Part 10: 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 Edition, McGraw-Hill, 1999: Chapter 4: "SQL".
- Kemper/Eickler: Datenbanksysteme (in German), Ch. 4, Oldenbourg, 1997.
- 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, there is an English version), Hanser, 1990.
- Sunderraman: Oracle Programming, A Primer. Addison-Wesley, 1999.
- Oracle8 SQL Reference, Oracle Corporation, 1997, Part No. A58225-01.
- Don 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 syntactical correctness of a given query.
- evaluate the portability of certain constructs.

Overview

Aggregations (1)

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

  E.g.: $\min\{41, 57, 19, 23, 27\} = 19$
- Typical aggregation functions are:
  number (count), sum, average, minimum and maximum.
- Aggregation functions can be used for statistical evaluations.
- Some aggregation functions are sensitive to duplicates.
  Then one has to decide whether to use duplicate elimination (input is a set) or not (input is a multiset).

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

Aggregations (2)

- 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.g. $+ \longrightarrow$ sum.
  - The conditions are necessary since a set has no order.
- Aggregation Functions SQL-86/92: `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`.

Example Database (1)

STUDENTS

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

ENROLLMENTS

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

Example Database (2)

COURSES

CRN	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 course 41590?

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

COUNT(*)
2

Simple Aggregations (2)

- How many students are enrolled in at least one course?

```
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)

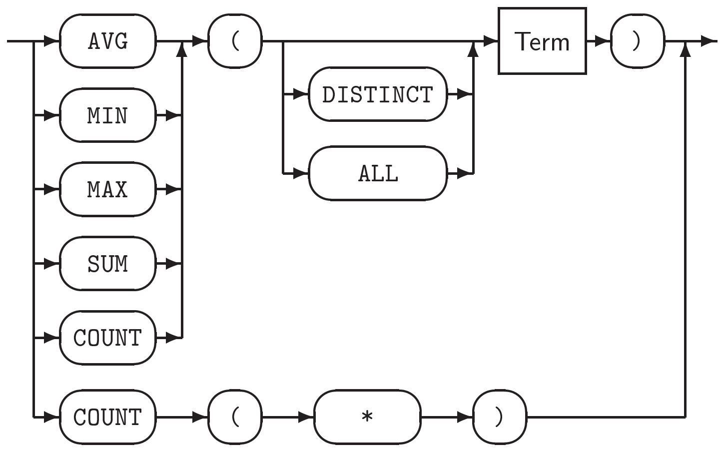
- The whole power of SQL can be used to compute the set of values which is aggregated. (E.g. joins.)
- E.g.: How many students are enrolled in courses by Brass?

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

- Multiple aggregations can be computed in the SELECT list.
- What is the minimum and maximum number of seats?

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


Syntax: Aggregation Terms



Evaluation

- First, the FROM-clause is evaluated by computing the cartesian product of the source tables.
- Second, the WHERE-clause is evaluated by a selection which filters those tuple combinations which satisfy the condition.

  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.
  - Usually, this is a projection.
  - But when the SELECT-list contains an aggregation term, only a single output row is computed by applying the aggregation operators (set of values $\rightarrow$ single value).

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 first aggregation only a single value remains.

  Thus, applying another aggregation would make no sense.
- Aggregations cannot be used in the WHERE-condition, only in the goal list after SELECT (but see HAVING below).

  `WHERE COUNT(*) > 1` **Wrong!**

  The WHERE-condition is evaluated before aggregations are computed (it determines which tuples enter the aggregation).

Syntax / Restrictions (2)

- If an aggregation function is used, no normal attributes can appear in the SELECT-list. (But see GROUP BY below.)

  Only a single output tuple is produced, but an attribute outside aggregations would not have a unique output value.

  ```
  SELECT CRN, COUNT(SSN)   Wrong!
  FROM   ENROLLMENT
  ```
- Every aggregation operator needs an argument (input).

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

  A subquery is required to find the course with minimal seats.

Syntax / Restrictions (3)

- 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
  ```

  Another example: Enrolled students plus three guests.

  ```
  SELECT COUNT(*) + 3
  FROM   ENROLLMENTS
  WHERE  CRN = 41590
  ```
- Terms can also be used inside aggregations: (E.g. capacity if 5 more chairs are placed in every room.)

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


Aggregations and Null Values (1)

- Usually, null values are ignored (filtered out) before the aggregation function is applied.
- Only COUNT(*) includes null values.
- So 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.

  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.
- Exercise/Puzzle: Is there any difference between AVG(SEATS) and SUM(SEATS)/COUNT(*) (if SEATS can be null)?

Aggregations and Null Values (2)

- 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.)

- It may happen that no row satisfies the `WHERE`-condition, so 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.
- Compute the number of enrolled students for every course.

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

CRN	COUNT(*)
41560	2
31864	2
22332	1

- E.g. currently 2 students are enrolled in course 41590.

GROUP BY (2)

- The `GROUP BY` clause splits the resulting table after evaluation of `FROM` ($\times$) and `WHERE` (σ) into groups which 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)

- Note that the 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 only be used inside aggregation functions in the SELECT-list.

- Since Title does not appear under GROUP BY, this is illegal:

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

- This is especially strange since CRN is a key of COURSES, so that TITLE is actually unique in the groups.

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	#
41560	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 allow `GROUP BY` with arbitrary expressions, whereas the SQL92 standard requires column names after `GROUP BY` (needs subquery under `FROM`):

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

Not SQL-92!

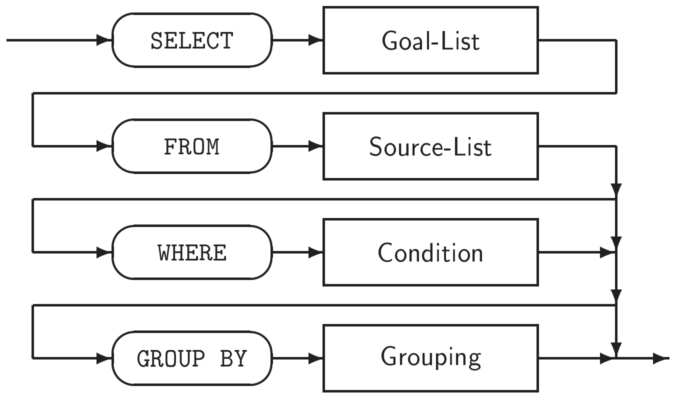
GROUP BY (9)

Warning:

- Many students mix up "`GROUP BY`" and "`ORDER BY`":
  - `GROUP BY` is important for the query result.
  - `ORDER BY` is only cosmetic (how the result is printed).
- It is true that `GROUP BY` normally requires sorting the tuples (so that tuples with the same values are adjacent).
- But after the sort, the `GROUP BY` does the grouping, whereas the sort for the `ORDER BY` is done at the very end.
- Also, the DBMS might sometimes evaluate the `GROUP BY` in more efficient ways without sorting.

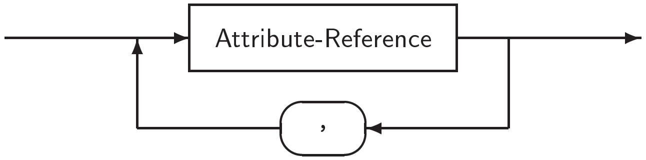
  E.g. if the data is stored clustered by the grouping columns.

Syntax (1)



Syntax (2)

Grouping:



- E.g. GROUP BY TITLE, C.CRN
- In Oracle, SQL Server, and DB2 "Attribute-Reference" would be replaced by "Term" (which is more general).
  Of course, no aggregation functions are allowed.

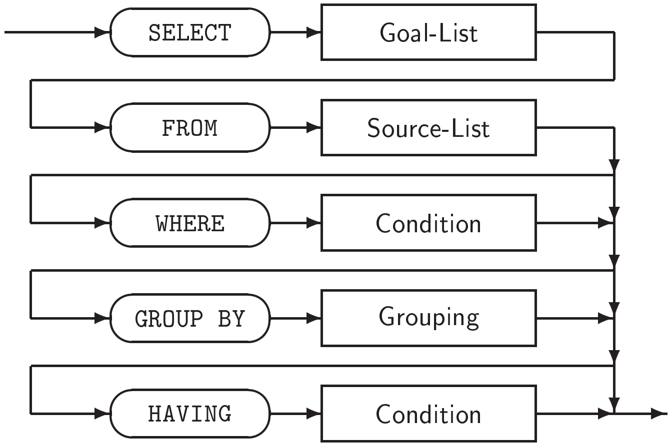
HAVING: Elimination of Groups

- Aggregations cannot be used in the WHERE-condition.
  But sometimes aggregation operators are needed to filter output rows, not only for computing output values.
- Aggregation operators can be used in the HAVING-condition.
  The HAVING-condition is for eliminating whole groups.
- 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

Syntax: SELECT-Query



Evaluation

- First, the cartesian product of the relations mentioned in the FROM-clause is constructed.
- Second, those combinations of tuples are selected which satisfy the condition in the WHERE-clause.
- Third, the remaining joined tuples are split into groups having equal values for the GROUP BY-attributes.
- Fourth, groups of tuples which do not satisfy the condition in the HAVING-clause are eliminated.
- Fifth, one output tuple for every group is produced by evaluating the terms in the SELECT-clause.

Syntax: Restrictions

- An aggregation is done if
  - an aggregation function is used in the SELECT-list, or
  - the GROUP BY or HAVING-clause is present.
- In this case: 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: A inside, B outside.
- The WHERE-condition can never contain aggregations.
  (Except in subqueries, see below.)
- A HAVING-clause without a GROUP BY-clause is legal, but very uncommon: It could return only 0 or 1 output rows.

WHERE vs. HAVING

- In most cases, the above restrictions uniquely define whether a condition must be put under WHERE or under HAVING.
- If a condition contains only GROUP BY-attributes, but no aggregations, it would be allowed in both clauses.
- But 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

Consider a database containing invoices (bills):

Customers(<u>CustNo</u>, Name, Street, City, State, ZIP, Phone)

Invoices(<u>No</u>, CustNo → Customers, Date, Paid, Amount)

Please formulate the following queries in SQL:

- What is the range of invoice amounts (lowest and highest)?
- What is the sum of all outstanding bills (i.e. Paid='N')?
- Print the total amount of all unpaid bills for every customer (output customer number and customer 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

NOT IN (1)

- 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 `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. $R - S = \{t \in R \mid t \notin S\}$.
- E.g. print 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

SSN	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 `STUDENT` tuple, a matching `SSN` is searched in the subquery result. If there is none, the student 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).
- E.g. students that are registered for at least one course:

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


NOT IN (4)

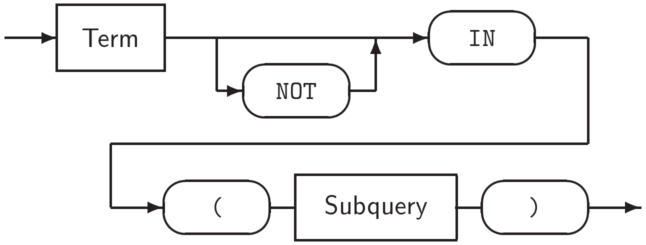
- In SQL-86, the subquery must have a single output column.
  So that the subquery result is really a set (or multiset), and not an arbitrary relation.
- In SQL-92, comparisons were extended to the tuple level, and therefore it is possible to write e.g.

```
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.
- 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 IN (5)

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`$_1$`, ..., Term`$_n$). Then the subquery must result in a table with exactly n columns.

NOT EXISTS (1)

- It is also possible to check in the outer query whether the result of the subquery is empty (`NOT EXISTS`).
- R − S are all tuples from R, such that the same tuple does not exist in S. "For all" is the same as "not exists not".
- 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. E.g. for `S.SSN` = `'123-45-6789'` the following is executed:

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

SSN	CRN
123-45-6789	41590

- The result is not empty. So nothing is printed for this student.

NOT EXISTS (3)

- Next, when `S` points to the second row of `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 same happens for the third student.
- However, the result for `S.SSN` = `'900-50-3000'` is empty:

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

no rows selected

- Thus, the condition of the main query is satisfied for this tuple `S`. The name of the student is printed: `Mary Jones`.

NOT EXISTS (4)

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

```
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.

  It works like the block structure of e.g. Pascal.

NOT EXISTS (5)

- Subqueries which 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 which 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 (6)

- Non-correlated subqueries with `NOT EXISTS` are almost always an error (but they do make sense with `IN`, see above):

```
SELECT FIRST, LAST
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 (7)

- Until now, for every unqualified attribute name (i.e. attribute name without tuple variable), there had to be a unique tuple varible to which it can refer.
- For subqueries, the proximity of the tuple variable declaration is used to disambiguate the attribute reference.
- 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 (8)

- 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 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” and cannot be accessed in the subquery.

NOT EXISTS (9)

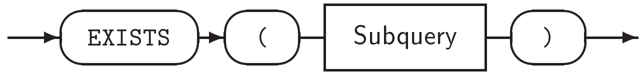
- What is the smallest class size (number of seats)?

```
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 always be used in the subquery.
  Some authors say that in some systems SELECT null or SELECT 1 is actually faster than SELECT *.

NOT EXISTS (10)

Atomic Formula (Form 7):



- A subquery is an expression of the form SELECT ... FROM ... [WHERE ...] [GROUP BY ...] [HAVING ...].
- 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 (11)

- 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".
- Another common error is to use a non-correlated subquery with `NOT EXISTS` (forgetting the join condition). See above.

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 (again)?

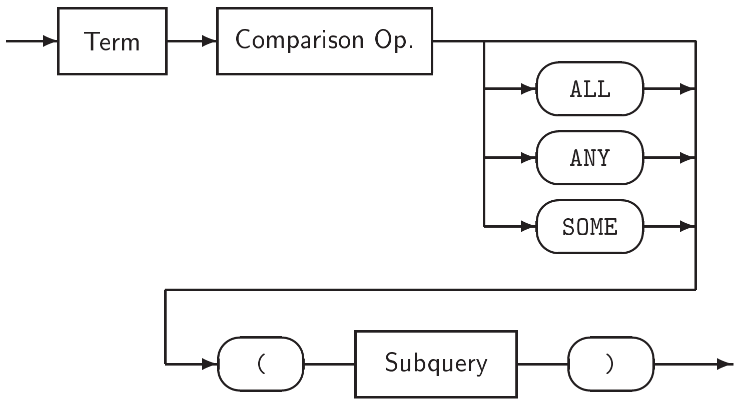
```
SELECT CRN, TITLE
FROM   COURSES
WHERE  SEATS <= ALL (SELECT SEATS
                     FROM COURSES)
```

- The Condition "`NOT SEATS > ANY (...)`" would give the same result (logically equivalent).

  Moreover, `ANY` and `SOME` are synonyms.

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 <> 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).

Subqueries Returning Data Elements

- 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. For instance, the symmetric version is equally legal:

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

- But Oracle 8.0 and SQL86 do not allow this.

Subqueries in the FROM-Clause (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), 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 a shorthand (macro), its name can be replaced by its definition in the `FROM`-clause.

Subqueries in the FROM-Clause (2)

- This feature is needed e.g. for a "nested aggregation".
- What is the average number of students registered for a course? (Count only courses with at least one student.)

```
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

Subqueries in the FROM-Clause (3)

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

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

- Oracle also supports nested aggregations in this way:

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

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

Subqueries in the SELECT-Clause

- 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.
- This makes the GROUP BY clause unnecessary.
- E.g. print for every course the number of enrolled students (including courses with 0 pieces):

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

- This does not work in Oracle 8.0.

Exercises

Consider again the database containing invoices (bills):

Customers(<u>CustNo</u>, Name, Street, City, State, ZIP, Phone)

Invoices(<u>No</u>, CustNo → Customers, Date, Paid, Amount)

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 ∪ 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 which contains values drawn from different tables/columns.
- SQL distinguishes between
  - `UNION`: ∪ with duplicate elimination, and
  - `UNION ALL`: ∪/concatenation (no duplicate elimination).

  Duplicate elimination is quite expensive.

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 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 (to code an **if . . . then . . . else . . .**).

  See also conditional expressions in Appendix H.

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 Algebra Operations in SQL

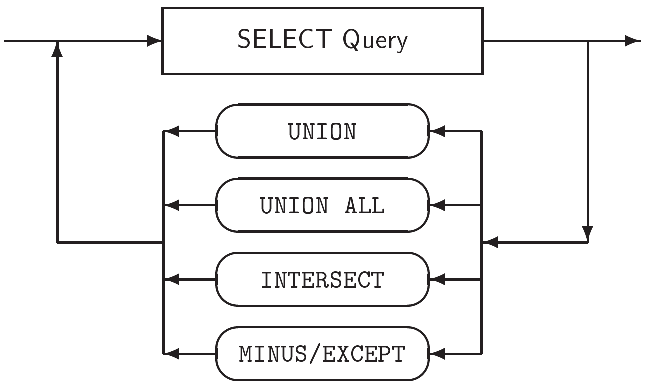
- SQL Server and SQL86 allow only `UNION`/`UNION ALL`.
- The SQL92 standard also allows `EXCEPT` (set difference, $-$) and `INTERSECT` ($\cap$). These operators are supported in DB2.
- In Oracle8, the operator is called `MINUS` instead of `EXCEPT`.

  `ALL` for `MINUS` and `INTERSECT` is not supported.
- These operations add nothing to the expressivity to the language.

  Queries containing `MINUS` and `INTERSECT` can be transformed into equivalent SQL-queries without these constructs, but queries containing `UNION` in general cannot. So `UNION` is needed for the relational completeness of SQL.

Algebra Operations in SQL: Syntax

Table Expression:



Sorting Output (1)

- 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
```

CRN	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

Sorting Output (2)

- Queries producing more than very few rows should always include an `ORDER BY` clause, since 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).
- One can only sort by columns appearing in the output.

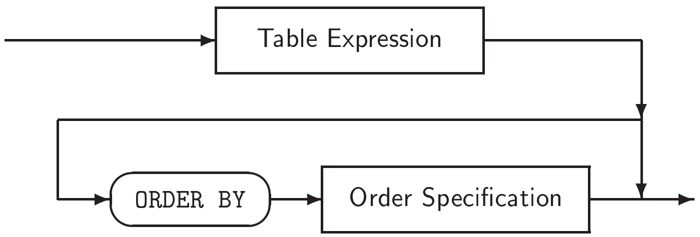
  E.g. it is impossible to sort by the seat number without printing it. But tools like SQL*Plus can suppress output columns from the query result.
- Sometimes it is necessary to add columns to database tables to get a sort value, e.g. if the "University of Pittsburgh" should appear in a sorted sequence under "P".

Sorting Output (3)

- "DESC" means descending (inverse order from high to low values), the default is "ASC" (ascending).
- The "ORDER BY" list can contain more than one column. The second column is only used for ordering two 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.
- Therefore, "ORDER BY" can only be applied at the very end of the query. It cannot be used in subqueries.
- Column name or column number can be used in the "ORDER BY" list.

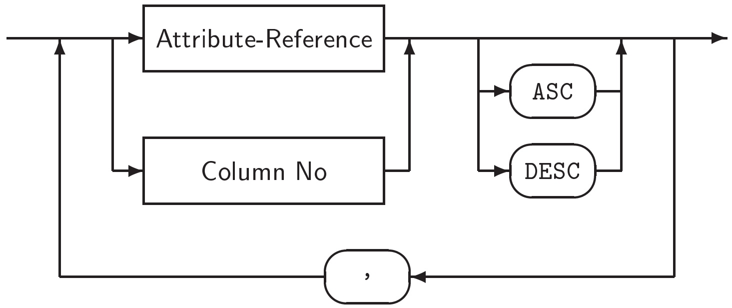
Sorting Output (4)

SQL Query:



Sorting Output (5)

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)

- SQL-92 allows joins in the FROM-list.

  And also like UNION in the outer query level.
- SQL86 and Oracle8 do not allow this, DB2 and SQL Server support part of the standard.
- These constructs are not needed, one can transform a query using them into an equivalent query without them.

  However, the outer join is useful, since an equivalent query with UNION becomes complicated.
- E.g. list enrollments with student names:

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

This query is actually not supported in any of the three DBMS.

Joins in SQL-92 (2)

- In DB2 and SQL Server, the join condition must be specified:

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

- Useful is an outer join which ensures that every input tuple from one table is represented in the result.
- E.g. number of students per course (including 0):

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


Joins in SQL-92 (3)

- SQL-92 has the following join types:
  - [INNER] JOIN: Usual Join.
  - LEFT [OUTER] JOIN: Left table tuples are preserved.
  - RIGHT [OUTER] JOIN: Right table tuples are preserved.
  - FULL [OUTER] JOIN: All input tuples are preserved.
  - CROSS JOIN: Cross product $\times$
  - UNION JOIN: This is a union which fills the columns of the other table with null values.

Joins in SQL-92 (4)

- The join condition (except for CROSS JOIN / UNION JOIN) is specified as follows:
  - The join name is preceded by the keyword NATURAL.
  - "ON ⟨Condition⟩" follows the join.
  - "USING ($A_1, \ldots, A_n$)" follows the join.

  Only one of these constructs can be used.
- The USING clause lists common attributes which should be used for the join:

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

Joins in SQL-92 (5)

- 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 support `CROSS JOIN`, but not `UNION JOIN`.

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

- The condition can also be written: "`E.CRN (+) = C.CRN`".