

Part 4: SQL II

References:

- Elmasri/Navathe: Fundamentals of Database Systems, 3rd Edition, 1999. Chap. 8, "SQL — The Relational Database Standard" (In this part, we treat only Section 8.3.)
- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Ed., McGraw-Hill, 1999. Chapter 4: "SQL".
- Kemper/Eickler: Datenbanksysteme (in German), 4th Ed., Oldenbourg, 1997. Chapter 4: Relationale Anfragesprachen (Relational Query Languages).
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Date/Darwen: A Guide to the SQL Standard, Fourth Edition, Addison-Wesley, 1997.
- van der Lans: SQL, Der ISO-Standard (in German), Hanser, 1990.
- Sunderraman: Oracle Programming, A Primer. Addison-Wesley, 1999.
- Oracle8 SQL Reference, Oracle Corporation, 1997, Part No. A58225-01.
- Chamberlin: A Complete Guide to DB2 Universal Database. Morgan Kaufmann, 1998.
- Microsoft SQL Server Books Online: Accessing and Changing Data.

Objectives

After completing this chapter, you should be able to:

- write advanced queries in SQL including aggregations, subqueries, and UNION.
- enumerate and explain the clauses of an SQL query.

SELECT, FROM, WHERE, GROUP BY, HAVING, . . . , ORDER BY

- explain joins in SQL-92.
- evaluate the correctness of a given query.
- evaluate the portability of certain constructs.

Overview

1. Aggregations I: Aggregation Functions
2. Aggregations II: GROUP BY, HAVING
3. Subqueries
4. UNION, ORDER BY
5. SQL-92 Joins, Outer Join in Oracle

Aggregations (1)

- Aggregation functions are functions from a set or multiset to a single value (usually a number).

$$\text{E.g.: } \min\{41, 57, 19, 23, 27\} = 19$$

- Aggregation functions aggregate or summarize an entire set of values to a single value.

Aggregation functions are also called “group functions” or “column functions”. They take not a single value as input, but an entire column (a set). The column does not have to be a column of a stored table, it can also be constructed by means of a query (see below).

- Typical aggregation functions are: number (count), sum, average, minimum and maximum.

Aggregations (2)

- Aggregation functions can be used for statistical evaluations.
- Some aggregation functions are sensitive to duplicates (e.g. sum), others are not (e.g. minimum).

E.g. the sum of all items of an invoice. If two items cost the same amount, nevertheless both must be added.

- In SQL, one can request duplicate elimination (input is a set) or not (input is a multiset).

A multiset is a set where each element has a multiplicity, e.g. an element can be contained in a multiset two times. In contrast to a list, there is still no specific order. Also the name “bag” is used.

Aggregations (3)

- SQL-86/92 has the five aggregation functions
COUNT, SUM, AVG, MAX, MIN.

Additional aggregation functions in certain systems:

Oracle and DB2: `VARIANCE`, `STDDEV` (DB2 also: `COUNT_BIG`).

SQL Server: `VAR`, `VARP`, `STDEV`, `STDEVP`.

- Theoretically, any commutative and associative binary operator with a neutral element can be extended to work on sets.

E.g. *sum* is the set-version of $+$.

The conditions are necessary since a set has no order.

However, SQL has a fixed set of aggregation functions.

E.g. SQL has no *prod* (set-version of $*$).

Example Database (1)

STUDENTS		
<u>SSN</u>	FIRST	LAST
123-45-6789	John	Smith
111-22-3333	Ann	Miller
543-76-9821	David	Meyer
900-50-3000	Mary	Jones

ENROLLMENTS	
<u>SSN</u>	<u>CRN</u>
123-45-6789	41590
111-22-3333	41590
111-22-3333	31864
543-76-9821	22332
543-76-9821	31864

Example Database (2)

COURSES				
<u>CRN</u>	TOPIC	TITLE	INSTRUCTOR	SEATS
22332	2710	DB Management	Flynn, R.	50
24271	2610	Data Structures	Flynn, R.	50
31864	2550	Client-Server	Spring, M.	30
41590	2711	DB ... Design	Brass, S.	70
37329	2711	DB ... Design	Brass, S.	30
25688	2770	Document Proc.		35

Simple Aggregations (1)

- What is the largest number of seats in a course?

```
SELECT MAX(SEATS)
FROM COURSES
```

MAX(SEATS)
70

- How many students are enrolled for the course with CRN 41590?

```
SELECT COUNT(*)
FROM ENROLLMENTS
WHERE CRN = 41590
```

COUNT(*)
2

Simple Aggregations (2)

- How many students are currently taking courses?

```
SELECT COUNT(DISTINCT SSN)
FROM ENROLLMENTS
```

COUNT(DISTINCT SSN)
3

- What is the average number of seats in a course?

```
SELECT AVG(SEATS) "Average Seats"
FROM COURSES
```

Average Seats
44.2

Simple Aggregations (3)

- The whole power of SQL can be used to compute the set of values which is aggregated. (E.g. joins.)
- How many students are enrolled in courses held by Brass?

```
SELECT COUNT(DISTINCT SSN)
FROM ENROLLMENTS E, COURSES C
WHERE E.CRN = C.CRN
AND C.INSTRUCTOR LIKE 'Brass%'
```

Simple Aggregations (4)

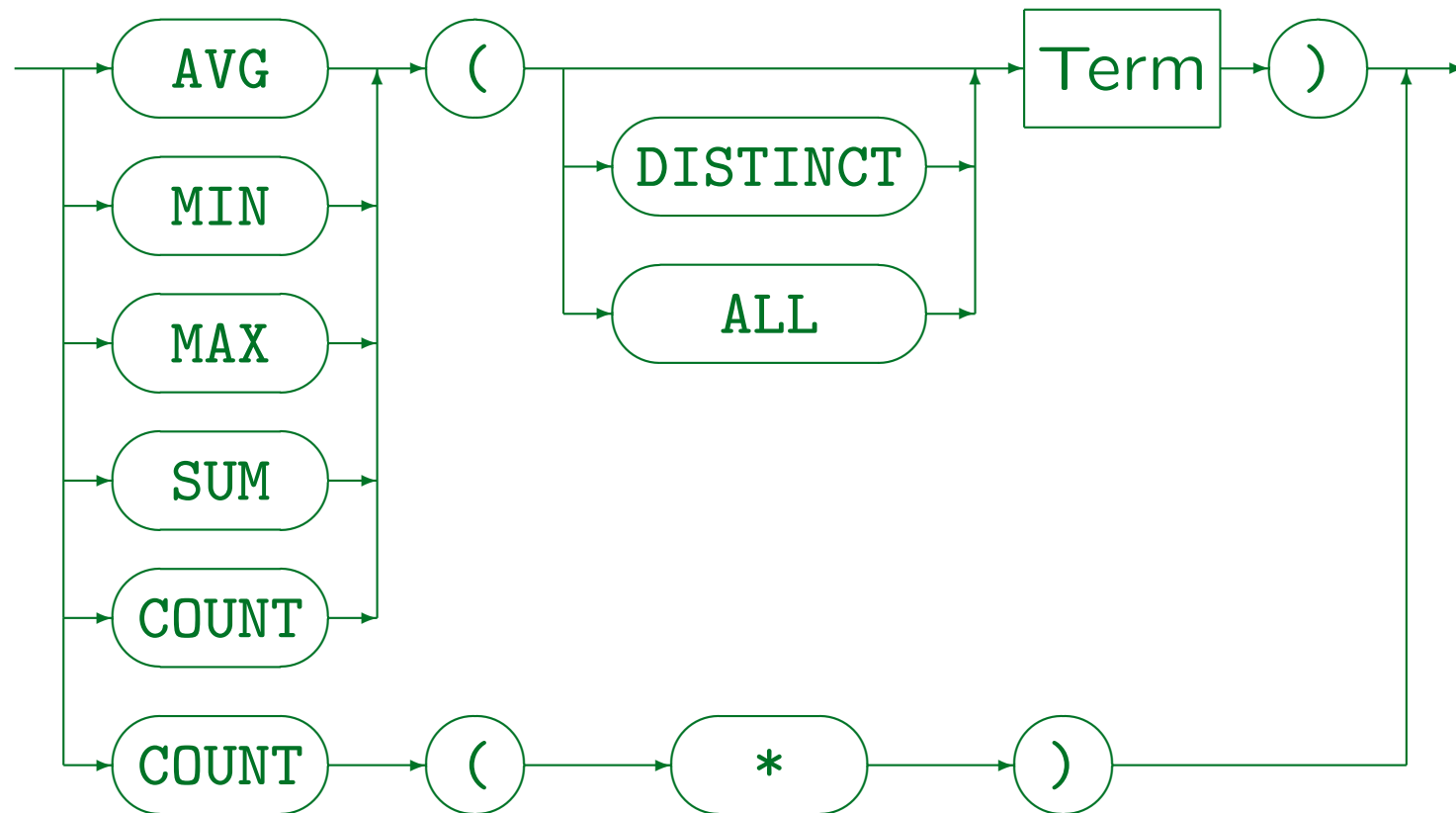
- It is possible to compute more than one aggregation in the `SELECT` list, e.g.: What is the minimum and maximum number of seats?

```
SELECT MIN(SEATS), MAX(SEATS)
FROM   COURSES C
```

- The aggregations can refer to different columns:

```
SELECT COUNT(DISTINCT TOPIC), SUM(SEATS)
FROM   COURSES C
```

Syntax: Aggregation Terms



Evaluation (1)

- First, the FROM-clause is evaluated.

Theoretically, all possible tuple combinations of the source tables are constructed (cartesian product, nested loops).

- Second, the WHERE-clause is evaluated.

Only those tuple combinations that satisfy the condition are further considered (selection, filter, if). Of course, in real systems the first and second step may be combined to allow a more efficient evaluation.

- Third, the SELECT-clause is evaluated. The evaluation is completely different depending on whether the SELECT-list contains aggregation terms or not.

Evaluation (2)

- Usually, the values of the terms/expressions in the `SELECT`-list are printed for every tuple combination that has passed step 2.
- But when the `SELECT`-list contains an aggregation term, only a single output row is computed by applying the aggregation operators.

Instead of printing the values of columns as in the standard case, the values are added to a set/multiset that is the input to the aggregation function. If the `SELECT`-list contains multiple aggregations, multiple such sets must be managed. However, if no `DISTINCT` is used, it is possible to incrementally compute the aggregated values without explicitly storing a temporary set of values (see next slide).

Evaluation (3)

- E.g. consider the query:

```
SELECT SUM(SEATS), COUNT(*)  
FROM   COURSES C  
WHERE  TOPIC = 2711
```

- This is evaluated as:

```
out1 = 0; out2 = 0;  
foreach row C in COURSES do  
    if C.TOPIC = 2711 then begin  
        out1 = out1 + C.SEATS;  
        out2 = out2 + 1;  
    end;  
print out1, out2;
```

Syntax / Restrictions (1)

- The arguments of **SUM** and **AVG** must be numeric. **COUNT**, **MIN**, and **MAX** accept any datatype.
- **Aggregations cannot be nested**, e.g. the following is illegal:

AVG(COUNT(*)) **Wrong!**

After the **COUNT** only a single value remains. Thus, applying another aggregation makes no sense.

There are cases where aggregations are first applied to groups of rows, and then the result is input to another aggregation. E.g. what is the average number of students per course? This is done by using **GROUP BY** and subqueries/views (see below).

Syntax / Restrictions (2)

- Aggregations cannot be used in the **WHERE**-clause.

The **WHERE**-condition is evaluated before aggregations are computed (it determines which tuples enter the aggregation). Conditions with aggregations can be specified under **HAVING** (see below).

WHERE COUNT(*) > 1 Wrong!

- If an aggregation function is used, no normal attributes can appear in the **SELECT**-list.

Only a single output tuple is produced, and an attribute outside aggregations would not have a unique output value. But see **GROUP BY**.

SELECT CRN, COUNT(SSN) Wrong!
FROM ENROLLMENT

Syntax / Restrictions (3)

- Every aggregation operator needs an argument (which specifies input values).

```
SELECT *  
FROM COURSES  
WHERE SEATS = MIN Wrong! Wrong!
```

Aggregations are also not allowed under WHERE.

- A subquery is required to find the course with minimal number of seats (see below).

Syntax / Further Possibilities

- An aggregation term like SUM(SEATS) is by itself a term (an expression), so e.g. the following is legal:

```
SELECT SUM(SEATS) / COUNT(SEATS)
FROM COURSES
```

E.g.: Enrolled students plus three guests:

```
SELECT COUNT(*) + 3
FROM ENROLLMENTS
WHERE CRN = 41590
```

- Terms can also be used inside aggregations:
Capacity if 5 more chairs are placed in every room.

```
SELECT SUM(SEATS + 5) FROM COURSES
```

Null Values in Aggregations

- Usually, null values are ignored (filtered out) before the aggregation function is applied.
- Only `COUNT(*)` includes null values.
- The only difference between `COUNT(INSTRUCTOR)` and `COUNT(*)` is that the first counts only those rows where “INSTRUCTOR” is not null, whereas the second counts all rows.

Otherwise, the actual attribute value is not important for `COUNT`, and one probably should use `COUNT(*)`. Of course, in `COUNT(DISTINCT SSN)` the attribute is important.

Empty Aggregations

- If the input set is empty, most aggregations yield a null value, only **COUNT** returns 0.

This is counter-intuitive at least for the **SUM**. One would expect that the **SUM** over the empty set is 0, but in SQL it returns **NULL**. (One reason for this behaviour might be that the **SUM** aggregation function cannot detect a difference between the empty input set because there was no qualifying tuple and the empty input set because all qualifying tuples had a null value in this argument.)

- Since it may happen that no row satisfies the **WHERE**-condition, programs must be prepared to process the resulting null value.

Alternative: Use e.g. **NVL** in Oracle to replace the null.

Overview

1. Aggregations I: Aggregation Functions

2. Aggregations II: GROUP BY, HAVING

3. Subqueries

4. UNION, ORDER BY

5. SQL-92 Joins, Outer Join in Oracle

GROUP BY (1)

- The above SQL constructs can produce a single aggregated output row only.
- The **GROUP BY** clause allows one to aggregate in groups rather than aggregate all tuples.
- E.g.: Compute the number of enrolled students for every course.

```
SELECT  CRN, COUNT(*)  
FROM    ENROLLMENTS  
GROUP BY CRN
```

CRN	COUNT(*)
41590	2
31864	2
22332	1

GROUP BY (2)

- The GROUP BY clause splits the resulting table after evaluation of FROM and WHERE into groups that have the same value in the GROUP BY columns.

SSN	CRN
123-45-6789	41590
111-22-3333	41590
111-22-3333	31864
543-76-9821	31864
543-76-9821	22332

- The aggregation is then done over every group.
So there will be one output row for every group.

GROUP BY (3)

- This construction can never produce empty groups. So it is impossible that a `COUNT(*)` results in the value 0.

The value 0 can be produced with `COUNT(A)` where the attribute `A` is null. If a query must produce groups with count 0, probably an outer join is needed (see below).

- On the other hand, simple aggregations (without `GROUP BY`) will always produce exactly one output row, and it is possible that their input set is empty (then `COUNT(*)` can be 0).

A `GROUP BY` query can result in none, one, or many output rows.

GROUP BY (4)

- Since the GROUP BY attributes have a unique value for every group, they can be used in the SELECT-list.

Other attributes can be used under SELECT only inside aggregations.

- E.g. this is illegal:

```
SELECT    C.CRN, TITLE, COUNT(*)   Wrong!
FROM      COURSES C, ENROLLMENTS E
WHERE     C.CRN = E.CRN
GROUP BY  C.CRN
```

Title does not appear under GROUP BY, therefore it cannot be used in the SELECT-list outside an aggregation function. This is especially strange since CRN is a key of COURSES, so that TITLE is actually unique in the groups. But the SQL rule is purely syntactic.

GROUP BY (5)

- Thus, one must group by CRN and TITLE:

```
SELECT  C.CRN, TITLE, COUNT(*) "#"  
FROM    COURSES C, ENROLLMENTS E  
WHERE   C.CRN = E.CRN  
GROUP BY C.CRN, TITLE
```

CRN	TITLE	#
41590	DB ... Design	2
31864	Client-Server	2
22332	DB Management	1

GROUP BY (6)

- Exercise: Is there any semantical difference between

```
SELECT  TITLE, COUNT(*)
FROM    COURSES C, ENROLLMENTS E
WHERE   C.CRN = E.CRN
GROUP BY TITLE
```

and the query which additionally groups by the CRN, but does not print it?

```
SELECT  TITLE, COUNT(*)
FROM    COURSES C, ENROLLMENTS E
WHERE   C.CRN = E.CRN
GROUP BY TITLE, C.CRN
```

GROUP BY (7)

- The sequence of attributes in the GROUP BY clause is not important:

GROUP BY A, B means that two tuples t, u belong into the same group if $t.A = u.A$ and $t.B = u.B$.

GROUP BY B, A means that two tuples t, u belong into the same group if $t.B = u.B$ and $t.A = u.A$.

- GROUP BY is evaluated before the SELECT clause.

Thus, one cannot refer to new attribute names:

```
SELECT    CRN "Course", COUNT(*) "Num Students"  
FROM      ENROLLMENT  
GROUP BY "Course"    Wrong!
```

GROUP BY (8)

- This means that it is also not possible to group by values computed in the SELECT-clause:

```
SELECT    FLOOR((SEATS-1)/25)+1 "Size", COUNT(*)
FROM      COURSES
GROUP BY  1 -- meaning first column    Wrong!
```

- Oracle, SQL Server, and DB2 all support GROUP BY with arbitrary expressions, but the SQL92 standard permits GROUP BY only with column names.

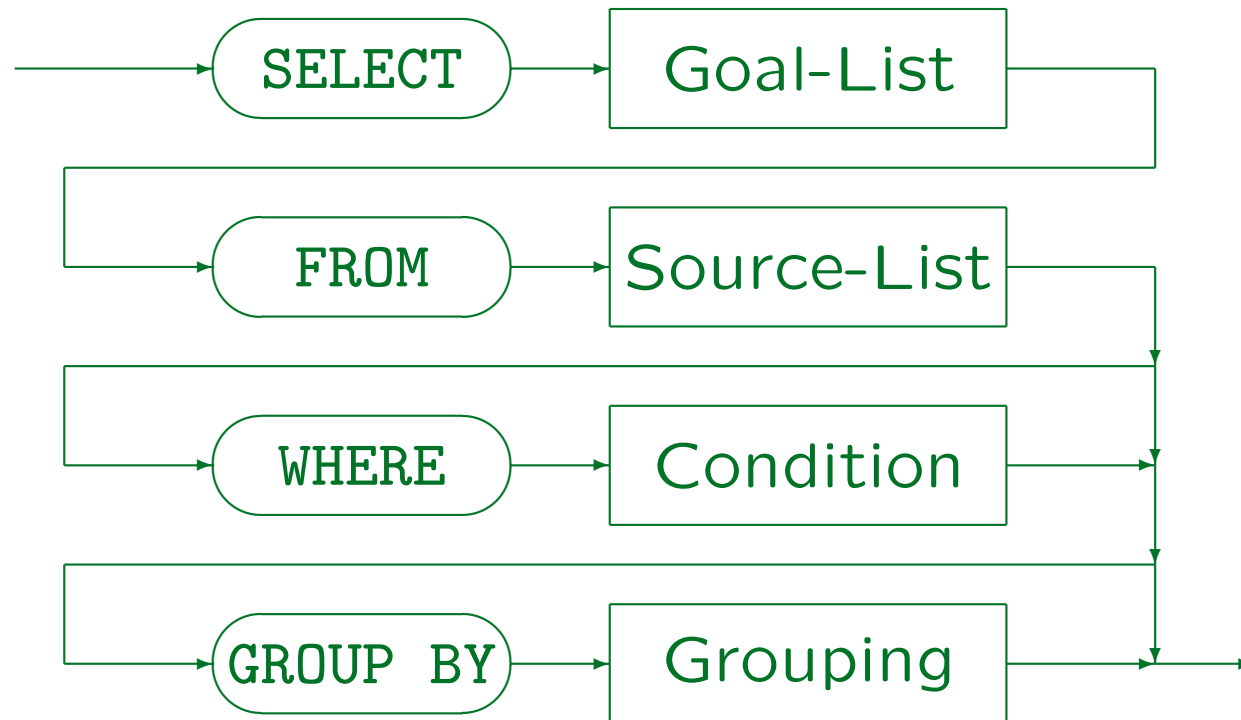
I.e. GROUP BY FLOOR((SEATS-1)/25)+1 will work in most major systems, but is not completely portable. Better use a view or a subquery under FROM (see below).

GROUP BY (9)

Warning:

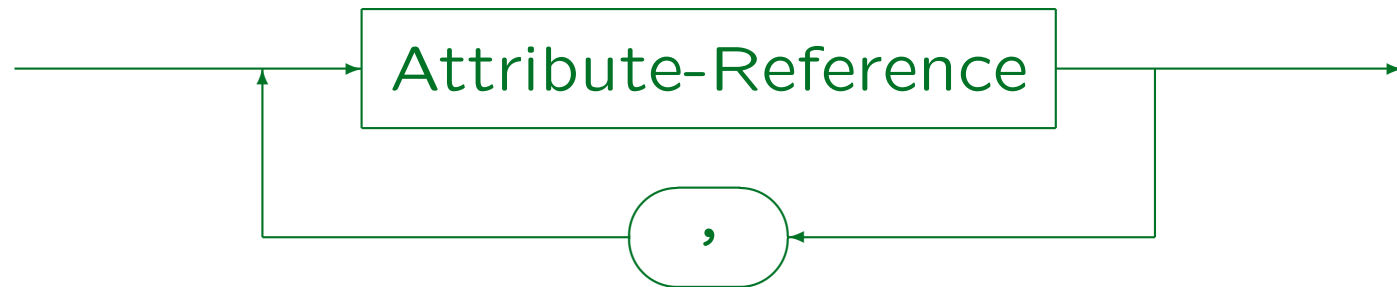
- Many students mix up “GROUP BY” and “ORDER BY”:
 - ◇ GROUP BY is important for the query result.
 - ◇ ORDER BY is only cosmetic (for a nice printout).
- GROUP BY usually internally sorts the tuples (so that tuples with the same values are adjacent).
- But then GROUP BY does the grouping, whereas the sort for the ORDER BY is done at the very end.
- Sometimes, the DBMS may evaluate the GROUP BY in more efficient ways without sorting.

Syntax (1)



Syntax (2)

Grouping:



- E.g. `GROUP BY TITLE, C.CRN`
- Oracle, SQL Server, and DB2 support the more general “Term” instead of “Attribute-Reference”.

Of course, no aggregation functions are permitted under `GROUP BY`.

HAVING (1)

- Aggregations cannot be used in the **WHERE**-clause.
- But sometimes aggregations are needed to filter output rows, not only for computing output values.
- For this reason, SQL has a second kind of condition, the **HAVING** clause. The purpose of the **HAVING** clause is to eliminate whole groups.
- Aggregation operators can be used in the **HAVING**-condition. But as under **SELECT**, outside aggregations, only **GROUP BY** attributes can be used.

HAVING (2)

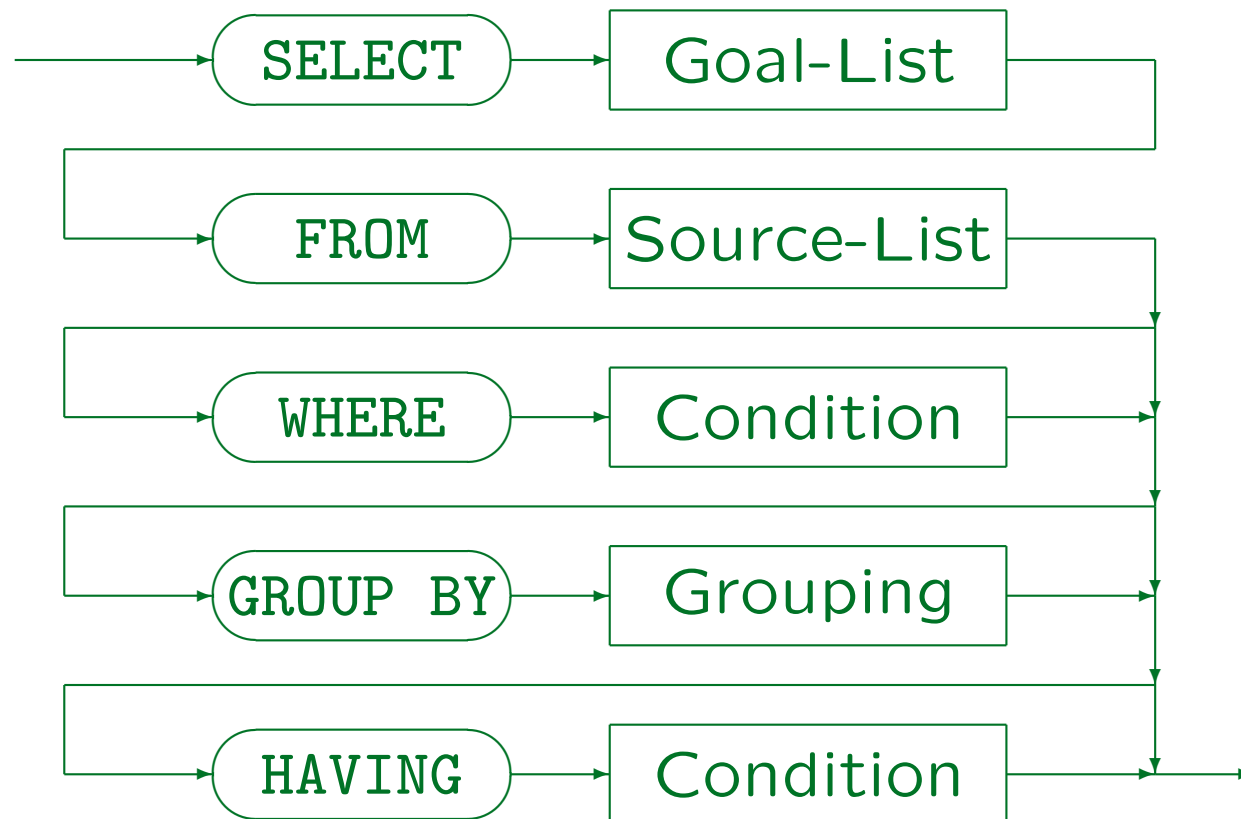
- Which students are enrolled in at least two courses?

```
SELECT  FIRST, LAST
FROM    STUDENTS S, ENROLLMENTS E
WHERE   S.SSN = E.SSN
GROUP BY S.SSN, FIRST, LAST
HAVING  COUNT(*) >= 2
```

FIRST	LAST
Ann	Miller
David	Meyer

- The `WHERE` condition refers to single tuple combinations, the `HAVING` condition to entire groups.

Syntax: SELECT-Query



Evaluation

1. All combinations of rows from tables under FROM are considered.
2. The WHERE-condition selects a subset of these.
3. The remaining joined tuples are split into groups having equal values for the GROUP BY-attributes.
4. Groups of tuples which do not satisfy the condition in the HAVING-clause are eliminated.
5. One output tuple for every group is produced by evaluating the terms in the SELECT-clause.

Syntax: Restrictions (1)

- An aggregation is done if
 - ◇ an aggregation function is used in the `SELECT`-list,
 - ◇ or the `GROUP BY` or `HAVING`-clause is present.
- If an aggregation is done, then: Only `GROUP BY` attributes can be used under `SELECT` or `HAVING` outside aggregation functions.

Inside aggregation functions, i.e. as their arguments, all attributes can be used. E.g. `AVG(A)/B`: The attribute `A` appears inside an aggregation function, `B` outside.

Syntax: Restrictions (2)

- The `WHERE`-condition cannot contain aggregations (Except in subqueries, see below.).
- A `HAVING`-clause without a `GROUPBY`-clause is legal, but very uncommon: It could return only 0 or 1 output rows.

WHERE vs. HAVING

- Normally, the restrictions uniquely define whether a condition must be put under **WHERE** or under **HAVING**.

Only if a condition contains only **GROUP BY**-attributes, but no aggregations, it would be allowed in both clauses.

- If both is possible, it is much more efficient to put it under **WHERE**. E.g. this query is legal, but slow and needs lots of memory:

```
SELECT    C.CRN, TITLE, COUNT(*)
FROM      COURSES C, ENROLLMENTS E
GROUP BY  C.CRN, E.CRN, TITLE
HAVING    C.CRN = E.CRN AND COUNT(*) >= 2
```

Exercises (1)

Consider a database containing invoices (bills):

CUSTOMERS						
CNO	NAME	ADR	CITY	STATE	ZIP	PHONE
100	Brass	...	Pittsburgh	PA	15213	624-9404
101	Spring	...	Pittsburgh	PA	15213	624-9424

INVOICES				
INO	CNO	DATE	PAID	AMOUNT
2000	100	May 05, 2000	N	2345.00
2001	100	Oct 20, 2000	N	1000.00
2002	101	Apr 24, 2000	Y	158.00

Exercises (2)

Please formulate the following queries in SQL:

- What is the range of invoice amounts?
I.e. what is the lowest and highest amount?
- What is the sum of all outstanding bills?
Outstanding bills are marked with Paid='N'.
- Print the total amount of all unpaid bills for every customer (output customer number and name).
- Print name and phone number of all customers having at least \$1000 as the sum of all their unpaid bills.

Overview

1. Aggregations I: Aggregation Functions
2. Aggregations II: GROUP BY, HAVING
3. Subqueries
4. UNION, ORDER BY
5. SQL-92 Joins, Outer Join in Oracle

Nonmonotonic Behaviour (1)

- Without aggregations, SQL queries compute monotonic functions on the existing tables: If further rows are inserted, one gets at least the same answers as before, and maybe more.
- However, not all queries behave monotonically in this way: E.g. print students that are not registered for any course.
- E.g. currently Mary Jones is not registered for any course. If a course registration were inserted for her, she would no longer qualify.

Nonmonotonic Behaviour (2)

- Therefore, this query cannot be formulated with the SQL constructs that were introduced so far (except aggregations).
- Even if it would be possible to formulate the query with aggregations, that would not be a natural formulation.

Actually, there is no way to formulate this query with the constructs introduced so far (even with aggregations). The reason is that if the `ENROLLMENTS` table is empty, any query that mentions `ENROLLMENTS` under `FROM` will return the empty result (or with aggregations a result of exactly one row). However, the `STUDENTS` table may contain two students.

Nonmonotonic Behaviour (3)

- Relational algebra (a formal, theoretical query language for the relational model) has the set difference as the only nonmonotonic operation

$$R - S = \{t \in R \mid t \notin S\}.$$

- The set difference computes the tuples that are contained in one subquery, but not contained in the result of another subquery.
- E.g. in the example, R is the set of all students, and S is the set of students that are registered for courses.

Nonmonotonic Behaviour (4)

- In the natural language version of queries, formulations like “there is no”, “does not exist” indicate nonmonotonic behaviour.
- Furthermore, “for all”, “the minimal/maximal”, also indicate nonmonotonic behaviour: In this case a violation of the “for all” condition must not exist.
For some such queries, a formulation with `HAVING` might be natural.
- When formulating queries in SQL, it is important to check whether the query requires that certain tuples do not exist.

NOT IN (1)

- With **IN** (\in) and **NOT IN** (\notin) it is possible to check whether an attribute value appears in a set that is computed by another SQL query.
- E.g. students that are not registered for any course:

```
SELECT FIRST, LAST
FROM STUDENTS
WHERE SSN NOT IN (SELECT SSN
                  FROM ENROLLMENTS)
```

FIRST	LAST
Mary	Jones

NOT IN (2)

- At least conceptually, the subquery is evaluated, before the execution of the main query starts:

STUDENTS		
<u>SSN</u>	FIRST	LAST
123-45-6789	John	Smith
111-22-3333	Ann	Miller
543-76-9821	David	Meyer
900-50-3000	Mary	Jones

Result of Subquery
SSN
123-45-6789
111-22-3333
111-22-3333
543-76-9821
543-76-9821

- Then for every **STUDENTS** tuple, a matching **SSN** is searched in the subquery result. If there is none, the student name is printed.

NOT IN (3)

- It is also possible to use **IN** (without NOT) for an element test.
- This is relatively seldom done, since it is equivalent to a join, which could be written without a subquery.
- But sometimes this formulation is more elegant. It might also help to avoid duplicates.

Or to get exactly the required duplicates (see example).

NOT IN (4)

- E.g. students that are registered for at least one course:

```
SELECT FIRST, LAST
FROM STUDENTS
WHERE SSN IN (SELECT SSN
              FROM ENROLLMENTS)
```

- Without subquery, duplicate elimination is required.
Exercise: Is there a difference to the above query?

```
SELECT DISTINCT FIRST, LAST
FROM STUDENTS S, ENROLLMENTS E
WHERE S.SSN = E.SSN
```

NOT IN (5)

- In SQL-86, the subquery must have a single output column.

So that the subquery result is really a set (or multiset), and not an arbitrary relation.

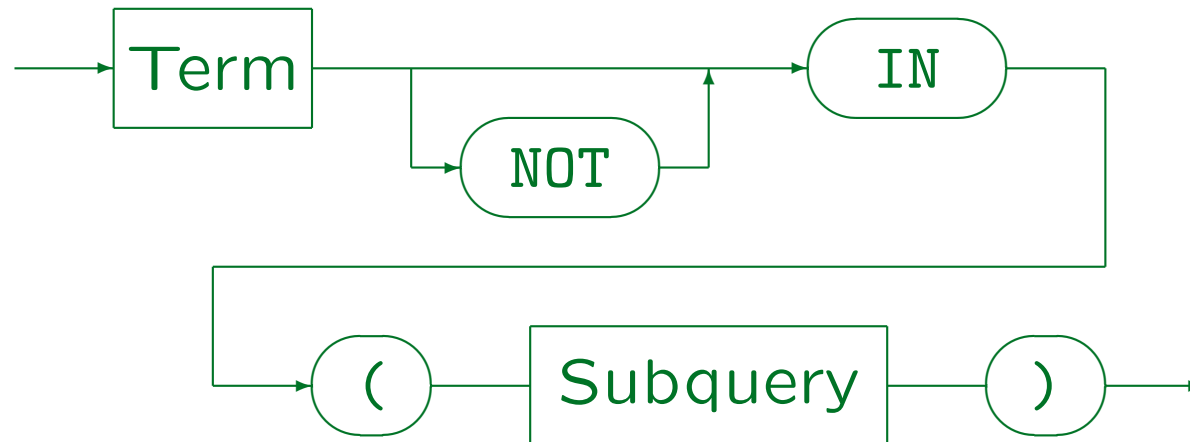
- In SQL-92, comparisons were extended to the tuple level, and therefore it is possible to write e.g.

```
WHERE (FIRST, LAST) NOT IN (SELECT FIRST, LAST  
                             FROM ...)
```

But is not very portable. E.g. SQL Server does not support it. An EXISTS subquery (see below) might be better. Oracle and DB2 do allow IN with multiple columns.

NOT IN (6)

Atomic Formula (Form 6):



- The Subquery must result in a table with a single column (a set).
- However, in SQL-92, Oracle, and DB2 it is possible to write a tuple on the left hand side in the form $(Term_1, \dots, Term_n)$. Then the subquery must result in a table with exactly n columns.
- The column names on the left and right hand side of IN do not have to match, but the data types must be compatible.

NOT EXISTS (1)

- It is possible to check in the outer query whether the result of the subquery is empty (**NOT EXISTS**).
- In the inner query, tuple variables declared in the **FROM** clause of the outer query can be accessed.

This is actually also possible for **IN** subqueries, but there it is an unnecessary and unexpected complication (bad style).

- This means that the subquery has to be evaluated once for every assignment of values to the accessed tuple variables in the outer query. The subquery can be seen as parameterized.

NOT EXISTS (2)

- E.g. students not registered for any course (again):

```
SELECT FIRST, LAST
FROM STUDENTS S
WHERE NOT EXISTS (SELECT * FROM ENROLLMENTS E
                  WHERE E.SSN = S.SSN)
```

- The tuple variable *S* loops over the four rows in *STUDENTS*. Conceptually, the subquery is evaluated four times. Each time, *S.SSN* is replaced by the *SSN* value of the current tuple.

Of course, the DBMS is free to choose another, more efficient evaluation strategy if that evaluation strategy is guaranteed to give the same result.

NOT EXISTS (3)

- First, **S** points to the **STUDENTS** tuple

SSN	FIRST	LAST
123-45-6789	John	Smith

- **S.SSN** in the subquery is conceptually replaced by '123-45-6789' and the following query is executed:

```
SELECT * FROM ENROLLMENTS E
WHERE E.SSN = '123-45-6789'
```

SSN	CRN
123-45-6789	41590

- The result is not empty. Thus, the **NOT EXISTS** condition in the outer query is not satisfied for this tuple **S**, and John Smith is not printed.

NOT EXISTS (4)

- The same happens for the second row in **STUDENTS**.
The subquery is executed for $S.SSN = '111-22-3333'$:

```
SELECT * FROM ENROLLMENTS E
WHERE E.SSN = '111-22-3333'
```

SSN	CRN
111-22-3333	41590
111-22-3333	31864

- The result is not empty, therefore the **NOT EXISTS** condition is not satisfied.
- Also for the third row in **STUDENTS**, the condition is not satisfied.

NOT EXISTS (5)

- Finally, s points to the STUDENTS tuple

SSN	FIRST	LAST
900-50-3000	Mary	Jones

- For $s.SSN = '900-50-3000'$, the result of the subquery is empty:

```
SELECT * FROM ENROLLMENTS E
WHERE E.SSN = '900-50-3000'
```

no rows selected

- Thus, the NOT EXISTS condition is satisfied for this tuple s . Mary Jones is printed as the query result.

NOT EXISTS (6)

- While in the inner query, tuple variables from the outer query can be accessed, the converse is illegal:

```
SELECT FIRST, LAST, E.CRN Wrong!
FROM   STUDENTS S
WHERE  NOT EXISTS (SELECT * FROM ENROLLMENTS E
                  WHERE E.SSN = S.SSN)
```

- This works like global and local variables: Variables defined in the outer query are valid for the entire query, variables defined in the subquery are valid only in the subquery.

This corresponds to the block structure of e.g. Pascal.

NOT EXISTS (7)

- Subqueries that access variables from the outer query are called “**correlated subqueries**”.

Correlated subqueries can be understood as being parameterized with the tuples chosen in the outer query. There can be optimizations, but conceptually they are executed once for every assignment of tuples to the tuple variables in the outer query.

- Subqueries that do not access variables from the outer query are called “**non-correlated subqueries**”.

It suffices to evaluate a non-correlated subquery only once (since the result does not depend on the tuples chosen for the tuple variables of the outer query).

NOT EXISTS (8)

- Non-correlated subqueries with NOT EXISTS are almost always an error (but they are ok with IN):

```
SELECT FIRST, LAST      Wrong!  
FROM STUDENTS S  
WHERE NOT EXISTS (SELECT * FROM ENROLLMENTS E)
```

- Here the join-condition in the subquery was forgotten, and it became a non-correlated subquery.
- If there is at least one row in the ENROLLMENTS table, no matter for what student (since there is no join condition), the NOT EXISTS will always be false.

NOT EXISTS (9)

- Until now, for attribute references without tuple variable (“unqualified attribute name”), there had to be a unique tuple variable to which it can refer.
- For subqueries, it is only required that there is a unique nearest tuple variable which has this attribute, e.g. this is legal (but bad style):

```
SELECT FIRST, LAST
FROM STUDENTS S
WHERE NOT EXISTS (SELECT * FROM ENROLLMENTS E
                  WHERE SSN = S.SSN)
```

NOT EXISTS (10)

- In general, for attribute reference without tuple variables, the SQL parser searches the FROM-clauses beginning from the current subquery towards outer queries (there can be several nesting levels).
- The first FROM-clause that declares a tuple variable with this attribute must have exactly one such variable. Then the attribute refers to this variable.
- This rule helps that non-correlated subqueries can be developed independently and inserted into another query without any change (so it makes sense).

NOT EXISTS (11)

- It is also legal to declare tuple variables in the subquery that have the same name as tuple variables in the outer query.

```
SELECT FIRST, LAST
FROM STUDENTS X
WHERE NOT EXISTS (SELECT * FROM ENROLLMENTS X
                  WHERE ???)
```

- References to X in the subquery mean ENROLLMENTS X. The variable declared in the outer query becomes “shadowed”: It cannot be accessed in the subquery.

NOT EXISTS (12)

- What is the smallest class size (number of seats)?

```
SELECT CRN, TITLE
FROM   COURSES X
WHERE  NOT EXISTS (SELECT * FROM COURSES Y
                  WHERE Y.SEATS < X.SEATS)
```

- I.e. a course X is selected if there is no course Y with a strictly smaller number of seats than X.
- Since for `NOT EXISTS` the returned columns do not matter, “`SELECT *`” should be used in the subquery.

Some authors say that in some systems `SELECT null` or `SELECT 1` is actually faster than `SELECT *`. “`SELECT null`” does not work in DB2.

NOT EXISTS (13)

Atomic Formula (Form 7):



- A subquery is an expression of the form `SELECT ... FROM ... [WHERE ...] [GROUP BY ...] [HAVING ...]`.
[...] means that these parts are optional. SQL-92 also allows `UNION` (see below) in subqueries, SQL-86 does not.
- `ORDER BY` is not allowed in subqueries.
It would make no sense there, it is only for the final output.
- Subqueries must be enclosed in parentheses (...).

NOT EXISTS (13)

- It is possible to use **EXISTS** without negation.
- Which students are enrolled for at least one course?

```
SELECT SSN, FIRST, LAST
FROM STUDENTS S
WHERE EXISTS (SELECT * FROM ENROLLMENTS E
              WHERE E.SSN = S.SSN)
```

- But the same query can be done with a usual join:

```
SELECT DISTINCT S.SSN, FIRST, LAST
FROM STUDENTS S, ENROLLMENTS E
WHERE S.SSN = E.SSN
```

Common Errors (1)

Exercises:

- Would this query find students without enrollments in the database? If not, what does it compute?

```
SELECT DISTINCT S.SSN, FIRST, LAST
FROM   STUDENTS S, ENROLLMENTS E
WHERE  S.SSN <> E.SSN
```

- What about this query?

```
SELECT DISTINCT S.SSN, FIRST, LAST
FROM   STUDENTS S, ENROLLMENTS E
WHERE  S.SSN = E.SSN AND E.CRN IS NULL
```

Common Errors (2)

- It is important to understand that the absence/non-existence of a row is very different than the existence of a row with a different value.

If the requested query behaves in a non-monotonic fashion (i.e. insertion of a row could invalidate an answer), then `NOT EXISTS`, `NOT IN`, `<>` `ALL` etc. are required.

- There is no way to write it without a subquery.

Except possibly using an outer join. Aggregations also change when tuples are inserted, but without subquery, they cannot express “for all” or “not exists”.

Common Errors (3)

- Does this query compute the courses with a minimal number of seats? If not, what does it compute?

```
SELECT DISTINCT X.CRN, X.TITLE
FROM   COURSES X, COURSES Y
WHERE  X.SEATS < Y.SEATS
```

Common Errors (4)

- Another common error is to use a non-correlated subquery with NOT EXISTS (forgetting the join condition in the subquery). Does this compute courses for which Ann Miller is not registered?

```
SELECT CRN, TITLE
FROM   COURSES C
WHERE  NOT EXISTS
      (SELECT *
       FROM   ENROLLMENTS E, STUDENTS S
       WHERE  E.SSN = S.SSN
       AND    S.FIRST='Ann' AND S.LAST='Miller')
```

ALL, ANY, SOME (1)

- It is possible to compare a value with all values in a set. One can require that the comparison returns true for all set elements (**ALL**) or for at least one set element (**ANY, SOME**).

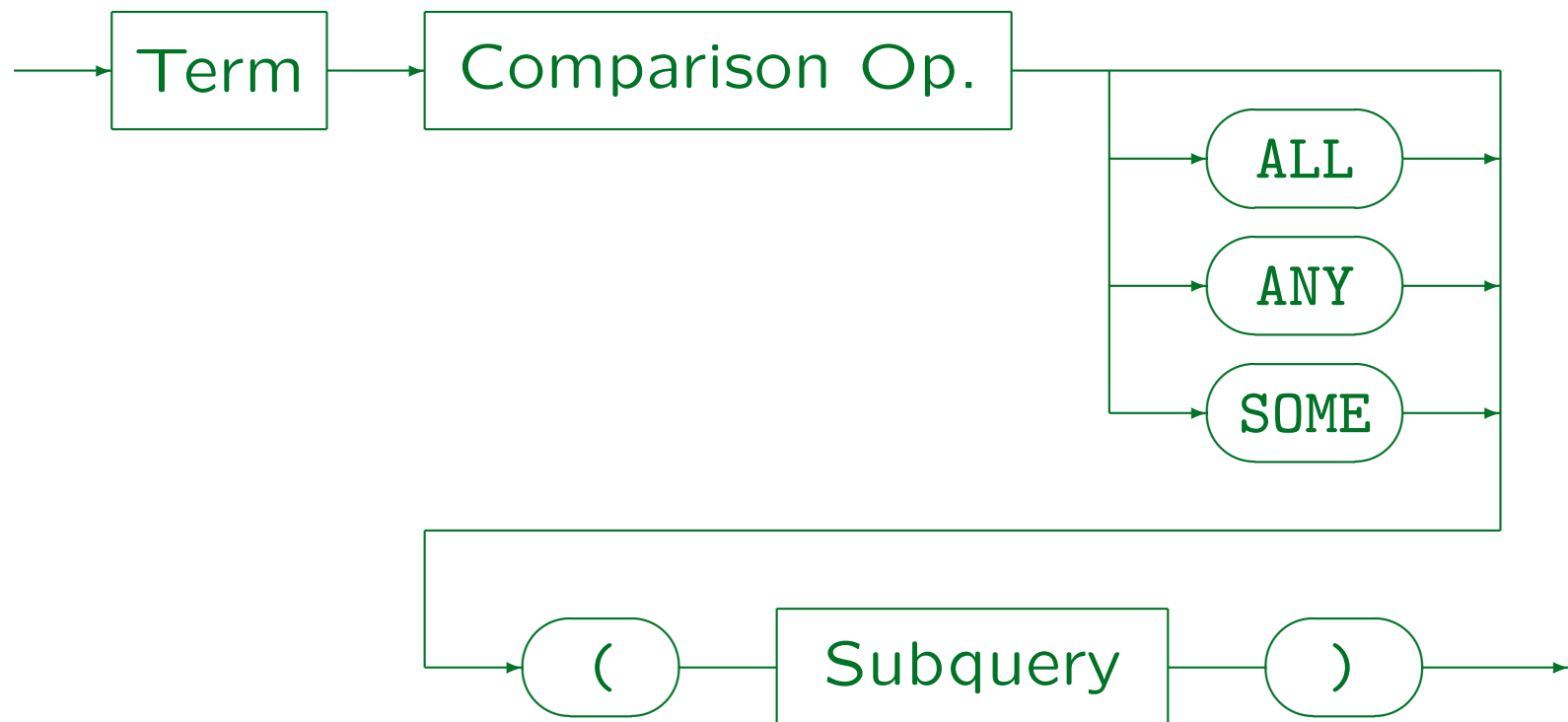
- Which class has the minimal number of seats?

```
SELECT CRN, TITLE
FROM COURSES
WHERE SEATS <= ALL (SELECT SEATS
                    FROM COURSES)
```

- The Condition “**NOT SEATS > ANY (...)**” would give the same result (logically equivalent).

ALL, ANY, SOME (2)

Atomic Formula (Form 8):



ALL, ANY, SOME (3)

Syntactic Remarks:

- **ANY** and **SOME** are synonyms.
- “**x IN S**” is equivalent to “**x = ANY S**”.
- The subquery must result in a table with a single column.

SQL92 allows comparisons also on a tuple basis. Oracle supports this only with $\langle \rangle$ and $=$, DB2 supports only $=ANY$ (which is equivalent to IN). SQL86 and SQL Server do not support tuple comparisons.

- If none of the keywords **ALL**, **ANY**, **SOME** are present, the subquery must yield exactly one row (i.e. a single data element).

Single Value Subqueries

- Which class has the minimal number of seats?

```
SELECT CRN, TITLE
FROM COURSES
WHERE SEATS = (SELECT MIN(SEATS)
               FROM COURSES)
```

- In SQL92, DB2, and SQL Server, a subquery returning a single data element can be used as a term/expression. Thus, this is equally legal:

```
WHERE (SELECT MIN(SEATS)
       FROM COURSES) = SEATS
```

In Oracle8 and SQL86, the subquery must be on the right hand side.

Subqueries under FROM (1)

- Since the result of an SQL-query is a table, it is natural that one can write a subquery instead of a table name in the FROM-clause.
- This was not allowed in SQL-86, and at that time SQL was often criticized as having “not orthogonal constructs”, which cannot be combined arbitrarily.

Also because of views (virtual tables defined by queries, see below), this feature is very natural. Older systems had a lot of restrictions on the use of views in queries. For SQL-92 systems, no such restrictions exist: A view is simply an abbreviation (macro), its name can be replaced by its definition in the FROM-clause.

Subqueries under FROM (2)

- This feature is needed e.g. for nested aggregations.
- What is the average number of students registered for a course?

```
SELECT AVG(X.NUM)
FROM (SELECT CRN, COUNT(*) NUM
      FROM ENROLLMENTS
      GROUP BY CRN) X
```

X	
CRN	NUM
41590	2
31864	2
22332	1

AVG(X.NUM)
1.67

This only counts courses with at least one student. In order to count all courses, one needs an outer join (see below).

Subqueries under FROM (3)

- SQL92, SQL Server, and DB2 require declaring a tuple variable for the subquery; in Oracle this is optional.
- SQL92, DB2, and SQL Server (but not Oracle8), permit to rename columns in this way:

```
FROM (...) X(CRN, NUM)
```

- In Oracle, columns can only be renamed inside the subquery.

All systems support the specification of new column names in the `SELECT`-clause, so that is the more portable way.

Subqueries under FROM (4)

- Oracle also supports nested aggregations written in this way:

```
SELECT    AVG(COUNT(*))    Only Oracle!  
FROM      ENROLLMENTS  
GROUP BY CRN
```

This is completely non-standard (not supported in SQL92, DB2, SQL Server).

Since it is much shorter than the equivalent standard query, it might be handy to use this when writing ad-hoc queries. However, in application programs, one should not create unnecessary portability problems.

Subqueries under SELECT

- Since in SQL92, DB2, and SQL Server a subquery returning a single data element can be used as an expression, subqueries are also allowed in the SELECT-clause. Oracle 8.0 does not support this.
- This makes the GROUP BY clause unnecessary.
E.g. print for every course the number of enrolled students (including courses with 0 students):

```
SELECT CRN, TITLE, (SELECT COUNT(*)  
                    FROM ENROLLMENTS E  
                    WHERE E.CRN = C.CRN)  
FROM COURSES C
```

Exercises (1)

Consider again a database containing invoices (bills):

CUSTOMERS						
CNO	NAME	ADR	CITY	STATE	ZIP	PHONE
100	Brass	...	Pittsburgh	PA	15213	624-9404
101	Spring	...	Pittsburgh	PA	15213	624-9424

INVOICES				
INO	CNO	DATE	PAID	AMOUNT
2000	100	May 05, 2000	N	2345.00
2001	100	Oct 20, 2000	N	1000.00
2002	101	Apr 24, 2000	Y	158.00

Exercises (2)

Please formulate the following queries in SQL:

- Which Customers (CustNo and Name) do not have any unpaid bills?
- Print the complete customer record of the customer(s) having the highest bill (paid or unpaid).
- What is the average amount sold to a customer (sum of bills)?

Overview

1. Aggregations I: Aggregation Functions
2. Aggregations II: GROUP BY, HAVING
3. Subqueries
4. UNION, ORDER BY
5. SQL-92 Joins, Outer Join in Oracle

UNION (1)

- In SQL it is possible to combine the results of two queries by **UNION**.

$R \cup S$ is the set of all tuples contained in R , in S , or in both.

- **UNION** is needed since otherwise there is no way to construct **one result column that contains values drawn from different tables/columns**.

This is necessary e.g. when subclasses are represented by different tables. For instance, there may be one table `GRADUATE_COURSES` and another table `UNDERGRADUATE_COURSES`.

- **UNION** is also very useful for case analysis (to code an **if ... then ... else ...**).

UNION (2)

- The subqueries which are operands to UNION must return tables with the same number of columns and corresponding columns must have compatible types.

The attribute names do not have to be equal. Oracle and SQL Server use the attribute names from the first operand in the result. DB2 uses artificial column names (1, 2, ...) if the input column names differ.

- SQL distinguishes between
 - ◇ UNION: \cup with duplicate elimination, and
 - ◇ UNION ALL: concatenation (retains duplicates).Duplicate elimination is quite expensive.

UNION (3)

- List all courses together with their number of enrolled students (including those with 0 students).

```
SELECT    C.CRN, TITLE, COUNT(*)
FROM      COURSES C, ENROLLMENTS E
WHERE     C.CRN = E.CRN
GROUP BY C.CRN, TITLE
UNION ALL
SELECT    CRN, TITLE, 0
FROM      COURSES C
WHERE     CRN NOT IN (SELECT CRN
                      FROM ENROLLMENTS)
```

UNION (4)

- Print all courses with a column SIZE that contains “small” for 1–20 seats, “medium” for 21–50 seats, and “large” for more seats.

```
SELECT CRN, TITLE, 'small' SIZE
FROM COURSES WHERE SEATS <= 20
UNION ALL
SELECT CRN, TITLE, 'medium' SIZE
FROM COURSES WHERE SEATS BETWEEN 21 AND 50
UNION ALL
SELECT CRN, TITLE, 'large' SIZE
FROM COURSES WHERE SEATS > 50
```

Other Set Operations in SQL

- SQL Server and SQL86 support only **UNION [ALL]**.
- The SQL92 standard also contains **EXCEPT** (set difference, $-$) and **INTERSECT** (\cap).

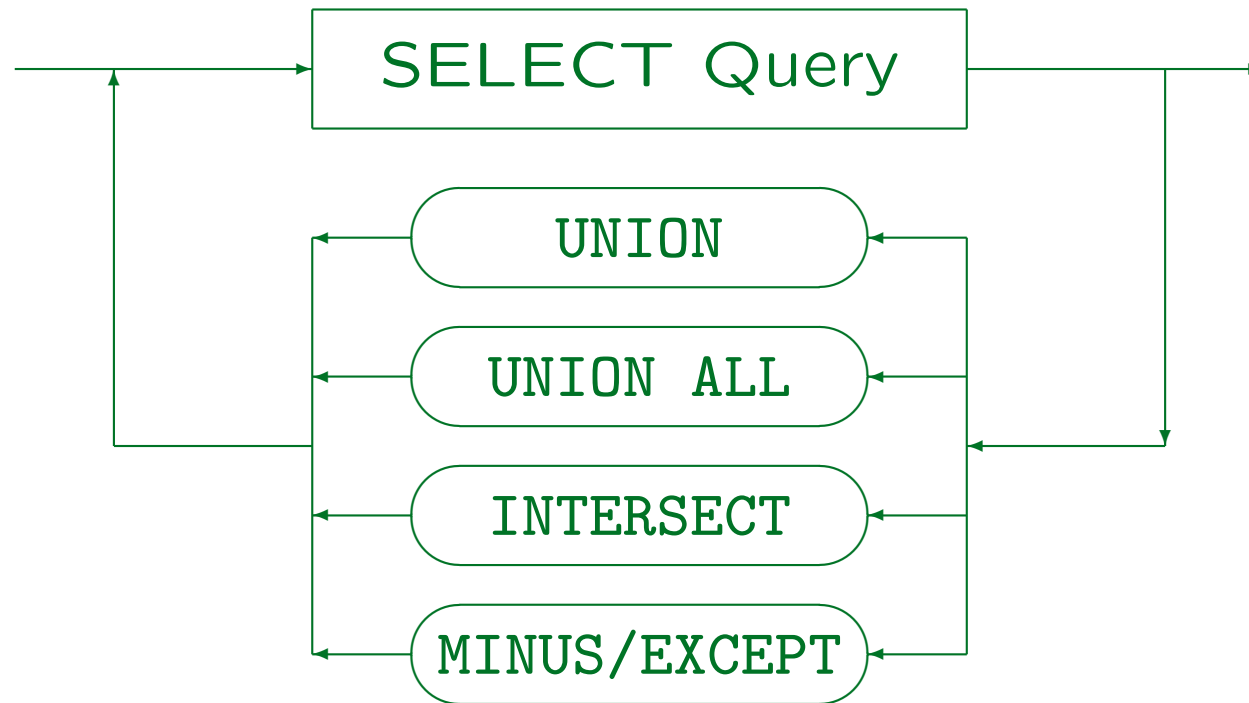
These operators are supported in DB2. In Oracle 8.0, the operator is called **MINUS** instead of **EXCEPT**. **ALL** for **MINUS** and **INTERSECT** is not supported in Oracle.

- These operations add nothing to the expressivity to the language.

Queries containing **EXCEPT**/**MINUS** and **INTERSECT** can be transformed into equivalent SQL-queries without these constructs, but queries containing **UNION** in general cannot. So only **UNION** is really important.

UNION etc: Syntax

Table Expression:



Union vs. Join

Exercise:

- Two alternatives for storing the homework, mid-term, and final results of the students are:

Results_1			
STUDENT	H	M	F
Jim Ford	95	60	75
Ann Lloyd	80	90	95

Results_2		
STUDENT	CAT	PCT
Jim Ford	H	95
Jim Ford	M	60
Jim Ford	F	75
Ann Lloyd	H	80
Ann Lloyd	M	90
Ann Lloyd	F	95

- Write SQL queries to translate between the two.

Sorting Output (1)

- Output that is longer than a few lines should be sorted in some understandable way.

It is much easier to search a specific value in a sorted table. Without “ORDER BY” the sequence of output rows means nothing (it depends on the algorithms used in the DBMS).

- However, it is important to understand that developing the logic of the query and nicely formatting the output are two separate things.

Whereas sorting is the only formatting command that found its way into the SQL standard, DBMS tools usually offer more options. E.g. to have a pagebreak one the value in one column changes, to show negative values in red ink, etc.

Sorting Output (2)

- In SQL, one can specify a prioritized list of sorting criteria.
- E.g.: Print the courses sorted by instructor, and for each instructor by the number of seats (largest first). If these two criteria still do not determine the order, use the CRN.

```
SELECT  CRN, INSTRUCTOR, TITLE, SEATS
FROM    COURSES
ORDER BY INSTRUCTOR, SEATS DESC, CRN
```

Sorting Output (3)

- Result of the example query on the previous page:

<u>CRN</u>	INSTRUCTOR	TITLE	SEATS
41590	Brass, S.	DB ... Design	70
37329	Brass, S.	DB ... Design	30
22332	Flynn, R.	DB Management	50
24271	Flynn, R.	Data Structures	50
31864	Spring, M.	Client-Server	30
25688		Document Proc.	35

- It is important here that the last name appears first in the instructor.

Important DB design question: What do I want to do with the data?

Sorting Output (4)

- One can only sort by columns that appear in the output.

E.g. it is impossible to sort by the seat number without printing it. But tools like SQL*Plus can suppress output columns from the query result.

- Sometimes it is necessary to add columns to database tables to get a sort value, e.g. if the “University of Pittsburgh” should appear in a sorted sequence under “P”, and not under “U”.

Sorting Output (5)

- “DESC” means descending (inverse order from high to low values), the default is “ASC” (ascending).
- The “ORDER BY” list can contain multiple columns.

The second column is only used for ordering two tuples which have the same value in the first column, and so on.

- It is also possible to refer to columns by number, e.g.: `ORDER BY 2, 4 DESC, 1`

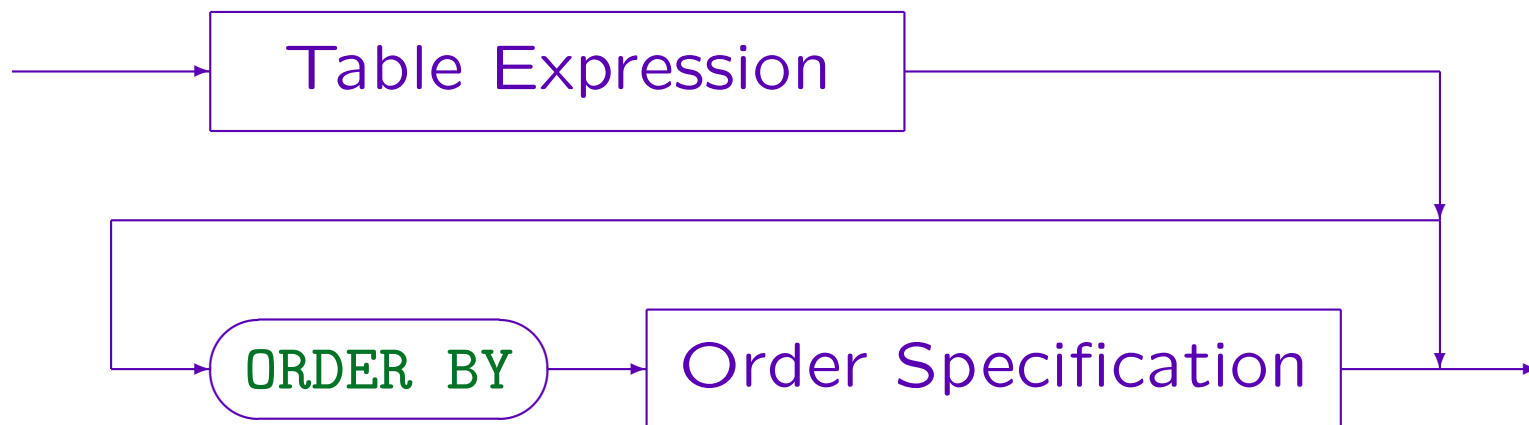
Column numbers refer to the sequence in the SELECT-list. They were important in earlier SQL versions, where one could not explicitly name the result columns. Today, one probably should use column names.

Sorting Output (6)

- The effect of “ORDER BY” is purely cosmetic. It does not change the set of output tuples in any way.
- Therefore, “ORDER BY” can only be applied at the very end of the query. It cannot be used in subqueries.
- Even when multiple SELECT-expressions are combined with UNION, the ORDER BY can only be placed at the very end (it refers to all result tuples).

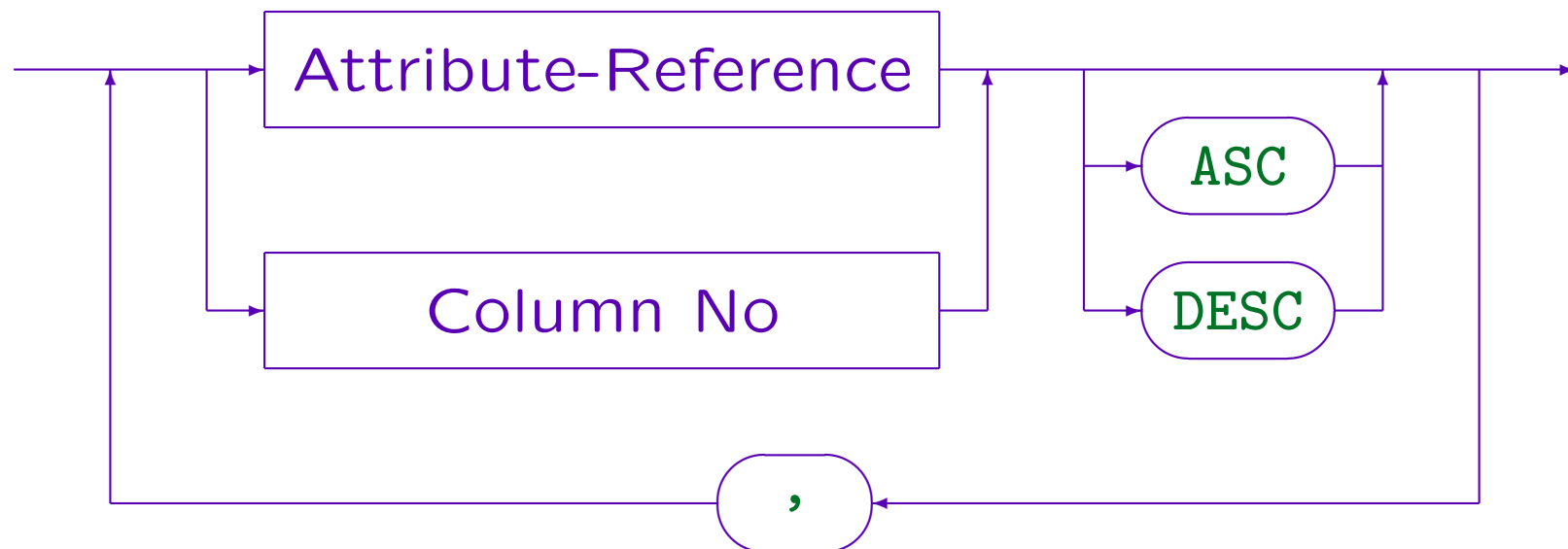
Sorting Output (7)

SQL Query:



Sorting Output (8)

Order Specification:



Overview

1. Aggregations I: Aggregation Functions
2. Aggregations II: GROUP BY, HAVING
3. Subqueries
4. UNION, ORDER BY
5. SQL-92 Joins, Outer Join in Oracle

Joins in SQL-92 (1)

- Relational algebra is a theoretical query language for the relational model.

Variants of it are used internally in a DBMS.

- The operations of relational algebra take relations as operands and yield relations as results.

Just like $+$ and $-$ take numbers as operands and yield numbers as results.

- E.g. **UNION** is one operation of relational algebra.

Since relations are sets (of tuples), the standard operations for sets are also operations of relational algebra.

Joins in SQL-92 (2)

- An important operation of relational algebra is the join (in several variants). Its purpose is to concatenate (combine) selected tuples from two relations.
- The “natural join” combines those tuples that have equal values in columns with the same name.
- E.g. the natural join of STUDENTS and ENROLLMENTS is:

```
SELECT S.SSN "SSN", FIRST, LAST, CRN
FROM   STUDENTS S, ENROLLMENTS E
WHERE  S.SSN = E.SSN
```

Joins in SQL-92 (3)

- In SQL-92 one can write e.g.

```
SELECT FIRST, LAST, CRN  
FROM STUDENTS NATURAL JOIN ENROLLMENTS
```

- Because of the keywords “NATURAL JOIN” the system automatically adds the join condition

```
STUDENTS.SSN = ENROLLMENTS.SSN
```

- SQL-92 permits to use joins in the FROM-clause and even on the outer query level (like UNION).

So one can write quite a lot in “relational algebra style”.

Joins in SQL-92 (4)

- Current systems support the standard only partially.
- Joins were not contained in the SQL-86 standard and are not supported in Oracle 8.0.
- Some types of joins are supported in DB2 and SQL Server, but the above “natural join” is not. The join condition must explicitly be specified:

```
SELECT FIRST, LAST, CRN
FROM   STUDENTS S JOIN ENROLLMENTS E
      ON S.SSN = E.SSN
```

Joins in SQL-92 (5)

- With the explicit join condition, the query is not shorter than the equivalent one with the standard `WHERE` condition.
- The power of SQL is not increased by adding the new join constructs.

Every query with the new join constructs can be translated in an equivalent one that does not use these constructs.

- The reason why joins were added to SQL is probably the “**outer join**”: For the outer join, the equivalent formulation in SQL-86 is significantly longer.

Outer Join (1)

- The usual join eliminates tuples without partner:

$$\begin{array}{|c|c|} \hline A & B \\ \hline a_1 & b_1 \\ \hline a_2 & b_2 \\ \hline \end{array} \bowtie \begin{array}{|c|c|} \hline B & C \\ \hline b_2 & c_2 \\ \hline b_3 & c_3 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline A & B & C \\ \hline a_2 & b_2 & c_2 \\ \hline \end{array}$$

- The left outer join guarantees that tuples from the left table will appear in the result:

$$\begin{array}{|c|c|} \hline A & B \\ \hline a_1 & b_1 \\ \hline a_2 & b_2 \\ \hline \end{array} \left\lrcorner \begin{array}{|c|c|} \hline B & C \\ \hline b_2 & c_2 \\ \hline b_3 & c_3 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline A & B & C \\ \hline a_1 & b_1 & \\ \hline a_2 & b_2 & c_2 \\ \hline \end{array}$$

Rows from the left table are filled with "null" if necessary.

Outer Join (2)

- The right outer join preserves tuples from the right table:

A	B	⋈	B	C	=	A	B	C
a ₁	b ₁		b ₂	c ₂		a ₂	b ₂	c ₂
a ₂	b ₂		b ₃	c ₃			b ₃	c ₃

- The full outer join does not eliminate any tuples:

A	B	⋈	B	C	=	A	B	C
a ₁	b ₁		b ₂	c ₂		a ₁	b ₁	
a ₂	b ₂		b ₃	c ₃		a ₂	b ₂	c ₂
							b ₃	c ₃

Outer Join in SQL-92 (1)

- E.g. number of students per course. Courses without students should be listed with the number 0:

```
SELECT    C.CRN, TITLE, COUNT(SSN)
FROM      COURSES C LEFT OUTER JOIN ENROLLMENTS E
          ON C.CRN = E.CRN
GROUP BY C.CRN, TITLE
```

- All courses are present in the result of the left outer join. In courses without students, the attributes of ENROLLMENTS are filled with null values.
- COUNT(SSN) does not count rows where SSN is null.

Outer Join in SQL-92 (2)

- The equivalent formulation without outer join is significantly longer (9 vs. 4 lines), see page 4-87:

```
SELECT    C.CRN, TITLE, COUNT(*)
FROM      COURSES C, ENROLLMENTS E
WHERE     C.CRN = E.CRN
GROUP BY  C.CRN, TITLE
UNION ALL
SELECT    CRN, TITLE, 0
FROM      COURSES C
WHERE     CRN NOT IN (SELECT CRN
                      FROM ENROLLMENTS)
```

Outer Join in SQL-92 (3)

- Of course, joins and WHERE can be used together:

```
SELECT    C.CRN, TITLE, COUNT(SSN)
FROM      COURSES C LEFT OUTER JOIN ENROLLMENTS E
          ON C.CRN = E.CRN
WHERE     INSTRUCTOR LIKE '%Brass%'
GROUP BY C.CRN, TITLE
```

- It is also possible to do a join and then declare further tuple variables (separated by “,” as usual).
- One can also join the result of joining two tables with a third one (and so on).

Join Syntax in SQL-92 (1)

- SQL-92 has the following join types:
 - ◇ [INNER] JOIN: Usual Join.
 - ◇ LEFT [OUTER] JOIN: Left table tuples are preserved.
 - ◇ RIGHT [OUTER] JOIN: Preserves right table tuples.
 - ◇ FULL [OUTER] JOIN: All input tuples are preserved.
 - ◇ CROSS JOIN: Cross product \times .
 - ◇ UNION JOIN: This is a union that fills the columns of the other table with null values.
- The brackets mean that INNER/OUTER are optional.

Join Syntax in SQL-92 (2)

- The join condition can be specified as follows:
 - ◇ The keyword **NATURAL** in front of the join name.
 - ◇ “**ON** **<Condition>**” follows the join.
 - ◇ “**USING** (A_1, \dots, A_n)” follows the join.
 USING lists join attributes, see next page.
- Only one of these constructs can be used.
- **CROSS JOIN** and **UNION JOIN** have no join condition.
- The first and third possibility produce a table with only one copy of the common columns.

Join Syntax in SQL-92 (3)

- The `USING` clause lists the join attributes. This must be common attributes, i.e. both tables must have attributes with the given names.

```
SELECT FIRST, LAST, CRN  
FROM STUDENTS JOIN ENROLLMENTS USING (SSN)
```

- The join condition is then:

```
STUDENTS.SSN = ENROLLMENTS.SSN
```

Join Syntax in SQL-92 (4)

- Both, DB2 and SQL Server require the use of `ON`. They do not support the natural join or `USING`.
- DB2 does not support `CROSS JOIN` and `UNION JOIN`.
You would simply write a comma for `CROSS JOIN`, so it is not very useful. The `UNION JOIN` is very strange, so again we do not lose much.
- SQL Server does not support `UNION JOIN`.
It does support the cross join.
- Both systems support the inner join as well as left, right, and full outer join.
- Oracle 8i does not support any SQL-92 joins.

Outer Join in Oracle

- In Oracle, one has to append “(+)” to attributes of the table which can be replaced with nulls.

So this protects the other table (not marked with “(+)”). This marking is only done in the WHERE-condition. There are many syntactic restrictions.

- E.g. number of students per course (including 0):

```
SELECT  C.CRAN, TITLE, COUNT(SSN)
FROM    COURSES C, ENROLLMENTS E
WHERE   C.CRAN = E.CRAN (+)
GROUP BY C.CRAN, TITLE
```

- One can also write: “E.CRAN (+) = C.CRAN”.