Objectives

After completing this chapter, you should be able to:

1. Write advanced queries in SQL including:
   - aggregations
   - sub queries
   - UNION

   - Enumerate and explain the clauses of an SQL query.
   - Explain joins in SQL.
   - Evaluate the syntactical correctness of a given query.
   - Evaluate the portability of certain constructs.

Overview

1. Aggregations I: Aggregation Functions

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

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

   Examples:
   - min: min(1, 5, 7, 19, 27) = 19
   - max: max(1, 5, 7, 19, 27) = 27

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

2. Aggregations II: GROUP BY, HAVING

3. Sub queries

4. UNION

5. SQL-92: Outer join in Oracle

References:

- Lippeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- van der Lans: SQL, Der ISO-Standard (in German, there is an English version), Hanser, 1990.
- Microsoft SQL Server Books Online: Accessing and Changing Data.
Aggregations increase the expressivity of a query language. While minimum and maximum can be computed in relational algebra, count and sum cannot. For COUNT and SUM, an unbounded number of tuples can create one output tuple (as for the transitive closure). Any commutative and associative binary operator with a neutral element can be extended to sets, $E + F = E + (F + 0)$. Any commutative and associative binary operator with a neutral element can be extended to sets, $E + F = E + (F + 0)$. Additional aggregation functions in certain systems:

- **Oracle and DB2:** VARIANCE, STDDEV (DB2 also: COUNT_BIG).
- **SQL Server:** VAR, VARP, STDEV, STDEVP.
Simple Aggregations

Evaluation

From courses:

- What is the minimum and maximum number of seats?
- What is the average number of seats in a course?
- How many students are enrolled in at least one course?

The whole power of SQL can be used to compute the set of values which is aggregated (e.g., joins).

Syntax: Aggregation Terms

Term

- All
- DISTINCT
- COUNT
- SUM
- MAX
- MIN
- AVG

AVG (SEATS)

MIN (SEATS), MAX (SEATS)

Multiple aggregations can be computed in the SELECT list.
The arguments of `SUM` and `AVG` must be numeric.

Count, `MIN`, and `MAX` accept any data type.

Aggregations cannot be nested, e.g., the following is illegal:
```
AVG(COUNT(*))
```

Wrong!

Aggregations cannot be used in the `WHERE`-condition, only in the goal list after `SELECT` (but see `HAVING` below).

Wrong!

Example:意外 more chairs are placed in every room.
```
SELECT COUNT(SRNS) / COUNT(*) FROM COURSES
```

Wrong!

Another example: Enrolled students plus three guests.
```
SELECT COUNT(*) FROM ENROLLMENTS
```

Wrong!

Terms can also be used inside aggregations:
```
SELECT SUM(SEATS) / COUNT(*) FROM COURSES
```

Wrong!

Terms can also be used inside aggregations:
```
SELECT SUM(SEATS) / COUNT(*) FROM COURSES
```

Wrong!

```
SELECT COUNT(*) FROM COURSES
```

Wrong!

This is applied if a subquery is used and no normal attributes can appear in the `SELECT`-list. (But see `HAVING` below.)

Exercise / puzzle: is there any difference between `AVG(SEATS)` and `AVG(COUNT(SEATS))`? (But see `HAVING` below.)

Wrong!

The first counts all rows, whereas the second counts all rows where `INSTRUCTOR` is not null.

Wrong!

Otherwise, the actual attribute value is not important for other those rows where `INSTRUCTOR` is not null.

Wrong!

The difference between `COUNT(*)` and `COUNT(INSTRUCTOR)` is that the first counts all rows, whereas the second counts only those rows where `INSTRUCTOR` is not null.

Wrong!

Only a single output tuple is produced, but an attribute function is applied.

Wrong!

Aggregations cannot be used in the `WHERE`-condition, only in the goal list after `SELECT` (but see `HAVING` below).

Wrong!

Aggregations cannot be used in the `WHERE`-condition, only in the goal list after `SELECT` (but see `HAVING` below).

Wrong!

The `WHERE`-condition is evaluated before the aggregation.

Wrong!

```
WHERE SEATS < 400
```

Wrong!

```
WHERE SEATS < 400
```

Wrong!

```
WHERE SEATS < 400
```

Wrong!

```
WHERE SEATS < 400
```

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!

Wrong!
Aggregations and Null Values

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 empty input set results in 0, but in SQL it returns NULL.

One reason for this behavior might be that the SUM aggregation function cannot detect a difference between the empty input set and all qualifying tuples with a null value in this argument.

It may happen that no rows satisfy the WHERE condition, so programs must be prepared to process the resulting null value.

Course on Database Systems
Stefan Brass
GROUP BY (6)

GROUP BY (4)

GROUP BY (3)

GROUP BY (2)

GROUP BY (1)
The sequence of attributes in the GROUPBY clause is not important:
{
GROUPBY A, B
means that two tuples belong to the same group if:
A = u and B = v

GROUPBY B, A
means that two tuples belong to the same group if:
B = u and A = v

GROUPBY
is evaluated before the SELECT clause.

SELECT
FROM COURSES
GROUPBY FLOOR((SEATS-1)/25)+1
Wrong!

Oracle, SQL Server, and DB2 all allow
GROUPBY with arbitrary expressions, whereas the SQL-92 standard requires
column names after GROUPBY.

GROUP BY 1 -- meaning first column
Wrong!

Syntax:

SELECT ... FROM ... GROUPBY ...
Evaluating the query execution steps:

1. The SELECT clause is evaluated.
2. The GROUP BY clause is evaluated.
3. The WHERE clause is evaluated.
4. The HAVING clause is evaluated.
5. Evaluation of output tuples.

Syntax:

```
SELECT Goal
FROM Source
WHERE Condition
GROUP BY Grouping
HAVING Condition
```

Example query:

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

```
<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>123</td>
<td>CS101</td>
</tr>
<tr>
<td>David</td>
<td>456</td>
<td>CS102</td>
</tr>
<tr>
<td>Miller</td>
<td>789</td>
<td>CS103</td>
</tr>
</tbody>
</table>
```

Of course, no aggregation functions are allowed.

In Oracle, SQL Server, and DB2, "attribute-reference" would be replaced by "term" (which is more general).

```
SELECT GROUP BY TITLE, C.
```

```
Course on Database Systems Stefan Brass
```

```
HAVING: Elimination of Groups
```

```
Evaluation: Elimination of Groups
```

```
Syntax: SELECT-Query
```

```
HAVING COUNT(*) > 2
```

```
WHERE S.SSN = E.SSN
```

```
GROUP BY S.SSN, FIRST, LAST
```

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

```
<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>123</td>
<td>CS101</td>
</tr>
<tr>
<td>David</td>
<td>456</td>
<td>CS102</td>
</tr>
<tr>
<td>Miller</td>
<td>789</td>
<td>CS103</td>
</tr>
</tbody>
</table>
```
An aggregation is done if:

1. An aggregation function is used in the `SELECT` list, or
2. The `GROUP BY` or `HAVING` clause is present.

In this case:
- Only `GROUP BY` attributes can be used under `SELECT`.
- An aggregation function is used in the `SELECT` list or
- `HAVING` clause is present.

WHERE vs. HAVING

In most cases, the above restrictions uniquely determine whether a condition must be put under `WHERE` or `HAVING`.

WHERE:
- A condition can be used in both clauses.
- But it is much more efficient to put it under `WHERE`.

HAVING:
- A condition can only be used in both clauses.
- It is much more efficient to put it under `HAVING`.

Exercises

Consider a database containing invoices (bills):

- Customers (CustNo, Name, Street, City, State, ZIP, Phone)
- Invoices (InvoiceNo, Customer, Date, Paid, Amount)
- Courses (CourseNo, Name, Duration, Credits, Location)
- Enrollments (Customer, CourseNo, Grade, Date)

Create a database containing invoices (bills):

<table>
<thead>
<tr>
<th>Customer</th>
<th>Course</th>
<th>Grade</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>101</td>
<td>A</td>
<td>2023</td>
</tr>
<tr>
<td>456</td>
<td>102</td>
<td>B</td>
<td>2023</td>
</tr>
</tbody>
</table>

Print name and phone number of customers having at least 2,000 as the sum of all unpaid bills.

Print invoice number and total amount of invoices having at least 2,000 as the sum of all unpaid bills.

Print the total amount of unpaid bills for every customer.

Which is the sum of all outstanding bills (i.e., Paid='N')?

What is the range of invoice amounts (lowest and highest)?

Consider a database containing invoices (bills):
So far, no construct corresponding to set difference (\(-\)) in relational algebra was presented. So "not exists" and "for all" queries cannot yet be formulated in SQL.

With \(\exists\) and \(\forall\) it is possible to check whether an attribute value appears in a set that is computed by another SQL query.

E.g., print students that are not registered for any course:

\[
\begin{align*}
\text{RESULT OF SUBQUERY} & = \{ \text{SSN} \mid \text{SELECT SSN FROM ENROLLMENTS} \} \\
\text{STUDENTS} & = \{ \text{FIRST, LAST} \} \\
\end{align*}
\]

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

It is also possible to use \(\ni\) (without \(\not\)) for an element test. This is strictly a SQL extension, but it is not very portable. E.g., SQL Server does not support it.

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. An EXISTS subquery (see below) might be better.

In SQL-92, comparisons were extended to the tuple level. So that the subquery result is really a set (or multiset), and therefore it is possible to write e.g.:

\[
\begin{align*}
\text{RESULT OF SUBQUERY} & = \{ \text{SSN} \mid \text{SELECT SSN FROM ENROLLMENTS} \} \\
\text{STUDENTS} & = \{ \text{FIRST, LAST} \} \\
\end{align*}
\]

But it is not very portable. E.g., SQL Server does not support it.

Oracle and DB2 do allow \(\ni\) with multiple columns.

The column names on the left and right hand side of \(\ni\) do not have to match, but the data types must be compatible.
The subquery can be seen as parameterized.

The syntax of the subquery has to be evaluated once for each tuple assignment of values to the accessed tuple variables in the outer query. The result of the subquery is empty (NOT EXISTS), if the tuple assignment violates the condition of the WHERE clause of the outer query.

Therefore, the result for $S$ with $S\text{.SSN} = '000-00-3000'$ is empty.

However, the result for $S$ with $S\text{.SSN} = '999-99-9999'$ is empty.

The subquery is executed for every row of $S$.

The same happens for the second student.

Next, when $S$ points to the second row of STUDENTS, the subquery is executed for $S$ with $S\text{.SSN} = '777-77-7777'$.

The subquery is executed for $S$ with $S\text{.SSN} = '666-66-6666'$.

The result is not empty.

Next, when $S$ points to the third row of STUDENTS, the subquery is executed for $S$ with $S\text{.SSN} = '555-55-5555'$.

The subquery is executed for $S$ with $S\text{.SSN} = '444-44-4444'$.

The same happens for the fourth student.

However, the result for $S$ with $S\text{.SSN} = '333-33-3333'$ is empty.

Thus, the condition of the outer query is satisfied for this tuple. The name of the student is printed: Mary Jones.
While in the inner query, tuple variables from the outer query can be accessed, the converse is forbidden:

```
SELECT FIRST, LAST, E SSN
FROM STUDENTS
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 sub query are valid only in the sub query.

It works like the block structure of a Pascal:

```
WHERE SSN = 'S.SSN'
```

Non-correlated subqueries with NOT EXISTS are always be false.

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

Subqueries which do not access variables from the outer query:

```
WHERE NOT EXISTS (SELECT * FROM ENROLLMENTS E
                  WHERE SSN = 'S.SSN')
```

Here the JOIN condition in the sub query was forgotten, and it matters for which student (since there is no JOIN condition):

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

If there is at least one row in the ENROLLMENTS table, no tuple variable was found, and it became a non-correlated subquery.

It works like the block structure of a Pascal:

```
WHERE SSN = 'S.SSN'
```

Non-correlated subqueries with NOT EXISTS are almost:

```
WHERE NOT EXISTS (SELECT * FROM ENROLLMENTS E
                  WHERE SSN = 'S.SSN')
```

Subqueries with variables in the outer query are called "non-correlated subqueries".

Subqueries which do not access variables from the outer query:

```
WHERE NOT EXISTS (SELECT * FROM ENROLLMENTS E
                  WHERE SSN = 'S.SSN')
```

The subquery was executed once for every assignment of tuples to the tuple variables defined with the tuples chosen in the outer query.

Correlated subqueries can be understood as being called "correlated subqueries".

Subqueries which access variables from the outer query are almost:

```
WHERE NOT EXISTS (SELECT * FROM ENROLLMENTS E
                  WHERE SSN = 'S.SSN')
```

These are called "non-correlated subqueries".

Subqueries which do not access variables from the outer query:

```
WHERE NOT EXISTS (SELECT * FROM ENROLLMENTS E
                  WHERE SSN = 'S.SSN')
```

The subquery was executed once for every assignment of tuples to the tuple variables defined with the tuples chosen in the outer query.

Correlated subqueries can be understood as being called "correlated subqueries".

Subqueries with variables in the outer query are called "non-correlated subqueries".

Subqueries which do not access variables from the outer query:

```
WHERE NOT EXISTS (SELECT * FROM ENROLLMENTS E
                  WHERE SSN = 'S.SSN')
```

The subquery was executed once for every assignment of tuples to the tuple variables defined with the tuples chosen in the outer query.

Correlated subqueries can be understood as being called "correlated subqueries".

Subqueries which access variables from the outer query are almost:
This rule helps that non-correlated subqueries can be developed independently and inserted into another query without any change (so it makes sense).

It is also legal to declare tuple variables in the subquery which have the same name as tuple variables in the subquery which

I.e., a course $X$ is selected if there is no course $Y$ with a strictly smaller number of places.

Since for `NOT EXISTS` the returned columns do not matter, `SELECT *` should always be used in the subquery.

Subqueries must be enclosed in parentheses: (...).

ORDER BY is not allowed in subqueries.

It would make no sense there, i.e., only for the final output.

`EXISTS` without negation.

Some authors say that in some systems `SELECT` must or `SELECT *` should always be used in the subquery.

This rule helps that non-correlated subqueries can be developed independently and inserted into another query without any change (so it makes sense).

The variable declared in the outer query becomes `shadowed` and cannot be accessed in the subquery.

Multiple columns in the `FROM` clause of the subquery.

In the subquery, only those columns that are mentioned in the subquery can be used. If a column is mentioned in the subquery which does not appear in the `FROM` clause of the subquery, then the subquery will simply contain a `WHERE` clause with `EXISTS`.

It is also legal to declare tuple variables in the subquery which have the same name as tuple variables in the subquery which

Since for `NOT EXISTS` the returned columns do not matter, `SELECT *` should always be used in the subquery.

Subqueries must be enclosed in parentheses: (...).

ORDER BY is not allowed in subqueries.

It would make no sense there, i.e., only for the final output.

`EXISTS` without negation.

Some authors say that in some systems `SELECT` must or `SELECT *` should always be used in the subquery.

This rule helps that non-correlated subqueries can be developed independently and inserted into another query without any change (so it makes sense).
Common Errors:

Exercise 1:

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 AND E.CRN IS NULL
```

Exercise 2:

What about this query?

```
WHERE S.SSN > E.SSN
```

Moreover, Any and Some are synonyms.

- The condition `NOT SEATS < ANY (SELECT SEATS FROM COURSES)` would give the result `WHERE SEATS -> ALL (SELECT SEATS FROM COURSES)`.
- Which class has the minimal number of seats (again)?
- When class has the maximal number of seats (again)?

Another common error is to use a non-correlated subquery with `NOT EXISTS` for all of its rows.

- The common errors for all of its rows.
- Common errors when subqueries are nested, but without subquery
- Common errors when subqueries are used in outer join. Aggregations also change when tuples are inserted, but without subquery.
- There is no way to write it without a subquery.
- If the request works in a non-monotonic fashion of a row is very different then the existence of a row with a different value.

- It is important to understand that the absence/non-existence

Common Errors (1):

```
ALL, ANY, SOME (1)
```

Axiom Formula (Form 8):

```
Comparison Op. \rightarrow (Term) \rightarrow (Term)
```

Common Errors (2):

```
ALL, ANY, SOME (2)
```

Moreover, Any and Some are synonyms.

- The condition `NOT SEATS > ALL (SELECT SEATS FROM COURSES)` would give the result `WHERE SEATS -> ALL (SELECT SEATS FROM COURSES)`.
- Which class has the minimal number of seats (again)?
- When class has the maximal number of seats (again)?

- It is possible to compare a value with all values in a set.
Subqueries in the FROM-Clause (2)

Subqueries in the FROM-Clause (1)

Subqueries Returning Data Elements

The feature is needed e.g. for „nested aggregation“.

What is the average number of students registered for a course?

SELECT AVG(X.NUM) FROM ENROLLMENTS GROUP BY CRN

What class has the minimal number of seats?

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

Subquery must yield exactly one row (i.e. a single data element)

If none of the keywords ALL, ANY, SOME are present, the feature is needed.

A view is simply a shorthand (macro), the name can be replaced by a subquery instead of a table name in the FROM-clause.

If a view is the result of an SQL-query it can be used in the FROM-clause.

SELECT AVG(X.NUM) FROM ENROLLMENTS GROUP BY CRN

Subquery Returning Data Elements

Subqueries Returning Data Elements

\( \text{AVG}(X.\text{NUM}) \)
Subqueries

Subqueries in the FROM-Claus

5. SQL-92: JOIN in Oracle

4. UNION, ORDER BY

3. Subqueries

2. Aggregations: GROUP BY, HAVING

1. Aggregations: Aggregation Functions

Overview

Exercises

What is the average amount sold to a customer (sum of bills)?

Having the highest bill (paid or unpaid)?

Print the complete customer record of the customer(s) that have not paid any unpaid bill?

Please formulate the following queries in SQL:

Invoice(Invoice No., Customer ID, Date, Paid, Amount)

Customer(Customer ID, Name, Street, City, State, Zip, Phone)

Consider again the database containing invoices ( bills):
In SQL it is possible to combine the results of two queries by using the `UNION` operator. `UNION` combines the results of two queries into a single result set.

The sub-queries which are operated to `UNION` must return table with the same number of columns and corresponding column names must not be equal. Oracle and SQL Server use the attribute names from the first operand table. DB2 uses artificial column names if the input column names do not agree. `UNION` is necessary if a column of the output table should contain values from different input columns.

`UNION` is also very useful for case analysis (e.g., if else...). `UNION ALL` is used if the input column names do not agree. SQL Server uses the attribute names from the first operand table. The attribute names do not have to be equal. `UNION ALL` is used for concatenation (no duplicate elimination).

Duplicate elimination is quite expensive.

- `UNION ALL` with duplicate elimination (no duplicate elimination).
- `UNION` with duplicate elimination.
- `SQL` distinguishes between different tables/columns.
- `UNION` is needed since otherwise there is no way to construct one result column which contains values drawn from two or more different tables.
- `UNION` is possible to combine the results of two queries by.

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

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

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 Algebra Operations in SQL

SQL Server and SQL/6/6 allow only UNION // UNION ALL.
The SQL/9/2 standard also allows EXCEPT (set difference, -) and INTERSECT (\, /\).
These operators are supported in DB2.

In Oracle/8/, the operator is called MINUS instead of EXCEPT.

These operations add nothing to the expressivity of the language.

**Query**: To get a sort value, e.g. the enrollment of students,

**Columns from the query results**

The sequence of output rows means nothing (it depends on the algorithms used in the DBMS).

One can only sort by columns appearing in the output.

Querying produces more than very few rows should always

- Print the course sorted by instructor, and for each
- Print the courses sorted by title; and for each
- Print the class sorted by course, and for each
- Print the course sorted by instructor, and for each

Other Algebra Operations in SQL
Sorting Output

The `ORDER BY` list can contain more than one column. These second columns are only used for ordering tuples which have the same value in the first column, and so on.

Note that the effect of `ORDER BY` is purely cosmetic. It does not change the set of output tuples in any way. The second column is only used for ordering two tuples which have the same value in the first column, and so on.

The `ORDER BY` list can contain more than one column.

`DESC` means descending (inverse order from high to low).

`ASC` means ascending (from low to high).

Therefore, `ORDER BY` can only be applied at the very end of the query. It cannot be used in subqueries.

In the query, any column name or column number can be used in the `ORDER BY` list. For example, `ORDER BY column1, column2` can only be used in subqueries if the query name associated with `ORDER BY` can only be applied at the very end of the query. It cannot be used in subqueries.
Joins in SQL (92) (4)

Joins in SQL (92) (3)

Joins in SQL (92) (1)
Joins in SQL

- Both DB2 and SQL Server require the use of ON.
- SQL Server does not support CROSS JOIN, but not UNION JOIN.
- SQL Server does not support CROSS JOIN and UNION JOIN.
- One has to append "(+" to attributes of the table which can be replaced with null.

Outer Join in Oracle

In Oracle, one has to append "(+" to attributes of the table which can be replaced with null.

The condition can also be written: "C.CRN = E.CRN - C.CRN".

GROUP BY C.CRN
WHERE C.CRN - E.CRN
FROM COURSES C, ENROLLMENTS E
SELECT C.CRN, TITLE, COUNT(SSN)

- Are many syntactic restrictions.
- The marking is only done in the WHERE-Condition. There are many syntactic restrictions.
- You would simply write a common TO OUTER JOIN, so it is not very useful. The marking is only done in the WHERE-Condition.
- You would simply write a common TO OUTER JOIN, so it is not very useful. The marking is only done in the WHERE-Condition.

- In Oracle, one has to append "(+" to attributes of the table which can be replaced with null.