Part 10: SQL II

Elmasri/Navathe:

- Chap. 8, “SQL — The Relational Database Standard”

  In this part, we treat only Section 8.3.

I have also used the following sources:

- 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

- You should be able to write advanced queries in SQL including
  - Several tuple variables over the same relation.
  - Aggregations, GROUP BY, HAVING.
  - NOT EXISTS / NOT IN ($\sim$ set difference).
- You should at least know all constructs from the SQL-86 standard. You should be able to evaluate the syntactical correctness of a given query.
- You should know a bit about the SQL-92 standard, and more or less be able to 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).

  E.g.: $\text{min}\{41, 57, 19, 23, 27\} \;=\; 19$
- Typical aggregation functions are:
  number (count), sum, average, minimum and maximum.
- So aggregation functions can be used for statistical evaluations.
- Sometimes, aggregations have to be done on multisets (sets with multiplicity). E.g. the sum of all items of an invoice. If two items cost the same amount, it has to be counted twice.

  E.g.: $\text{sum}\{\{1, 3, 3, 7\}\} \;=\; 14$

Aggregations (2)

- Aggregations increase the expressivity of a query language. While minimum and maximum can already 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.

- Standard SQL has the five aggregation functions COUNT, SUM, AVG, MAX, MIN.

  Additional aggregation functions:
  Oracle and DB2: VARIANCE, STDDEV (DB2 also: COUNT_BIG).
  SQL Server: VAR, VARP, STDEV, STDEVP.

Example Database

Composer			
CNo	CName	Born	Died
1	Vivaldi	1678	1741
2	Bach	1685	1750
3	Mozart	1756	1791
4	Prokofiev	1891	1953
5	Dvorak	1841	1904

Piece		
PNo	CNo	PName
1000	1	The Four Seasons
1002	2	B-minor Mass
1003	2	Christmas Oratorio
1004	3	Missa solemnis
1006	4	Classical Symphony
1007	4	Cinderella

Simple Aggregations (1)

- Which composer was born first? Print only his birth year.

```
SELECT MIN(Born)
FROM   Composer
```

MIN(Born)
1678

- How many pieces by “Bach” (CNo=2) are in the database?

```
SELECT COUNT(*)
FROM   Piece
WHERE  CNo = 2
```

COUNT(*)
2

Simple Aggregations (2)

- How many composers have pieces stored in the DB?

```
SELECT COUNT(DISTINCT CNo)
FROM   Piece
```

COUNT(DISTINCT CNo)
4

- What is the average lifespan of the composers in the DB?

```
SELECT AVG(Died - Born) "Average Lifespan"
FROM   Composer
```

Average Lifespan
57.6

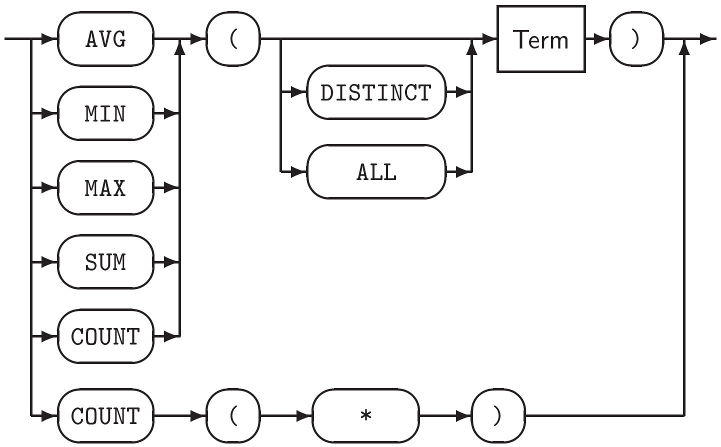
Simple Aggregations (3)

- What is the earliest birth year, number, and latest death year of a composer in the DB who wrote a "Symphony"?

```
SELECT MIN(Born), COUNT(DISTINCT CNo), Max(Died)
FROM   Composer, Piece
WHERE  Composer.CNo = Piece.CNo
AND    (PName LIKE '%Symphon%'
        OR PName LIKE '%Sinfon%')
```

- So you can use the whole power of SQL to compute the set of values which is aggregated.
- You can have multiple aggregations in the SELECT list.

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 the aggregations are computed (it determines which tuples enter the aggregation).

Syntax / Restrictions (2)

- If you use an aggregation function, you cannot have a normal attribute in the SELECT-list, because in this case only a single output tuple is produced, and this attribute would not have a unique value. (But see GROUP BY below.)

```
SELECT CNo, COUNT(*)    Wrong!
FROM   Piece
```

- Every aggregation operator needs an argument (input).

```
SELECT *
FROM   Composer
WHERE  Born = MIN    Wrong! Wrong!
```

You need a subquery to find the earliest born compostor.

Syntax / Restrictions (3)

- An aggregation term like SUM(Born) is by itself a term (an expression), so you can write e.g.

  SUM(Born) / COUNT(Born).

- You also can use terms inside aggregations:

  AVG(Died - Born).

- You only cannot nest aggregations.

Aggregations and Null Values (1)

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

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

- Exercise/Puzzle: Is there any difference between SUM(DIED-BORN)/COUNT(*) and AVG(DIED-BORN)?

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. It may happen that no row satisfies the WHERE-condition, so you must be prepared to process a null value for the result (or use e.g. NVL in Oracle to replace it).

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)

- Until now, we could compute 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 music pieces for every composer.

```
SELECT    CNo, COUNT(*)
FROM      Piece
GROUP BY  CNo
```

CNo	COUNT(*)
1	1
2	2
3	1
4	2

- E.g. one piece of Vivaldi (CNo=1) is in the database.

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.

PNo	CNo	PName
1000	1	The Four Seasons
1002	2	B-minor Mass
1003	2	Christmas Oratorio
1004	3	Missa solemnis
1006	4	Classical Symphony
1007	4	Cinderella

GROUP BY (3)

- The aggregation is then done over every group. So we get one output row for every group.
- Note that the construction can never produce empty groups. So it is impossible that a `COUNT(*)` results in the value 0.

  You might get the value 0 with `COUNT(A)` where the attribute A can be null. If your query must produce groups with count 0, you probably need an outer join (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 CName does not appear under GROUP BY, this is illegal:

```
SELECT   CName, COUNT(*)  Wrong!
FROM     Composer, Piece
WHERE    Composer.CNo = Piece.CNo
GROUP BY Composer.CNo
```

- This is especially strange since CNo is a key of Composer, so that CName is actually unique in the groups.

GROUP BY (5)

- So you have to group by CName in this case:

```
SELECT   CName, COUNT(*) "#"
FROM     Composer C, Piece P
WHERE    C.CNo = P.CNo
GROUP BY CName
```

CName	#
Vivaldi	1
Bach	2
Mozart	1
Prokofiev	2

- If there can be composers with the same name, you must list CNo and CName after GROUP BY:

```
SELECT   CName, COUNT(*)
...
GROUP BY C.CNo, CName
```


GROUP BY (6)

- Note that 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. So you cannot refer to new attribute names:

```
SELECT   CNO "Composer", COUNT(*) "Num Pieces"
FROM     PIECE
GROUP BY "Composer"    Wrong!
```


GROUP BY (7)

- This means that you also cannot group by values which you compute in the SELECT-clause:

```
SELECT   FLOOR(BORN/100)+1 "Century", COUNT(*)
FROM     PIECE
GROUP BY 1 -- meaning first column     Wrong!
```

- Oracle, SQL Server, and DB2 all allow to group by arbitrary expressions, whereas the SQL92 standard requires column names after GROUP BY (would need subquery under FROM):

```
SELECT   FLOOR(BORN/100)+1 "Century", COUNT(*)
FROM     PIECE
GROUP BY FLOOR(BORN/100)+1
```

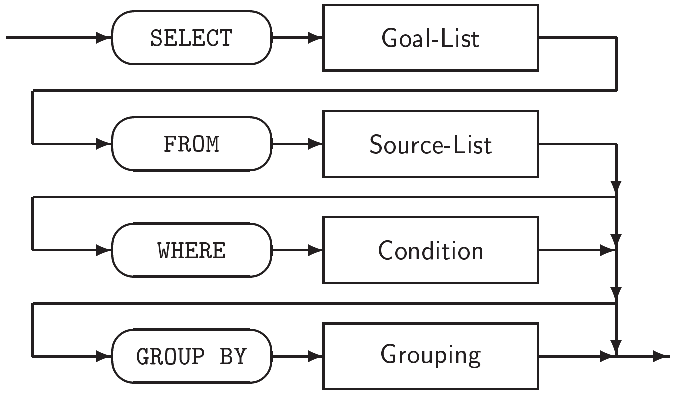
GROUP BY (8)

Warning:

- Many students mix up “GROUP BY” and “ORDER BY”:
  - GROUP BY is important for the query result.
  - ORDER BY is only cosmetics (how the result is printed).
- It is true that GROUP BY normally requires to sort 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.

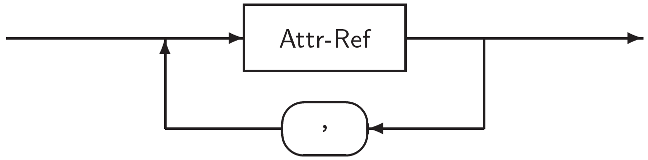
  E.g. if the data is stored clustered by the grouping columns.

Syntax (1)



Syntax (2)

Grouping:



- E.g. GROUP BY CName, C.CNo
- In Oracle, SQL Server, and DB2 “Attr-Ref” 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.

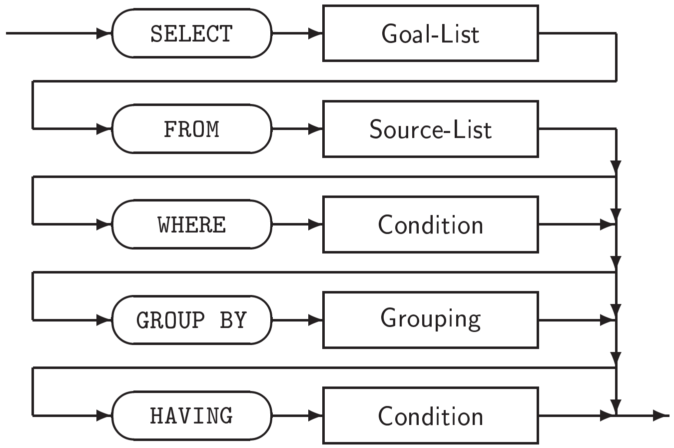
  But sometimes we want to use aggregation operators to filter output rows, not only for computing output values.
- Aggregation operators can be used in the HAVING-condition.

  The HAVING-condition is for eliminating whole groups.
- Which composers have written at least two pieces?

```
SELECT   CName
FROM     Composer C, Piece P
WHERE    C.CNo = P.CNo
GROUP BY CName
HAVING   COUNT(*) >= 2
```

CName
Bach
Prokofiev

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 are eliminated which do not satisfy the condition in the `HAVING`-clause.
- 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.
- If an aggregation is done, then: All attributes used outside aggregation functions in the `SELECT`-list or `HAVING`-clause must also appear under `GROUP BY`.

  `AVG(A)/B`: `B` is used outside an aggregation function.
- 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    CName
FROM      Composer C, Piece P
GROUP BY  CName, C.CNo, P.CNo
HAVING    C.CNo = P.CNo AND COUNT(*) >= 2
```

Exercises

Consider a database containing invoices (bills):

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

Invoice(<u>No</u>, CustNo → Customer, 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 EXISTS (1)

- Until now, we have no SQL construct corresponding to the set difference (−) in relational algebra (or "for all" in tuple calculus).

  Sometimes it might be possible to simulate this with aggregations, but this can be complicated.
- SQL queries can contain other SQL queries and especially you can check in the outer query whether the result of the inner query 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, you can access the tuple variable declared in the `FROM`-clause of the outer query.

NOT EXISTS (2)

- Which composers do not have any piece in the DB?

```
SELECT CName
FROM   Composer C
WHERE  NOT EXISTS (SELECT * FROM Piece P
                   WHERE P.CNo = C.CNo)
```

- So the tuple variable `C` loops over the five rows in the `Composer` table. Conceptually, the subquery is evaluated five times. E.g. for `C.CNo = 1` the following subquery is executed:

```
SELECT * FROM Piece P
WHERE P.CNo = 1
```

PNo	CNo	PName
1000	1	The Four Seasons

- The result is not empty. So composer 1 does not qualify.

NOT EXISTS (3)

- Next, when C points to the second row of `Composer`, the subquery is executed for `C.CNo = 2`:

```
SELECT * FROM Piece P
WHERE P.CNo = 2
```

PNo	CNo	PName
1002	2	B-minor Mass
1003	2	Christmas Oratorio

- And so on for the other rows of `Composer`. For Composer 1–4, a solution for the subquery exists. So they do not qualify.
- However, the result for `C.CNo = 5` is empty:

```
SELECT * FROM Piece P
WHERE P.CNo = 5
```

no rows selected

- Thus composer 5 (`Dvorak`) is the result of the main query.

NOT EXISTS (4)

- Note that while in the inner query, you can access tuple variables from the outer query, the converse is forbidden:

```
SELECT CName, P.PNO     Wrong!
FROM   Composer C
WHERE  NOT EXISTS (SELECT * FROM Piece P
                   WHERE P.CNo = C.CNo)
```

- 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 are called “correlated subqueries”.

  You can think of correlated subqueries 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 acces variables from the outer query are called “non-correlated subqueries”.

  So 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 below):

```
SELECT CName
FROM   Composer C
WHERE  NOT EXISTS (SELECT * FROM Piece P)
```

- 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 `Piece` table, no matter for what composer (since there is no join condition), the `NOT EXISTS` will always be false.

NOT EXISTS (7)

- Until now, we required that for every unqualified attribute name (i.e. attribute name without tuple variable), there is 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 CName
FROM   Composer C
WHERE  NOT EXISTS (SELECT * FROM Piece P
                   WHERE CNo = C.CNo)
```


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 CName
FROM   Composer X
WHERE  NOT EXISTS (SELECT * FROM Piece X
                   WHERE ...)
```

- The rule of proximity applies: References to X in the subquery mean Piece X. The variable declared in the outer query becomes “shadowed” and cannot be accessed in the subquery.

NOT EXISTS (9)

- Who is the earliest born composer in the database?

```
SELECT CName
FROM   Composer X
WHERE  NOT EXISTS (SELECT * FROM Composer Y
                   WHERE Y.Born < X.Born)
```

- I.e. you want a composer X such that there is no composer Y who is born before X.
- Since for NOT EXISTS the returned columns do not matter, you should use “SELECT *” in the subquery.

  Some authors say that in some systems SELECT null or SELECT 1 is actually faster than SELECT *.

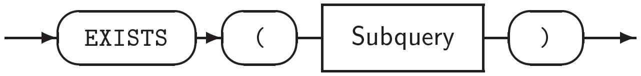
NOT EXISTS (10)

- Which composers born between 1700 and 1800 did not write a symphony?

```
SELECT CName
FROM   Composer C
WHERE  Born BETWEEN 1700 AND 1800
AND    NOT EXISTS (SELECT * FROM Piece P
                   WHERE P.CNo = C.CNo
                   AND PName LIKE '%Symphony%')
```

NOT EXISTS (11)

Atomic Formula (Form 6):



- A subquery is an expression of the form `SELECT ... FROM ... [WHERE ...] [GROUP BY ...] [HAVING ...]`.
- SQL-92 does also allow `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.
- Note that subqueries must be enclosed in parentheses `(...)`.

NOT EXISTS (12)

- Of course, it is possible to use `EXISTS` also without negation.
- Which composers have at least one piece stored in the DB?

```
SELECT CNo, CName
FROM   Composer C
WHERE  EXISTS (SELECT * FROM Piece P
               WHERE P.CNo = C.CNo)
```

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

```
SELECT DISTINCT C.CNo, C.CName
FROM   Composer C, Piece P
WHERE  C.CNo = P.CNo
```


Common Errors (1)

Exercises:

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

```
SELECT CName
FROM   Composer C, Piece P
WHERE  C.CNo <> P.CNo
```

- What about this query?

```
SELECT CName
FROM   Composer C, Piece P
WHERE  C.CNo = P.CNo AND P.PNo IS NULL
```


Common Errors (2)

- Will this query find composers who have not written a symphony?

```
SELECT CName
FROM   Composer C, Piece P
WHERE  C.CNo = P.CNO
AND    NOT (P.PName LIKE '%Symphony%')
```

- Will this query find the earliest born composer?

```
SELECT X.CName
FROM   Composer X, Composer Y
WHERE  X.Born < Y.BORN
```

Common Errors (3)

- It is important to understand that the absence/non-existence of a row is something very different than the existence of a row with a different value.
- If you need to write a query which behaves non-monotonic (i.e. insertion of a row could invalidate an answer) you need `NOT EXISTS`, `NOT IN`, `<>ALL` etc.
- There is no way to write it without a subquery.

  Except possibly using an outer join.
- Another common error is to use a non-correlated subquery with `NOT EXISTS` (forgetting the join condition in the subquery). See above.

Subqueries Returning Sets (1)

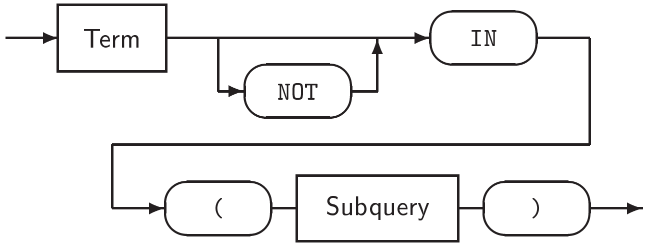
- `NOT EXISTS` is not the only way for using subqueries in SQL.
- You can also use `IN` ($\in$) and `NOT IN` ($\notin$) if the subquery returns a table with a single column (i.e. a set).

  In SQL92, Oracle and DB2 you can do comparisons for tuples, so that the subquery can have more than one column. In SQL86 and SQL Server this is not possible.
- Which composers have no pieces in the database (again)?

```
SELECT CName
FROM   Composer C
WHERE  CNo NOT IN (SELECT CNo FROM Piece P)
```


Subqueries Returning Sets (2)

Atomic Formula (Form 7):



- The Subquery must result in a table with a single column (a set).
- As mentioned above, SQL92, Oracle, and DB2 also support e.g.

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


Subqueries Returning Sets (3)

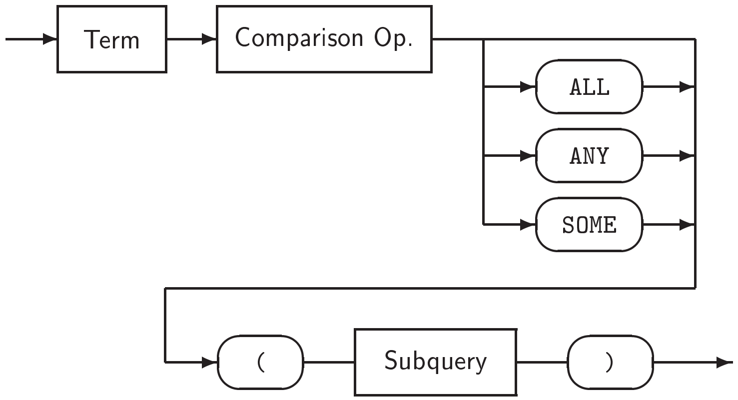
- It is possible to compare a value with all values in a set. You can require that the comparison returns true for all set elements (`ALL`) or for at least one set element (`ANY`, `SOME`).
- Which composer has the earliest birth year (again)?

```
SELECT CName
FROM   Composer
WHERE  Born <= ALL (SELECT Born
                    FROM COMPOSER)
```

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

Subqueries Returning Sets (4)

Atomic Formula (Form 8):



Subqueries Returning Sets (5)

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 you to do the 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 composer has the earliest birth year (again)?

```
SELECT CName
FROM   Composer
WHERE  Born = (SELECT MIN(Y.Born)
                 FROM COMPOSER Y)
```

- 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(Y.Born)
         FROM COMPOSER Y) = BORN
```

- 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 reasonable 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 for instance needed if we want a "nested aggregation".
- What is the average number of pieces in the DB for each composer? (Count only composers with at least one piece.)

```
SELECT AVG(X.Num)
FROM   (SELECT   CNo, COUNT(*) Num
        FROM     Piece
        GROUP BY CNo) X
```

X

CNo	Num
1	1
2	2
3	1
4	2

AVG(X.Num)
1.5

Subqueries in the FROM-Clause (3)

- In SQL92, SQL Server, and DB2 you must declare a tuple variable for the subquery, in Oracle this is optional.
- In SQL92, DB2, and SQL Server (but not in Oracle8), you can rename the columns in this way:

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

- Oracle also supports nested aggregations in this way:

```
SELECT   AVG(COUNT(*))
FROM     Piece
GROUP BY CNo
```

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 actually makes the GROUP BY clause unnecessary.
- E.g. print for every composer the number of his/her pieces in the database (including composers with 0 pieces):

```
SELECT CName, (SELECT COUNT(*)
               FROM   Piece P
               WHERE  P.CNo = C.CNo)
FROM   Composer C
```

- This does not work in Oracle8.

Exercises

Consider again the database containing invoices (bills):

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

Invoice(<u>No</u>, CustNo → Customer, 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`.

  $R \cup S$ is the set of all tuples contained in R, in S, or in both.
- `UNION` is needed since otherwise there would be no way to construct one result column which contains values drawn from different tables/columns.
- SQL distinguishes between
  - `UNION`: $\cup$ with duplicate elimination, and
  - `UNION ALL`: $\cup$/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 (when you want to code an **if ... then ... else ...**).

  But see also conditional expressions in Appendix H.

UNION (3)

- List all composers together with the number of music pieces from them in the database (including those with 0 pieces).

```
SELECT   CName, COUNT(*)
FROM     Composer C, Piece P
WHERE    C.CNo = P.CNo
GROUP BY CName
UNION ALL
SELECT   CName, 0
FROM     Composer
WHERE    CNo NOT IN (SELECT P.CNo FROM Piece P)
```

UNION (4)

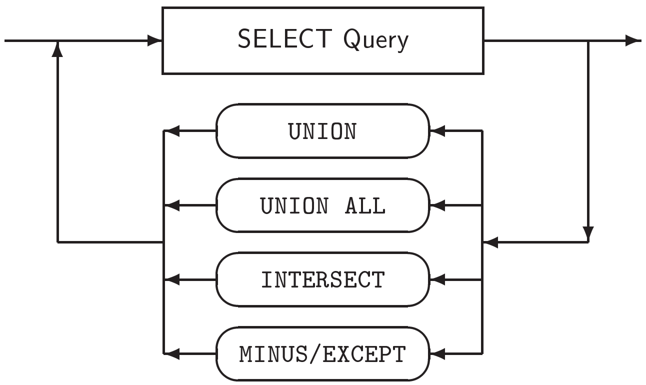
- SQL Server and SQL86 have only `UNION`/`UNION ALL`.
- The SQL92 standard also allows `EXCEPT` (set difference, $-$) and `INTERSECT` ($\cap$). These operators are supported in DB2.
- In Oracle8, you must write `MINUS` instead of `EXCEPT`.

  `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.

UNION (5)

Table Expression:



Sorting Output (1)

- Print the pieces sorted chronologically by composer (youngest first), for each composer alphabetically by title.

```
SELECT   CName, Born, PName
FROM     Composer C, Piece P
WHERE    C.CNo = P.CNo
ORDER BY Born DESC, PName
```

CName	Born	PName
Prokofiev	1891	Cinderella
Prokofiev	1891	Classical Symphony
Mozart	1756	Missa Solemnis
Bach	1685	B-minor Mass
Bach	1685	Christmas Oratorio
Vivaldi	1678	The Four Seasons

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.

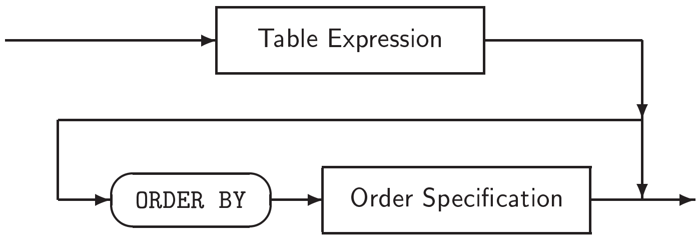
  In this example, it is not possible to sort the composers chronologically without printing their birth year. But tools like SQL*Plus can suppress columns.
- Sometimes one needs to add columns to database tables to get a sort value, e.g. if the “University of Pittsburgh” should appear in a sorted sequence 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 “`ORDER BY`” is pure cosmetics. 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.
- You can use column name or number in the “`ORDER BY`” list.

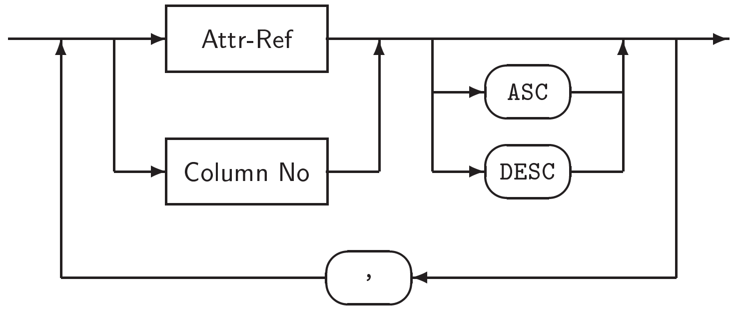
Sorting Output (4)

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)

- In SQL-92 one can use 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 composers with their pieces:

```
SELECT CName, PName
FROM   Composer NATURAL JOIN Piece
```

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

Joins in SQL-92 (2)

- One can also explicitly specify the join condition (actually, one has to do this in DB2 and SQL Server):

```
SELECT CName, PName
FROM   Composer C JOIN Piece P ON C.CNo = P.CNo
```

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

```
SELECT   CName, COUNT(PNo)
FROM     Composer LEFT OUTER JOIN Piece
         ON Composer.CNo = Piece.CNo
GROUP BY CName
```


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 CName, PName
FROM   Composer JOIN Piece USING (CNo)
```

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 music pieces per composer (including 0):

```
SELECT   CName, COUNT(PNo)
FROM     Composer C, Piece P
WHERE    C.CNo = P.Cno (+)
GROUP BY CName
```

- The condition can also be written: “`P.CNo (+) = C.Cno`”.