create problems with keys and foreign keys.
- We have seen that insertions can violate key constraints.
- The situation is more complex with foreign keys.

```
create table R (a int not null, b int, primary key (a))

create table S (a int not null, foreign key (a) references r)

insert into R values (1,1)

insert into S values 1
```

## Referential integrity and updates cont'd

So far so good, but inserting 2 into S results in an error:

```
db2 => insert into s values 2
```

```
DB21034E The command was processed as an SQL statement because it was
not a valid Command Line Processor command. During SQL processing it
returned: SQL0530N The insert or update value of the FOREIGN KEY
"LIBKIN.S.SQL010129175143860" is not equal to any value of the parent
key of the parent table. SQLSTATE=23503
```

## Referential integrity and updates cont'd

- More serious problem: deletion
- Tables: R(A,B), A primary key; S(A,C)
- Suppose S.A is a foreign key for R.A
- S has 

| A | C |
|---|---|
| 1 | 2 |
| 2 | 2 |

, R has 

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |
- We now delete (1,2) from R, what happens then?
- Possibilities:
  - 1) reject the deletion operation
  - 2) propagate it to S and delete (1,2) from S
  - 3) "we don't know approach": keep the tuple, but put no value for the A attribute there.

## Referential integrity and updates cont'd

- All three approaches are supported in SQL
- ```
create table R1 (a int not null primary key, b int)
create table S1 (a int, c int,
  foreign key (a) references r1)
```
- and insert (1,2) and (2,3) in R1, and (1,2) and (2,2) in S1
- delete from r1 where a=1 and b=2
- results in an error due to the foreign key constraint

Referential integrity and updates cont'd

- All three approaches are supported in SQL
- create table R1 (a int not null primary key, b int)
create table S2 (a int, c int,
foreign key (a) references r1 on delete cascade)
create table S3 (a int, c int,
foreign key (a) references r1 on delete set null)
- insert (1,2) and (2,3) in R1, and (1,2) and (2,2) in S2 and S3
- delete from r1 where a=1 and b=2
- What do we get?

Referential integrity and updates cont'd

- For on delete cascade
db2 => select * from s2
- | A | C |
|---|---|
| 2 | 2 |
- For on delete set null
db2 => select * from s3

A	C
-	2
2	2