Part 6: Advanced SQL Repetition

References:

- Elmasri/Navathe:Fundamentals of Database Systems, 3rd Edition, 1999.
  Chap. 8, "SQL — The Relational Database Standard" (Sect. 8.2, 8.3.3, part of 8.3.4.)
- 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.
- Heuer/Saake: Datenbanken, Konzepte und Sprachen (in German), Thomson, 1995.
- Date/Darwen: A Guide to the SQL Standard, Fourth Edition, Addison-Wesley, 1997.
- Date: A Guide to the SQL Standard, First Edition, Addison-Wesley, 1987.
- van der Lans: SQL, Der ISO-Standard (in German). Hanser, 1990.
- Sunderraman: Oracle Programming, A Primer. Addison-Wesley, 1999.
- Oracle 8i SQL Reference, Release 2 (8.1.6), Dec. 1999, Part No. A76989-01.
- Chamberlin: A Complete Guide to DB2 Universal Database. Morgan Kaufmann, 1998.
- Microsoft SQL Server Books Online: Accessing and Changing Data.
- Microsoft Jet Database Engine Programmer's Guide, 2nd Edition (Part of MSDN Library Visual Studio 6.0).
- DuBois: MySQL. New Riders Publishing, 2000, ISBN 0-7357-0921-1, 756 pages.

Objectives

After completing this chapter, you should be able to:

- write advanced queries in SQL including, e.g.,
  - several tuple variables over the same relation.
  - Aggregations, `GROUP BY`, `HAVING`
  - `NOT EXISTS` / `NOT IN`
- Avoid errors and unnecessary complications.
- Check given queries for errors or equivalence.
- Evaluate the portability of certain constructs.

Overview

Example Database

STUDENTS

SID	FIRST	LAST	EMAIL
101	Ann	Smith	...
102	Michael	Jones	(null)
103	Richard	Turner	...
104	Maria	Brown	...

EXERCISES

CAT	ENO	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

RESULTS

SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

Lexical Syntax Overview

- The lexical syntax of a language defines how word symbols ("tokens") are composed from single characters. E.g. it defines the exact syntax of
  - Identifiers (names for e.g. tables, columns),
  - Literals (datatype constants, e.g. numbers),
  - Keywords, Operators, Punctation marks.
- Thereafter, the syntax of queries and other commands is defined in terms of these word symbols.

White Space and Comments

White space is allowed between words (tokens):

- Spaces (normally also tabulator characters)
- Line breaks
- Comments:
  - ◇ From "--" to ⟨Line End⟩

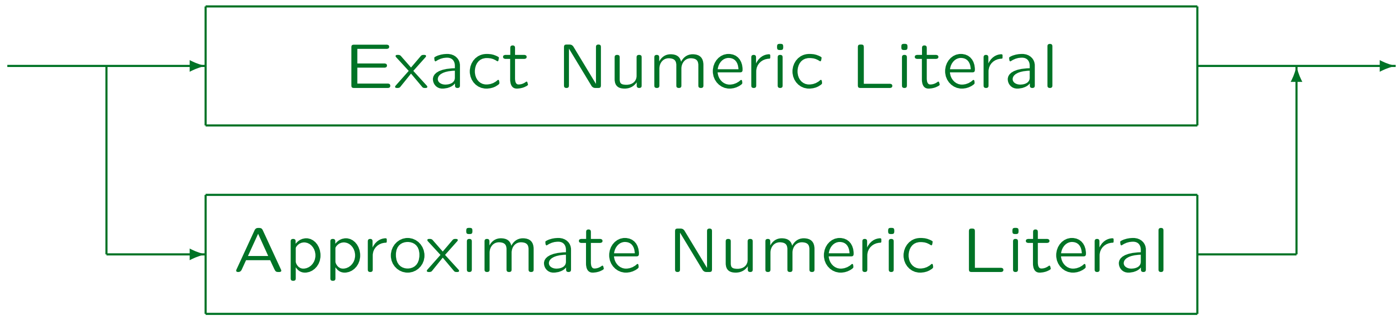
    Supported in SQL-92, Oracle, SQL Server, IBM DB2, MySQL.
    MySQL requires a space after the "--", SQL-92 does not.
    Access does not support this comment, and also not /* ...*/.
  - ◇ From "/*" to "*/"

    Supported only in Oracle, SQL Server, MySQL: Less portable.

SQL is a free-format language like Pascal, C, Java.

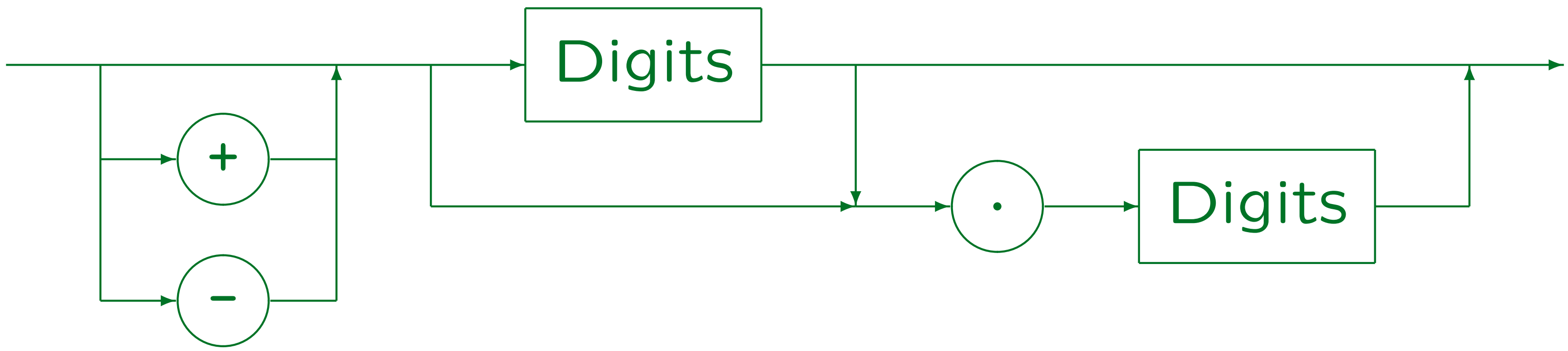
Numbers (1)

- Numeric literals are constants of numeric data types (fixed point and floating point numbers).
- E.g.: `1, +2., -34.5, -.67E-8`
- Note that numbers are not enclosed in quotes.
- Numeric Literal:



Numbers (2)

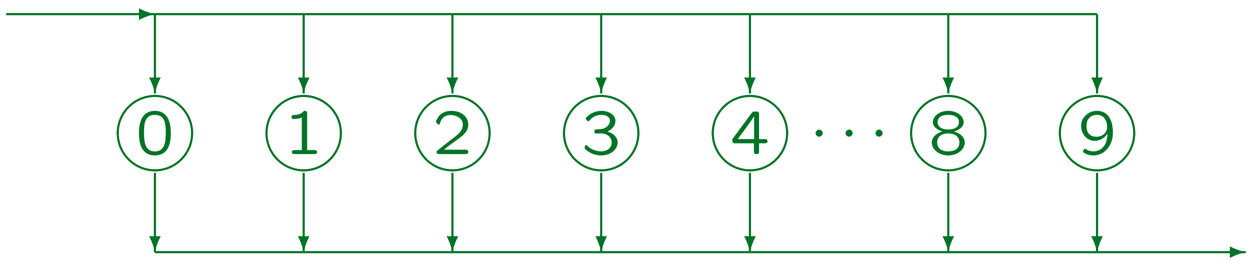
- Exact Numeric Literal:



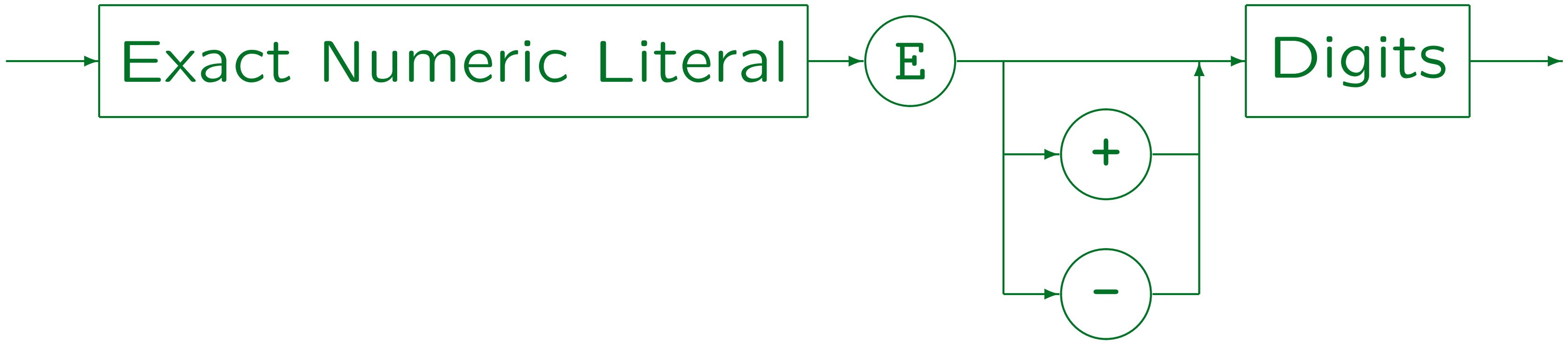
- Digits (Unsigned Integer):

Numbers (3)

- Digit:



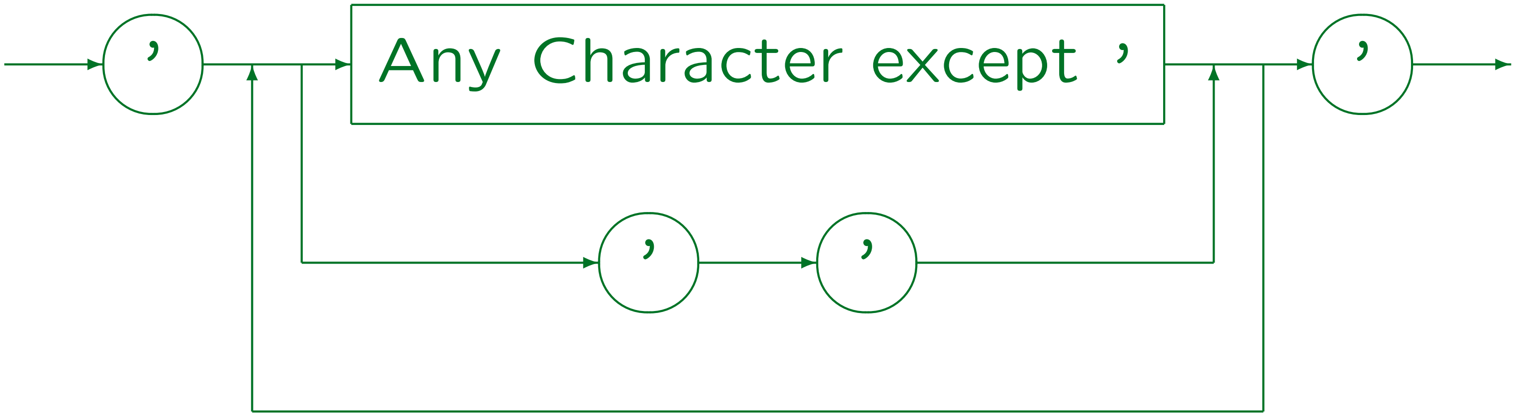
- Approximate Numeric Literal:



Character Strings (1)

- A character string constant/literal is a sequence of characters enclosed in single quotes, e.g. `'abc'`.
- Single quotes in a string must be doubled, e.g. `'John''s Book'`.

  The real value of the string is `John's Book` (with a single quote). The doubling is only a way to input it.



Character Strings (2)

- The SQL-92 standard allows splitting strings between lines (with each segment enclosed in ').

  MySQL does support this syntax. Oracle, SQL Server, and Access do not support it. However, strings can be combined with the concatenation operator (|| in Oracle, + in SQL Server and Access).

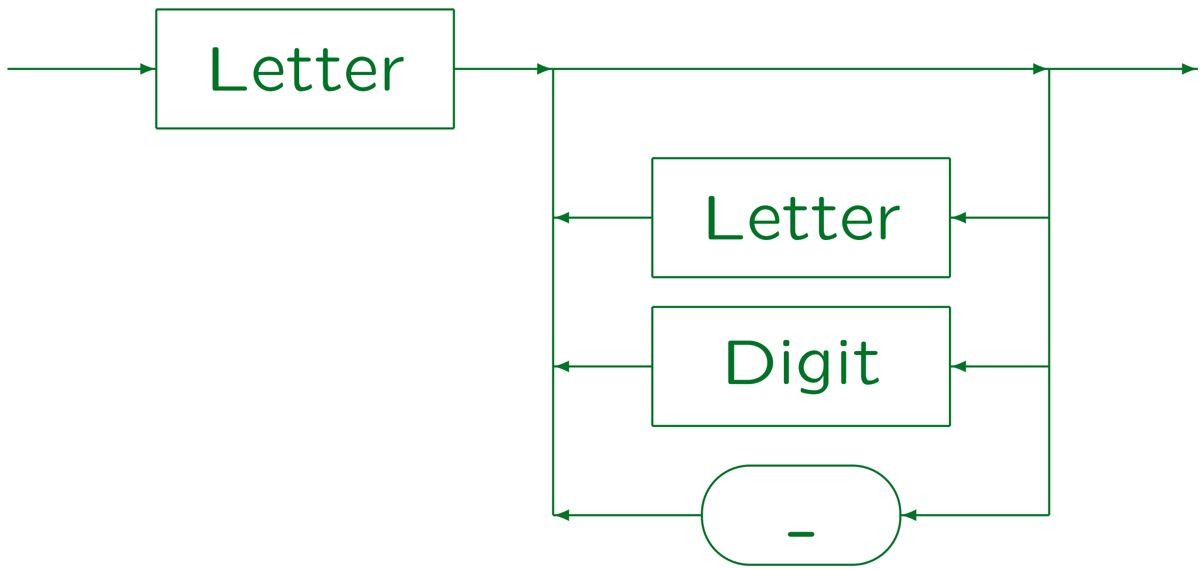
- SQL-92 and all five DBMS allow line breaks inside string constants.

  I.e. the quote can be closed on a subsequent line.

- Microsoft SQL Server, MS Access, and MySQL accept also string literals enclosed in double quotes. This does not conform to the standard.

Identifiers (1)

- Identifiers are used e.g. as table and column names.



- E.g. `Instructor_Name`, `X27`, but not `_XYZ`, `12`, `2BE`.

Identifiers (2)

- Identifiers can have up to 18 characters (at least).

System	Length	First Character	Other Characters
SQL-86	$\leq$ 18	A-Z	A-Z,0-9
SQL-92	$\leq$ 128	A-Z,a-z	A-Z,a-z,0-9,_
Oracle	$\leq$ 30	A-Z,a-z	A-Z,a-z,0-9,_,#,$
SQL Server	$\leq$ 128	A-Z,a-z,_,(@,#)	A-Z,a-z,0-9,_,@,#,$
IBM DB2	$\leq$ 18 (8)	A-Z,a-z	A-Z,a-z,0-9,_
Access	$\leq$ 64	A-Z,a-z	A-Z,a-z,0-9,_
MySQL	$\leq$ 64	A-Z,a-z,0-9,_,$	A-Z,a-z,0-9,_,$

Intermediate SQL-92: "_" at the end forbidden. Entry Level: Like SQL-86 (plus "_").
In MySQL, identifiers can start with digits, but must contain at least one letter.
Access might permit more characters, depending on the context.

- Names must be different from all reserved words.

  There are a lot of reserved words, see below. Embeddings in a programming language (PL/SQL, Visual Basic) add reserved words.

Identifiers (3)

- Identifiers (and keywords) are not case sensitive.

  It seems that this is what the SQL-92 standard says (the book by Date/Darwen about the Standard states this clearly). Oracle SQL*Plus converts all letters outside quotes to uppercase. In SQL Server, case sensitivity can be chosen at installation time. In MySQL, case sensitivity of table names depends on the case sensitivity of file names in the underlying operating system (tables are stored as files). Within a query, one must use consistent case. However, keywords and column names are not case sensitive.

- It is possible to use also national characters.

  This is implementation dependent. E.g. in Oracle, one chooses a database character set when the database is installed. Alphanumeric characters from this character set can be used in identifiers.

SQL Reserved Words (1)

[1] = Oracle 8.0
[2] = SQL-92
[3] = SQL Server 7

— A —

ABSOLUTE[2]
ACCESS[1]
ACTION[2]
ADD[1,2,3]
ALL[1,2,3]
ALLOCATE[2]
ALTER[1,2,3]
AND[1,2,3]
ANY[1,2,3]
ARE[2]
AS[1,2,3]
ASC[1,2,3]
ASSERTION[2]
AT[2]
AUTHORIZATION[2,3]
AUDIT[1]
AVG[2,3]

— B —

BACKUP[3]
BEGIN[2,3]
BETWEEN[1,2,3]
BIT[2]
BIT_LENGTH[2]
BOTH[2]
BREAK[3]
BROWSE[3]
BULK[3]
BY[1,2,3]

— C —

CASCADE[2,3]
CASCADED[2]
CASE[2,3]
CATALOG[2]
CHAR[1,2]
CHARACTER[2]
CHAR_LENGTH[2]
CHARACTER_LENGTH[2]
CHECK[1,2,3]
CHECKPOINT[3]
CLOSE[2,3]
CLUSTER[1]
CLUSTERED[3]
COALESCE[2,3]
COLLATE[2]
COLLATION[2]
COLUMN[1,3]
COMMENT[1]
COMMIT[2,3]
COMMITTED[3]
COMPRESS[1]
COMPUTE[3]

SQL Reserved Words (2)

CONFIRM[3]
CONNECT[1,2]
CONNECTION[2]
CONSTRAINT[2,3]
CONSTRAINTS[2]
CONTAINS[3]
CONTAINSTABLE[3]
CONTINUE[2,3]
CONTROLROW[3]
CONVERT[2,3]
CORRESPONDING[2]
COUNT[2,3]
CREATE[1,2,3]
CROSS[2,3]
CURRENT[1,2,3]
CURRENT_DATE[2,3]
CURRENT_TIME[2,3]
CURRENT_TIMESTAMP[2,3]
CURRENT_USER[2,3]
CURSOR[2,3]

— **D** —

DATABASE[3]
DATE[1,2]
DAY[2]
DBCC[3]
DEALLOCATE[2,3]
DEC[2]
DECIMAL[1,2]
DECLARE[2,3]
DEFAULT[1,2,3]
DEFERRABLE[2]
DEFERRED[2]
DELETE[1,2,3]
DENY[3]
DESC[1,2]
DESCRIBE[2]
DESCRIPTOR[2]
DIAGNOSTICS[2]
DISCONNECT[2]
DISK[3]
DISTINCT[1,2,3]
DISTRIBUTED[3]
DOMAIN[2]
DOUBLE[2,3]
DROP[1,2,3]
DUMMY[3]
DUMP[3]

— **E** —

ELSE[1,2,3]
END[2,3]
END-EXEC[2]
ERRLVL[3]
ERROREXIT[3]
ESCAPE[2,3]
EXCEPT[2,3]
EXCEPTION[2]

SQL Reserved Words (3)

EXCLUSIVE[1]
EXEC[2,3]
EXECUTE[2,3]
EXISTS[1,2,3]
EXIT[3]
EXTERNAL[2]
EXTRACT[2]
— F —
FALSE[2]
FETCH[2,3]
FILE[1,3]
FILLFACTOR[3]
FIRST[2]
FLOAT[1,2]
FLOPPY[3]
FOR[1,2,3]
FOREIGN[2,3]
FOUND[2]
FREETEXT[3]
FREETEXTTABLE[3]
FROM[1,2,3]
FULL[2,3]
— G —
GET[2]
GLOBAL[2]
GO[2]
GOTO[2,3]
GRANT[1,2,3]
GROUP[1,2,3]
— H —
HAVING[1,2,3]
HOLDLOCK[3]
HOUR[2]
— I —
IDENTITY[2,3]
IDENTITY_INSERT[3]
IDENTITYCOL[3]
IDENTIFIED[1]
IF[3]
IMMEDIATE[1,2]
IN[1,2,3]
INCREMENT[1]
INDEX[1,3]
INDICATOR[2]
INITIAL[1]
INITIALLY[2]
INNER[2,3]
INPUT[2]
INSENSITIVE[2]
INSERT[1,2,3]
INT[2]
INTEGER[1,2]
INTERSECT[1,2,3]
INTERVAL[2]
INTO[1,2,3]
IS[1,2,3]

SQL Reserved Words (4)

ISOLATION[2,3]
— J —
JOIN[2,3]
— K —
KEY[2,3]
KILL[3]
— L —
LANGUAGE[2]
LAST[2]
LEADING[2]
LEFT[2,3]
LEVEL[1,2,3]
LIKE[1,2,3]
LINENO[3]
LOAD[3]
LOCAL[2]
LOCK[1]
LONG[1]
LOWER[2]
— M —
MATCH[2]
MAX[2,3]
MAXEXTENTS[1]
MIN[2,3]
MINUS[1]
MINUTE[2]
MIRROREXIT[3]
MODE[1]
MODIFY[1]
MODULE[2]
MONTH[2]
— N —
NAMES[2]
NATIONAL[2,3]
NATURAL[2]
NCHAR[2]
NETWORK[1]
NEXT[2]
NO[2]
NOAUDIT[1]
NOCHECK[3]
NOCOMPRESS[1]
NONCLUSTERED[3]
NOT[1,2,3]
NOWAIT[1]
NULL[1,2,3]
NULLIF[2,3]
NUMBER[1]
NUMERIC[2]
— O —
OCTET_LENGTH[2]
OF[1,2,3]
OFF[3]
OFFLINE[1]
OFFSETS[3]
ON[1,2,3]

SQL Reserved Words (5)

ONCE[3]
ONLINE[1]
ONLY[2,3]
OPEN[2,3]
OPENDATASOURCE[3]
OPENQUERY[3]
OPENROWSET[3]
OPTION[1,2,3]
OR[1,2,3]
ORDER[1,2,3]
OUTER[2,3]
OUTPUT[2]
OVER[3]
OVERLAPS[2]

— **P** —

PARTIAL[2]
PCTFREE[1]
PERCENT[3]
PERM[3]
PERMANENT[3]
PIPE[3]
PLAN[3]
POSITION[2]
PRECISION[2,3]
PREPARE[2,3]
PRESERVE[2]
PRIMARY[2,3]
PRINT[3]
PRIOR[1,2]
PRIVILEGES[1,2,3]
PROC[3]
PROCEDURE[2,3]
PROCESSEXIT[3]
PUBLIC[1,2,3]

— **R** —

RAISERROR[3]
RAW[1]
READ[2,3]
READTEXT[3]
REAL[2]
RECONFIGURE[3]
REFERENCES[2,3]
RELATIVE[2]
RENAME[1]
REPEATABLE[3]
REPLICATION[3]
RESOURCE[1]
RESTORE[3]
RESTRICT[2,3]
RETURN[3]
REVOKE[1,2,3]
RIGHT[2,3]
ROLLBACK[2,3]
ROW[1]
ROWCOUNT[3]
ROWGUIDCOL[3]

SQL Reserved Words (6)

ROWID[1]
ROWNUM[1]
ROWS[1,2]
RULE[3]
— S —
SAVE[3]
SCHEMA[2,3]
SCROLL[2]
SECOND[2]
SECTION[2]
SELECT[1,2,3]
SERIALIZABLE[3]
SESSION[1,2]
SESSION_USER[2,3]

SET[1,2,3]
SETUSER[3]
SHARE[1]
SHUTDOWN[3]
SIZE[1,2]
SMALLINT[1,2]
SOME[2,3]
SQL[2]
SQLCODE[2]
SQLERROR[2]
SQLSTATE[2]
START[1]
STATISTICS[3]
SUBSTRING[2]

SUCCESSFUL[1]
SUM[2,3]
SYNONYM[1]
SYSDATE[1]
SYSTEM_USER[2,3]
— T —
TABLE[1,2,3]
TAPE[3]
TEMP[3]
TEMPORARY[2,3]
TEXTSIZE[3]
THEN[1,2,3]
TIME[2]
TIMESTAMP[2]

TIMEZONE_HOUR[2]
TIMEZONE_MINUTE[2]
TO[1,2,3]
TOP[3]
TRAILING[2]
TRAN[3]
TRANSACTION[2,3]
TRANSLATE[2]
TRANSLATION[2]
TRIGGER[1,3]
TRIM[2]
TRUE[2]
TRUNCATE[3]
TSEQUAL[3]

SQL Reserved Words (7)

— **U** —

UID[1]
UNCOMMITTED[3]
UNION[1,2,3]
UNIQUE[1,2,3]
UNKNOWN[2]
UPDATE[1,2,3]
UPDATETEXT[3]
UPPER[2]
USAGE[2]
USE[3]
USER[1,2,3]
USING[2]

— **V** —

VALIDATE[1]
VALUE[2]
VALUES[1,2,3]
VARCHAR[1,2]
VARCHAR2[1]
VARYING[2,3]
VIEW[1,2,3]

— **W** —

WAITFOR[3]
WHEN[2,3]
WHENEVER[1,2]
WHERE[1,2,3]
WHILE[3]
WITH[1,2,3]
WORK[2,3]
WRITE[2]
WRITETEXT[3]

— **Y** —

YEAR[2]

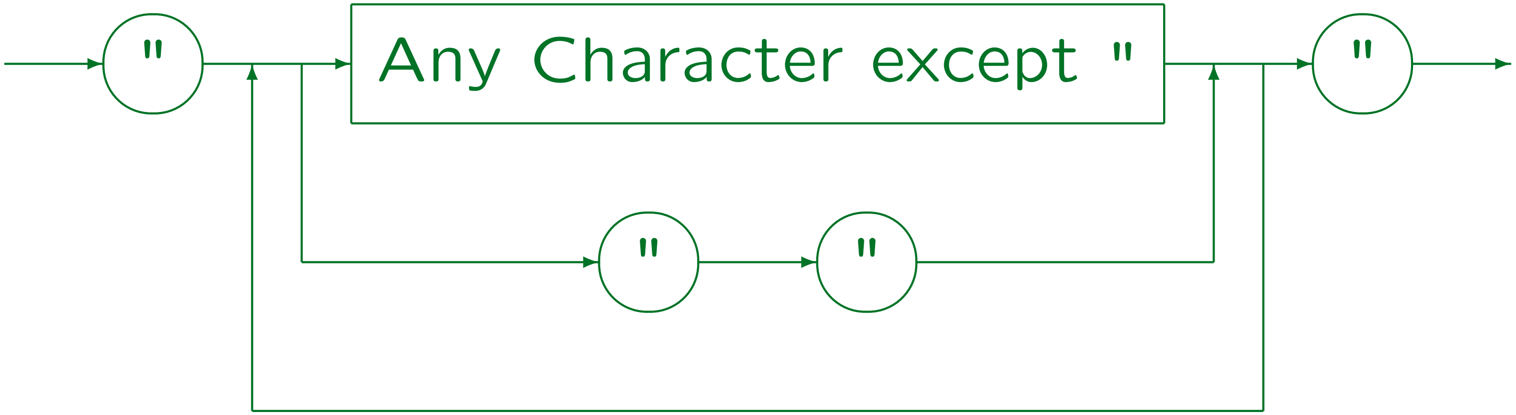
— **Z** —

ZONE[2]

Delimited Identifiers (1)

- It is possible to use any sequence of characters in double quotes as identifiers, e.g. "id, 2!".

  Such identifiers are case-sensitive, and there are no conflicts with reserved words. SQL-86 does not contain them.



Delimited Identifiers (2)

- Delimited identifiers are not character string constants! Character strings have the form '...'.

  SQL Server accepts ' and " for string constants, and uses [...] for delimited identifiers. "SET QUOTED_IDENTIFIER ON" selects the SQL-92 standard behaviour (but quoted identifiers are not case sensitive). Access understands [...] and `...` for delimited identifiers and excludes the characters ! . ` [ ] " and leading spaces in delimited identifiers.

- E.g. if you write in Oracle:

  ```
  SELECT * FROM EMP WHERE ENAME = "JONES"
  ```

  Error: "JONES" is an invalid column name.

  Quoted identifiers are normally used only to rename output columns (or if column names become reserved words in a new DBMS version).

Delimited Identifiers (3)

- Delimited identifiers are often used when output columns are renamed, e.g.

```
SELECT FIRST AS "First Name", LAST "Last Name"
FROM   STUDENTS
```

  Note that "AS" is optional (except in MS Access).

- But if the new column name is a legal identifier, the double quotes are not necessary:

```
SELECT FIRST AS FIRST_NAME, LAST Last_Name
FROM   STUDENTS
```

- At least in Oracle, it will be printed all-uppercase.

Summary: Lexical Errors

- Using double quotes, e.g. `"Smith"`, for string constants. This is a delimited identifier, no string.

  Some systems accept `"..."`, but that is a violation of the standard.

- Using quotes for numbers, e.g. `'123'`.

  This should give a type error. However, the DBMS may simply convert the type of one of the operands. Since < and so on are differently defined for strings and for numbers, this might be dangerous and should be avoided. E.g. `'12' < '3'`.

- Using reserved words as table, column, or tuple variable names.

  The error message might be strange (not understandable). Therefore, one should keep this possibility in mind.

Delimiting SQL Queries

- In Oracle SQL*Plus, every SQL statement must be terminated with a semicolon ";".

  Since SQL statements can extend over several lines, this is necessary so that SQL*Plus can see where the SQL statement is complete. Also when SQL is embedded into C programs, the semicolon is used as delimiter.

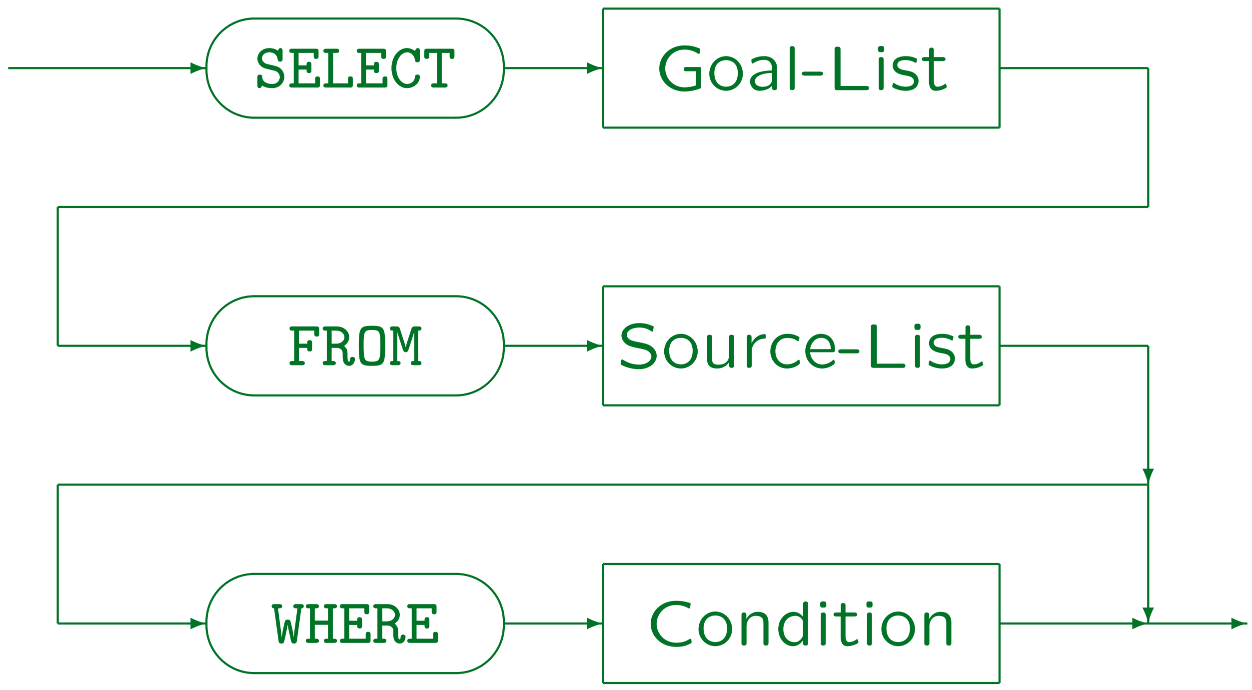
- But strictly speaking the semicolon is not part of the SQL statement.

  E.g. in the query analyzer window of MS SQL Server no semicolon is necessary. It might even be an error, as in the command line interface of DB2. Also, when SQL statements are passed to interface procedures as strings, as e.g. in ODBC, no semicolon is necessary.

Overview

Basic Query Syntax (1)

SELECT-Expression (Simplified):



Basic Query Syntax (2)

- E.g., to list the complete table "STUDENTS":

```
SELECT * FROM STUDENTS
```

- Every SQL query must contain the keywords SELECT and FROM.

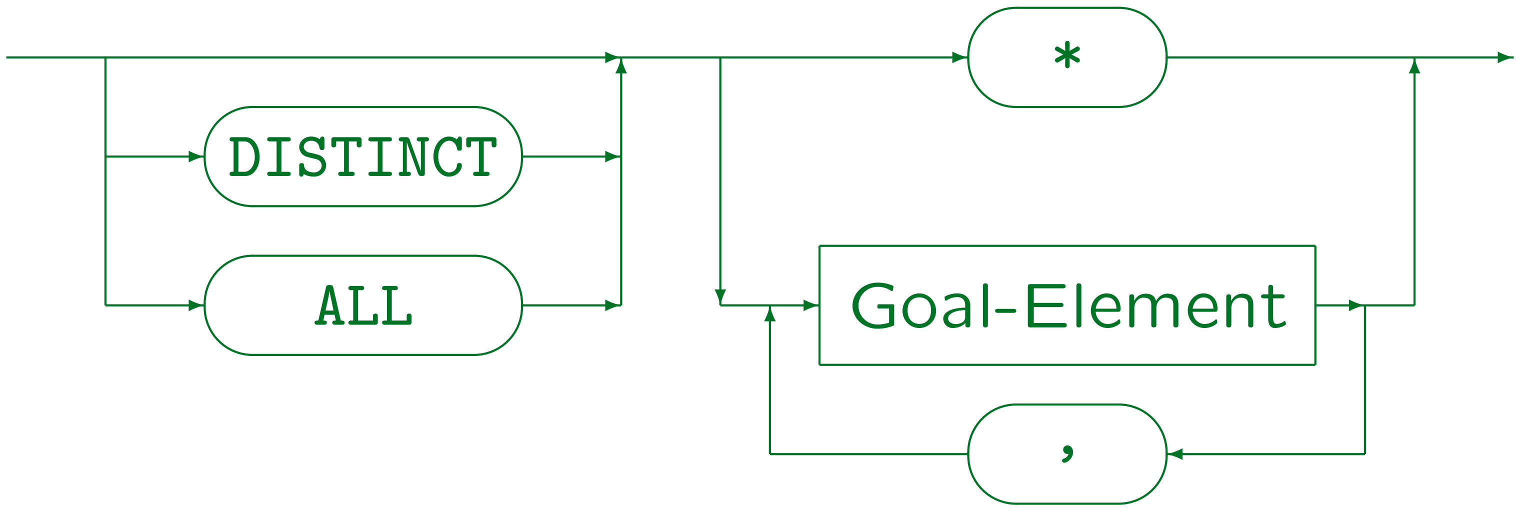
  Oracle provides a relation "DUAL" which has only one row. It can be used if only a computation is done without access to the database: `SELECT TO_CHAR(SQRT(2)) FROM DUAL`.

- However, in SQL Server, Access, and MySQL, the FROM-clause can be omitted, e.g. `SELECT 1+1`.

  In Oracle, DB2, and the SQL-92 Standard, this is a syntax error.

SELECT Syntax (1)

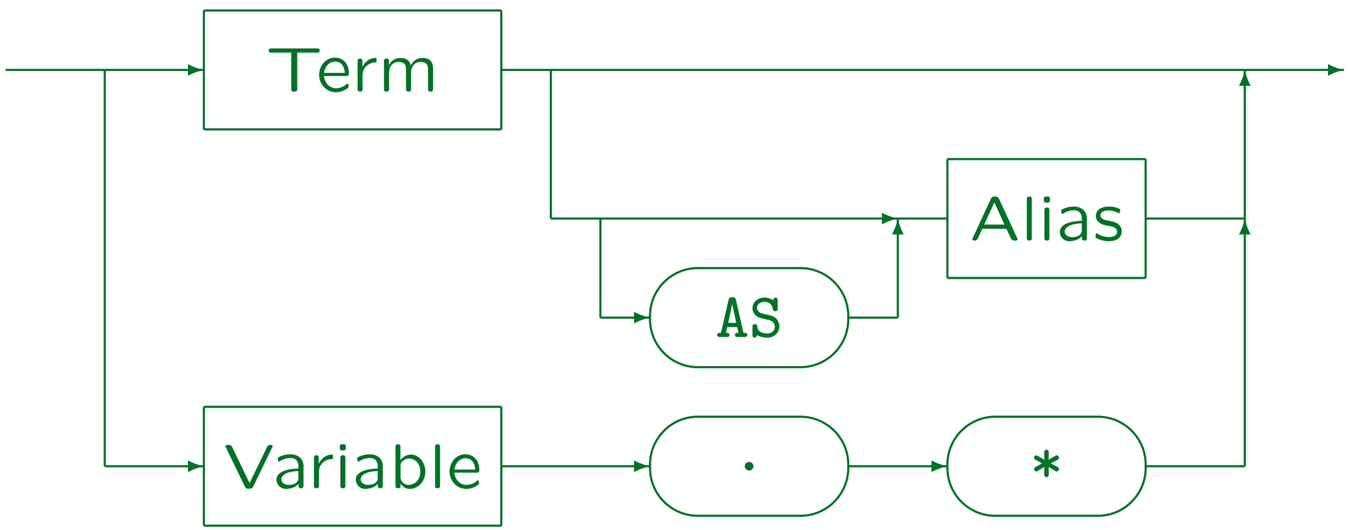
Goal-List (after SELECT):



- ALL (no duplicate elimination) is the default.

SELECT Syntax (2)

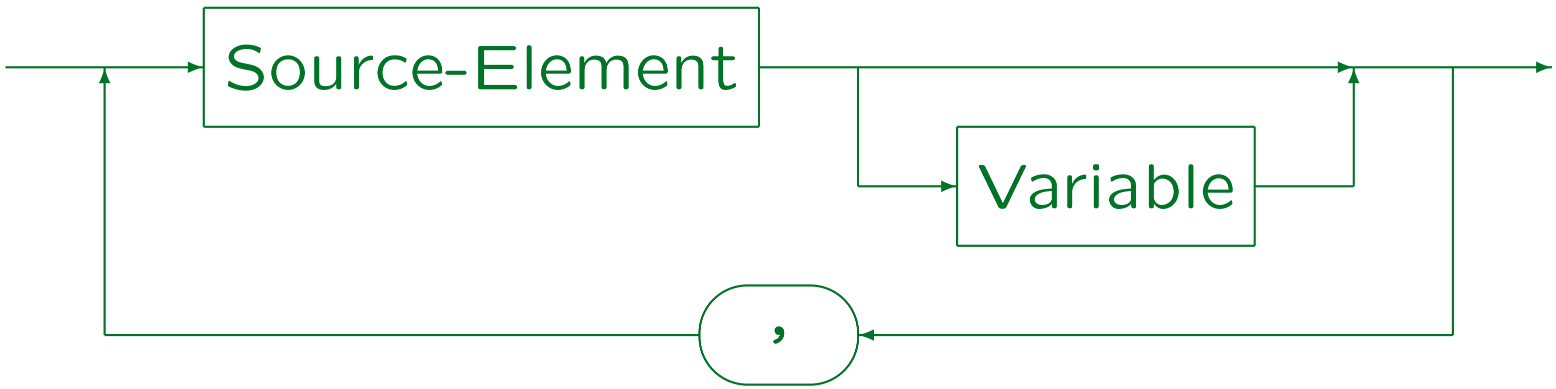
Goal-Element:



- "Variable.*" and "[AS] Alias" work in SQL-92, Oracle, SQL Server, and DB2, MySQL and Access (in Access "AS" is required). These constructs are not contained in the old SQL-86 standard.

FROM Syntax (1)

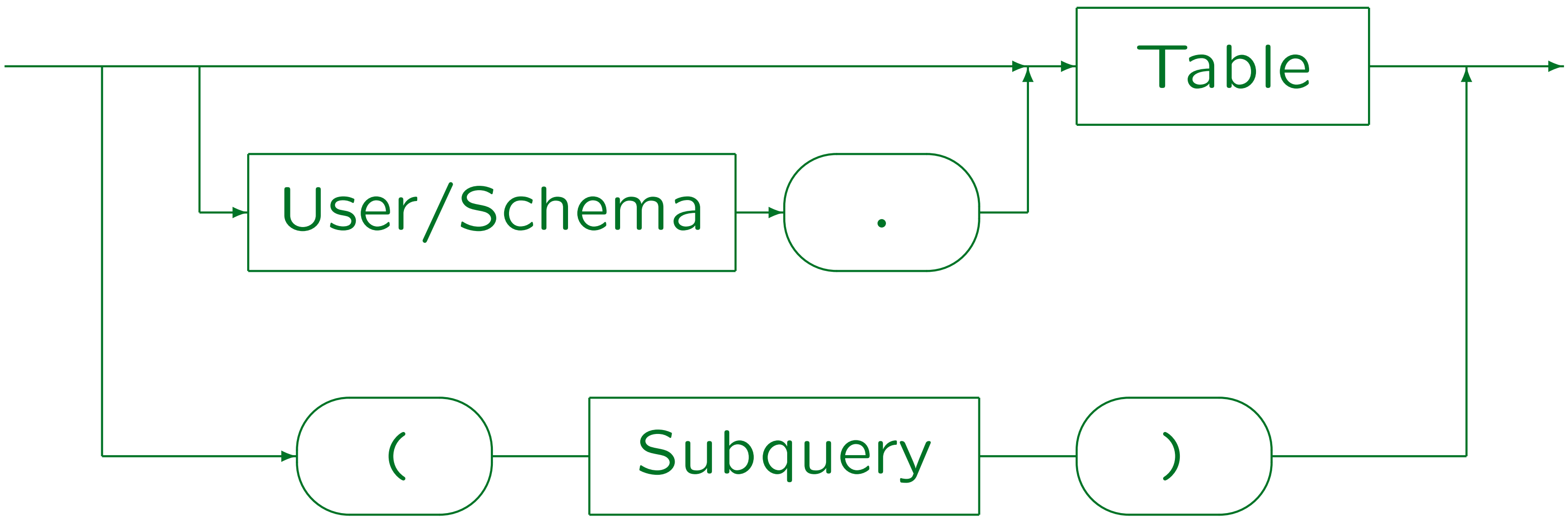
Source-List (after FROM):



- In SQL-92, SQL Server, Access, DB2, and MySQL (but not in Oracle 8i) one can write "AS" between Source-Element and Variable.
- In SQL-92 and DB2 (but not Oracle, SQL Server, Access, MySQL) new column names can be defined: "STUDENTS AS S(NO,FNAME,LNAME, EMAIL)".
- If the "Source-Element" is a subquery, the tuple variable is required in SQL-92, SQL Server, and DB2, but not in Oracle and Access. In this case the above column renaming syntax suddenly works in SQL Server.
- SQL-92, SQL Server, Access, DB2 support joins under FROM (see below).

FROM Syntax (2)

Source-Element:



- SQL-86 did not allow subqueries in the FROM-list.
- MySQL does not support subqueries at all.
- Basic (simplified) syntax of the FROM-clause:

```
FROM Table [Variable], ..., Table [Variable]
```

FROM Syntax (3)

Table Names:

- Tables of other users can be referenced in the FROM-list (if read permission was granted):

```
SELECT * FROM BRASS.EXERCISES
```

- The username is here really a name of a DB schema (one DBMS server can manage several schemas).

In Oracle, schema and user are more or less the same: Every user has his/her own schema, every schema belongs to exactly one user. In DB2, there can be multiple schemas per user and you can write "schema.table" as in Oracle. In SQL Server, a fully qualified name has the form "server.database.owner.table", but there are various abbreviations including "owner.table" or simply "table". In MySQL, one can write "database.table".

Example Database (again)

STUDENTS

SID	FIRST	LAST	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

EXERCISES

CAT	ENO	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

RESULTS

SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

Tuple Variables (1)

- The FROM clause can be understood as declaring variables that range over all tuples of a relation:

```
SELECT E.ENO, E.TOPIC
FROM   EXERCISES E
WHERE  E.CAT = 'H'
```

- This can be executed as:

```
for E in EXERCISES do
    if E.CAT = 'H' then
        print E.ENO, E.TOPIC
```

- E stands here for a single row in the table EXERCISES (the loop assigns each row in succession).

Tuple Variables (2)

- A tuple variable is always created: If not given a name explicitly, it will have the name of the relation:

```
SELECT EXERCISES.ENO, EXERCISES.TOPIC
FROM   EXERCISES
WHERE  EXERCISES.CAT = 'H'
```

- I.e. writing only `FROM EXERCISES` is understood as:

```
FROM   EXERCISES EXERCISES
```

  (The tuple variable called "`EXERCISES`" ranges over the rows of the table "`EXERCISES`".)

Tuple Variables (3)

- If a tuple variable name is explicitly declared, e.g.,

  ```
  FROM  EXERCISES E
  ```

  it is an error to try to access "EXERCISES.ENO".

  The tuple variable is now called "E", not "EXERCISES".

- When one refers to an attribute A of a tuple variable R, it is possible to write simply A instead of $R.A$ if R is the only tuple variable that has attribute A.

  This is explained further below. In the example, one can write "ENO" for the attribute, no matter whether one explicitly introduces a tuple variable or not.

Joins (1)

- Consider a query with two tuple variables:

```
SELECT A1, ..., An
FROM   STUDENTS S, RESULTS R
WHERE  C
```

- Then S will range over the 4 tuples in STUDENTS, and R will range over the 8 tuples in RESULTS. In principle, all $4 * 8 = 32$ combinations are considered:

for S **in** STUDENTS **do**
  for R **in** RESULTS **do**
    if C **then print** $A_1, \ldots, A_n$

Joins (2)

- A good DBMS might use a better evaluation algorithm (depending on the condition C).

  This is the task of the query optimizer. E.g. if C contains the join condition `S.SID = R.SID`, the DBMS might loop over all tuples in `RESULTS`, and find the corresponding `STUDENTS` tuple by using an index over `STUDENTS.SID` (most systems automatically create an index over the key attributes).

- But in order to understand the meaning of a query, it suffices to consider this simple algorithm.

  The query optimizer can use any algorithm that produces the same output, possibly in a different sequence (SQL does not define the sequence of the result tuples).

Joins (3)

- The join must be explicitly specified in the WHERE-condition:

```
SELECT R.CAT, R.ENO, R.POINTS
FROM   STUDENTS S, RESULTS R
WHERE  S.SID = R.SID     -- Join Condition
AND    S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

- Exercise: What will be the output of this query?

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S, RESULTS R
WHERE  R.CAT = 'H' AND R.ENO = 1
```

Wrong!

Joins (4)

- It is almost always an error if there are two tuple variables which are not linked (maybe indirectly) via join conditions.

  However, it is also possible that constant values are required for the join attributes instead. In seldom cases a connection might also be done in a subquery.

- In this query, all three tuple varibles are connected:

```
SELECT E.CAT, E.ENO, R.POINTS, E.MAXPT
FROM   STUDENTS S, RESULTS R, EXERCISES E
WHERE  S.SID = R.SID
AND    R.CAT = E.CAT AND R.ENO = E.ENO
AND    S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

Joins (5)

- The tuple variables are connected as follows:

```
(S) ------------------- (R) ------------------------ (E)
      S.SID = R.SID             R.CAT = E.CAT
                            AND R.ENO = E.ENO
```

- This corresponds to the key-foreign key relationships between the tables.

- If one forgets a join condition, one will often get many duplicates.

  Then it would be wrong to specify DISTINCT without thinking about the reason for the duplicates.

Attribute References (1)

- Attributes can be accessed in the form

```
Variable.Attribute
```

- If only one variable has this attribute, the variable name can be left out. E.g. this query is legal:

```
SELECT CAT, ENO, POINTS
FROM   STUDENTS S, RESULTS R
WHERE  S.SID = R.SID
AND    FIRST = 'Ann' AND LAST = 'Smith'
```

"FIRST" and "LAST" can only refer to "S". The attributes "CAT", "ENO", and "POINTS" can only refer to "R". However, "SID" alone would be ambiguous, since "S" and "R" both have an attribute with this name.

Attribute References (2)

- Consider this query:

```
SELECT ENO, SID, POINTS, MAXPT          Wrong!
FROM   RESULTS R, EXERCISES E
WHERE  R.ENO = E.ENO
AND    R.CAT = 'H' AND E.CAT = 'H'
```

- SQL requires that the user specifies whether he/she wants R.ENO or E.ENO in the SELECT-clause, although both are equal, so it actually does not matter.

  The rule is purely syntactic: If more than one tuple variable in the FROM clause has the attribute "ENO", the tuple variable cannot be left out, or the DBMS (e.g. Oracle) will print the error message "ORA-00918: column ambiguously defined". DB2, SQL Server, Access, MySQL are equally pedantic.

Query Formulation (1)

- Task: Write an SQL query which prints the topics of all exercises solved by Ann Smith.
- First it must understood that Ann Smith is a student, requiring a tuple variable `S` over `STUDENTS` and the condition `S.FIRST='Ann' AND S.LAST='Smith'`.
- Exercise topics are requested, so a tuple variable `E` over `EXERCISES` is needed, and the following piece can already be generated (several exercises can have the same topic):

```
SELECT DISTINCT E.TOPIC
```

Query Formulation (2)

- Finally, S and E are not connected.
- When trying to understand a relational database schema, it helps to draw a connection graph of the tables based on common columns (foreign keys):

- This shows that a tuple variable R over RESULTS is required, and yields the condition

```
S.SID = R.SID AND R.CAT = E.CAT AND R.ENO = E.ENO
```

Query Formulation (3)

- It is not always that simple. The connection graph may contain cycles, which makes the selection of the right path more difficult and error-prone.
- E.g. consider a course registration database that also contains GSA assignments.

  Graduate student assistants are advanced students (often PhD students) who help correcting homeworks etc.

Unnecessary Joins (1)

- Do not join more tables than needed.

  Queries will run more slowly: Most optimizers do not remove joins.

- E.g. results for Homework 1:

```
SELECT R.SID, R.POINTS
FROM   RESULTS R, EXERCISES E
WHERE  R.CAT = E.CAT AND R.ENO = E.ENO
AND    E.CAT = 'H' AND E.ENO = 1
```

- Can the following query ever give a different result?

```
SELECT SID, POINTS
FROM   RESULTS R
WHERE  R.CAT = 'H' AND R.ENO = 1
```

Unnecessary Joins (2)

- What will be the result of this query?

```
SELECT R.SID, R.POINTS
FROM   RESULTS R, EXERCISES E
WHERE  R.CAT = 'H' AND R.ENO = 1
```

- Is there any difference between these two queries?

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S
```

```
SELECT DISTINCT S.FIRST, S.LAST
FROM   STUDENTS S, RESULTS R
WHERE  S.SID = R.SID
```

Self Joins (1)

- It might be possible that in order to generate a result tuple, more than one tuple must be considered from the same relation.
- Task: Is there a student who got 10 points for both, Homwork 1 and Homework 2?

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S, RESULTS H1, RESULTS H2
WHERE  S.SID = H1.SID AND S.SID = H2.SID
AND    H1.CAT = 'H' AND H1.ENO = 1
AND    H2.CAT = 'H' AND H2.ENO = 2
AND    H1.POINTS = 10 AND H2.POINTS = 10
```

Self Joins (2)

- Find students who solved at least two exercises:

```
SELECT S.FIRST, S.LAST                    Wrong!
FROM   STUDENTS S, RESULTS E1, RESULTS E2
WHERE  S.SID = E1.SID AND S.SID = E2.SID
```

- The tuple variables `E1` and `E2` can point to the same input tuple.

- One must explicitly request that they are different:

```
WHERE S.SID = E1.SID AND S.SID = E2.SID
AND   (E1.CAT <> E2.CAT OR E1.ENO <> E2.ENO)
```

- This task can also be solved with aggregations.

Duplicate Elimination (1)

- One difference of SQL to relational algebra is that duplicates have to be explicitly eliminated in SQL.
- E.g. which exercises have already been solved by at least one student?

```
SELECT CAT, ENO
FROM   RESULTS
```

CAT	ENO
H	1
H	2
M	1
H	1
H	2
M	1
H	1
M	1

Duplicate Elimination (2)

- If the query might contain duplicates, and there is no specific reason why they should be shown, use "`SELECT DISTINCT`" (`DISTINCT` applies to rows, not columns):

```
SELECT DISTINCT CAT, ENO
FROM   RESULTS
```

CAT	ENO
H	1
H	2
M	1

- To emphasize that there are duplicates and that they are really wanted, one can write "`SELECT ALL`".

  However, "`ALL`" is the default.

Duplicate Elimination (3)

Sufficient condition for unnecessary DISTINCT:

- Let $\mathcal{K}$ be the set of attributes that appear as output columns under SELECT.

  The elements of $\mathcal{K}$ are of the form "Tuplevariable.Attribute". $\mathcal{K}$ is the set of attributes that have a unique value for a given output row.

- Add to $\mathcal{K}$ attributes A such that $A = c$ with a constant c appears in the WHERE-condition.

  This test assumes that the condition is a conjunction. Of course, a condition $c = A$ is treated in the same way. Conditions in subqueries do not count (subqueries are simply removed before the test is done).

Duplicate Elimination (4)

Test for unnecessary DISTINCT, continued:

- As long as something changes, do the following:
  - ◇ Add to $\mathcal{K}$ attributes A such that $A = B$ appears in the WHERE-condition and $B \in \mathcal{K}$.
  - ◇ If $\mathcal{K}$ contains a key of a tuple variable, add all other attributes of this tuple variable.
- If $\mathcal{K}$ contains a key of every tuple variable listed under FROM, DISTINCT is superfluous.

  If the query contains GROUP BY, one checks instead whether all GROUP BY columns are contained in $\mathcal{K}$.

Duplicate Elimination (5)

Example:

- Consider the following query:

```
SELECT DISTINCT S.FIRST, S.LAST, R.ENO, R.POINTS
FROM   STUDENTS S, RESULTS R
WHERE  R.CAT = 'H' AND R.SID = S.SID
```

- Let us assume that FIRST, LAST is declared as an alternative key for STUDENTS.
- $\mathcal{K}$ is initialized with S.FIRST, S.LAST, R.ENO, R.POINTS.
- R.CAT is added because of the condition R.CAT = 'H'.

Duplicate Elimination (6)

Example, continued:

- S.SID and S.EMAIL are added, because $\mathcal{K}$ contains a key of STUDENTS S (S.FIRST and S.LAST).
- R.SID is added because of R.SID = S.SID.
- Now $\mathcal{K}$ contains also a key of RESULTS R (R.SID, R.CAT, R.ENO), thus DISTINCT is superfluous.
- If FIRST, LAST were not a key of STUDENTS, this test would not succeed.

  However, this case it might be useful to print duplicates since in the real world, students are identified by name ("soft key").

Duplicate Elimination (7)

- Duplicates should be eliminated with DISTINCT, although it works also with GROUP BY:

```
SELECT   CAT, ENO      Bad Style!
FROM     RESULTS
GROUP BY CAT, ENO
```

This splits the table into groups of tuples: each group contains tuples that agree in the values for the grouping attributes CAT, ENO. For each group, only one output tuple is produced. Normally this is used to compute aggregation functions (SUM, COUNT) for each group.

- I would consider this as an abuse of GROUP BY.

However, GROUP BY is more flexibe than DISTINCT if one wants to eliminate only some duplicates. Also old versions of MySQL did not support DISTINCT. Then one had to use GROUP BY.

Summary: Join Errors

- Missing join conditions (very common)
- Unnecessary joins (make query slower)
- Problems when several tuple variables over the same relation are required: If these are "merged", one often gets an inconsistent condition (see below).
- Duplicates are often an indication for errors: One should understand the source of the duplicates and not simply specify `DISTINCT` to avoid the problem.
- An unnecessary `DISTINCT` should be avoided.

Overview

1. Lexical Syntax
2. Tuple Variables, Joins
3. Terms, Conditions, Logic, Null Values
4. Subqueries, Nonmonotonic Constructs
5. Aggregations
6. Union, ORDER BY, Outer Join

Terms (1)

- A term denotes a data element.

  Instead of term, one can also say "expression".

- Terms are:
  - ◇ Attribute References, e.g. `STUDENT.SID`.
  - ◇ Constants ("literals"), e.g. `'Ann'`, `1`.
  - ◇ Composed Terms, using datatype operators like `+`, `-`, `*`, `/` (for numbers), `||` (string concatenation), and datatype functions, e.g. `0.9 * MAXPT`.
  - ◇ Aggregation terms, e.g. `MAX(POINTS)`.

Terms (2)

- The SQL-86 standard contained only +, -, *, /.
- Current database management systems still differ in other data type operations.
- E.g. the operator || is contained in the SQL-92 standard, but does not work e.g. in SQL Server.

  String concatenation is written "+" in SQL Server and Access.
  In MySQL, one must write "concat(s_1, s_2)" (but there is "--ansi").
  Other datatype functions (e.g. SUBSTR) are even less standardized.
- SQL knows the standard precedence rules, e.g. that A+B*C means A+(B*C). Parentheses may be used.

Terms (3)

- Terms are used in conditions, e.g.

  `R.POINTS > E.MAXPT * 0.8`

  contains the terms "`R.POINTS`" and "`E.MAXPT * 0.8`".

- Also the `SELECT`-list can contain arbitrary terms:

```
SELECT LAST || ', ' || FIRST "Name"
FROM   STUDENTS
```

Name
Smith, Ann
Jones, Michael
Turner, Richard
Brown, Maria

Conditions (1)

- Conditions consist of atomic formulas, e.g.

  `POINTS >= 8,`

  connected by "AND", "OR", "NOT".

- AND binds more strongly than OR, thus

  `CAT = 'H' AND ENO = 1 OR ENO = 2`

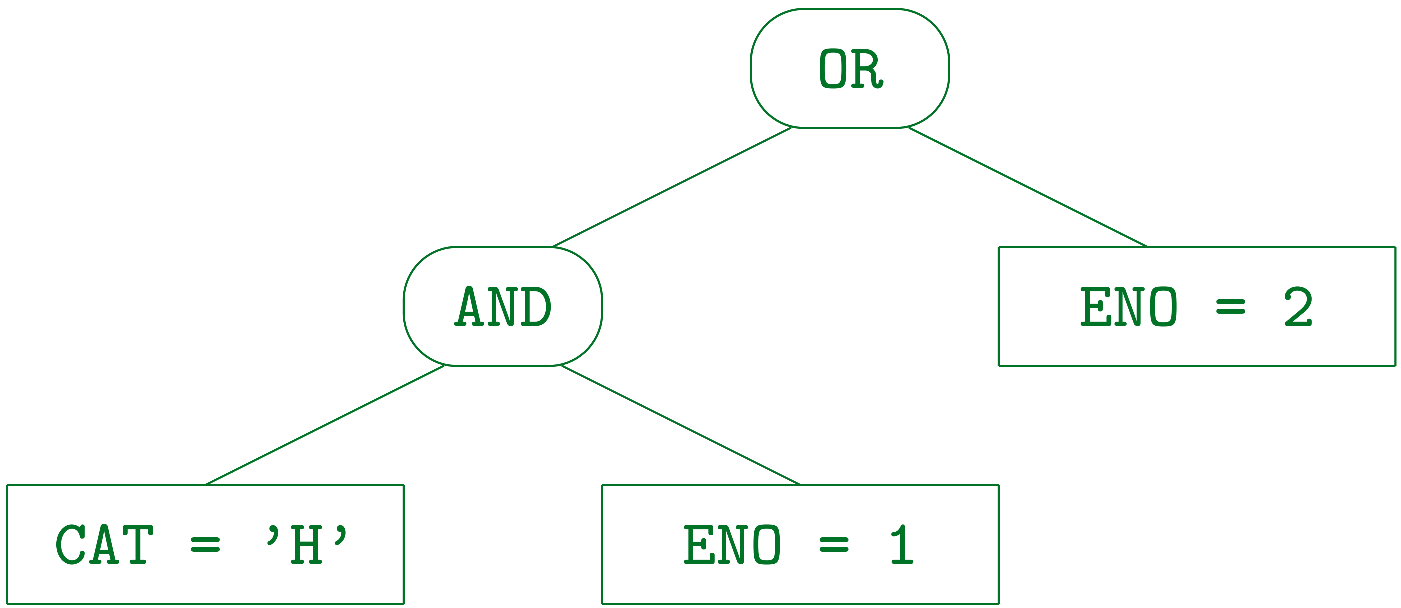
  is implicitly parenthesized as

  `(CAT = 'H' AND ENO = 1) OR ENO = 2`

- In this example, this is probably not intended.

Conditions (2)

- It might help to draw a complex condition (or complex term) as an "operator tree":



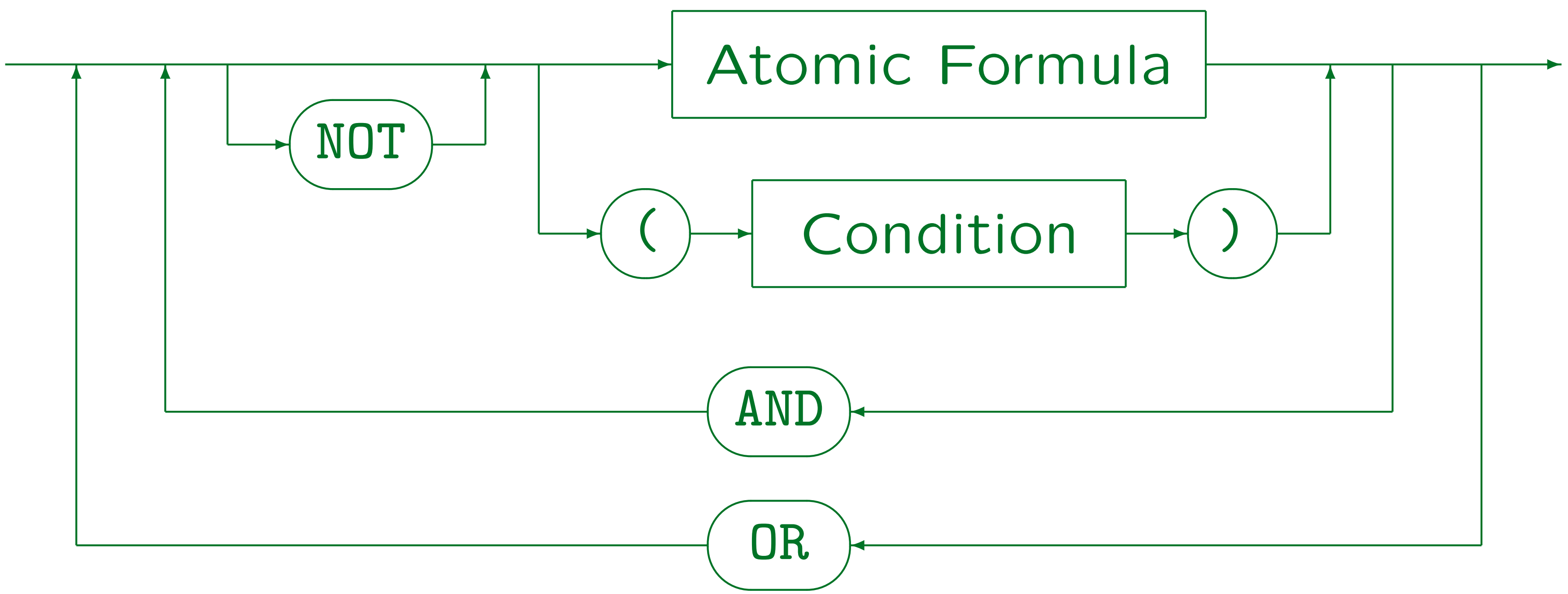
Conditions (3)

- `NOT` binds most strongly, i.e. it is applied only to the immediately following condition (atomic formula).
- Parentheses `( ... )` can be used to override the operator priorities (precedences, binding strengths).
- Sometimes, it might be clearer to use parentheses even if they are not necessary to enforce the right structure of the formula.

  However, beginners tend to use a lot of parentheses (probably because they are unsure about the operator priorities). This does not make the formula easier to understand.

Conditions (4)

Condition:



- SQL-92 allows "IS NOT TRUE", "IS FALSE" etc. after formulas (not supported in Oracle 8.0, SQL Server, DB2, MySQL, Access).

Conditions (5)

- AND and OR must take complete logical conditions (something that is true or false) on both sides.
- So the following is a syntax error although it is similar to natural language:

```
SELECT DISTINCT SID        Wrong!
FROM   RESULTS
WHERE  CAT = 'H' AND POINTS >= 9
AND    ENO = 1 OR 2
```

- Exception: ... BETWEEN ... AND ...

  Here the word AND does not denote the logical connective.

Comparisons (1)

Atomic Formula (Form 1):

- Comparison operators: `=`, `<>`, `<`, `>`, `<=`, `>=`.
- Comparison operators can be used for numbers as well as for strings, e.g.: `POINTS >= 8`, `LAST < 'M'`.
- "Not equals" is written in standard SQL as "`<>`".

  Oracle, SQL Server, DB2, and MySQL understand also "`!=`" (Access does not accept this notation). "`^=`" works in Oracle and DB2, but not in SQL Server, Access, or MySQL.

Comparisons (2)

- Numbers are compared differently than strings, e.g. 3 < 20, but '3' > '20'.

  String comparison is done character by character until the outcome is clear. In this case, "3" comes alphabetically after "2", therefore the rest of the string is not important.

- According to the SQL-92 standard, it is an error to compare strings with numbers, e.g. 3 > '20'.

  The two compared values must be of compatible types: All numeric types are compatible, and all string types are compatible, but numeric types are not compatible with string types.

Comparisons (3)

- Comparing a string with a number should be avoided, since the outcome is very system dependent:
  - ◇ SQL-92, DB2, and Access produce a type error.
  - ◇ Oracle, MySQL, and SQL Server convert the string to a number and do a numeric comparison.

    If the string is not of numeric format, MySQL simply converts it to 0. E.g. `0 = 'abc'` is true in MySQL. In Oracle and SQL Server, one gets an error if the string is not of numeric format. This might be a runtime error if the string is a column value.
  - ◇ However, if a column is compared with a constant, SQL server uses the column type.

    Aggregate functions have still higher priority than columns.

String Comparisons (1)

- The outcome of comparing (=, <>, <, <=, >, >=) two character strings may depend on the DBMS.

  Or settings within the DBMS.

- The SQL-92 standard defines the notion of "collation sequences" (or "collations") which determine
  - for any pair X and Y of characters, whether $X < Y$, $X = Y$, or $X > Y$, and
  - whether the blank-padded semantics (`PAD SPACE`) or the non-padded semantics (`NO PAD`) is used.

String Comparisons (2)

- `'a' < 'b'` etc., and `'A' < 'B'` etc. can be expected.
- But the systems differ in the comparison of uppercase and lowercase characters. The defaults are:
  - ◇ In Oracle all uppercase characters come before all lowercase characters (ASCII), e.g. `'Z' < 'a'`.
  - ◇ In DB2, uppercase and lowercase characters are interleaved, e.g.: `'a' < 'A'`, `'A' < 'b'`.
  - ◇ SQL Server, MS Access, and MySQL are case-insensitive, e.g.: `'a' = 'A'`.

String Comparisons (3)

- It might be possible to change this, but e.g. only during installation (SQL Server), or during database creation (Oracle, DB2).
- When the order (<, =, >) of every two characters is known, the comparison of strings of the same length is clear:
  - ◇ The system compares character by character, the first comparison which does not give "=" determines the result.

    Actually, DB2 makes two passes: It first compares the character "weights", and if there is no difference, also the character codes.

String Comparisons (4)

- For strings of different lengths, there are
  - ◇ Non-Padded Comparison Semantics:

    E.g. `'a' < 'a '`.

    Strings are compared character by character. When one string ends and no difference was found, the shorter string is considered less than the longer one.

  - ◇ Blank-Padded Comparison Semantics:

    E.g. `'a' = 'a '`.

    The shorter string is filled with `' '` before the comparison.

String Comparisons (5)

- DB2, SQL Server, Access, and MySQL use the blank-padded semantics (at least by default).
- Oracle uses the nonpadded semantics if at least one operand of a comparison has type VARCHAR2.

Oracle has introduced a type VARCHAR2(n). It is currently synonymous to VARCHAR(n), but Oracle intends to change the comparison semantics for VARCHAR, while the semantics for VARCHAR2 will remain stable. String literals (constants) in the query have type CHAR(n). E.g. a comparison of CHAR(10) and CHAR(20) columns can possibly yield "true" as can a comparison of these columns with, e.g., 'abc'. But CHAR(10) and VARCHAR(20) can only be equal if the VARCHAR happens to be of length 10. Trailing spaces in VARCHAR2-columns can be quite annoying: They are not visible in the output, but the comparison does not work.

String Comparisons (6)

- If the system uses a case-sensitive semantics, one can get a case-insentive comparison by converting both sides e.g. to uppercase:

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  UPPER(EMAIL) = UPPER('xyz@hotmail.com')
```

- UPPER works in SQL-92, Oracle, SQL Server, DB2, MySQL. In Access, use UCASE.

  UCASE works also in DB2 and MySQL. The book by Chamberlin about DB2 lists only UCASE.

String Comparisons (7)

- The opposite case (case-sensitive comparison with a case-insensitive system) is more difficult.

  But also much more seldom required.

- E.g. in MySQL, one can convert a string to a binary string in order to get case-sensitive comparison:

```
BINARY EMAIL = 'xyz@hotmail.com'
```

- The same trick works in SQL Server:

```
  CAST(EMAIL AS VARBINARY(255))
= CAST('...' AS VARBINARY(255))
```

String Comparisons (8)

- If one suspects that trailing spaces make a comparison fail, one can make them visible in this way:

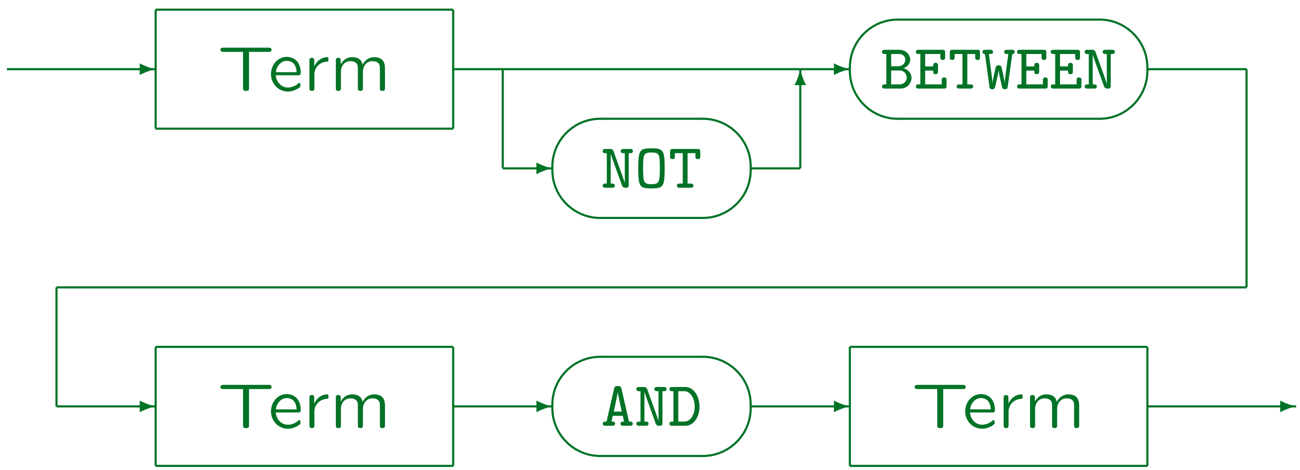
```
SELECT '"' || LAST || '"' AS LAST_NAME
FROM   STUDENTS
```

- One can also remove trailing spaces:
  - ◇ `TRIM(TRAILING ' ' FROM LAST)`
    in SQL-92 (works in MySQL)

    This syntax is not supported in Oracle, DB2, SQL Server, Access.
  - ◇ `RTRIM(LAST)`
    in Oracle, DB2, SQL Server, MySQL, Access.

BETWEEN Conditions

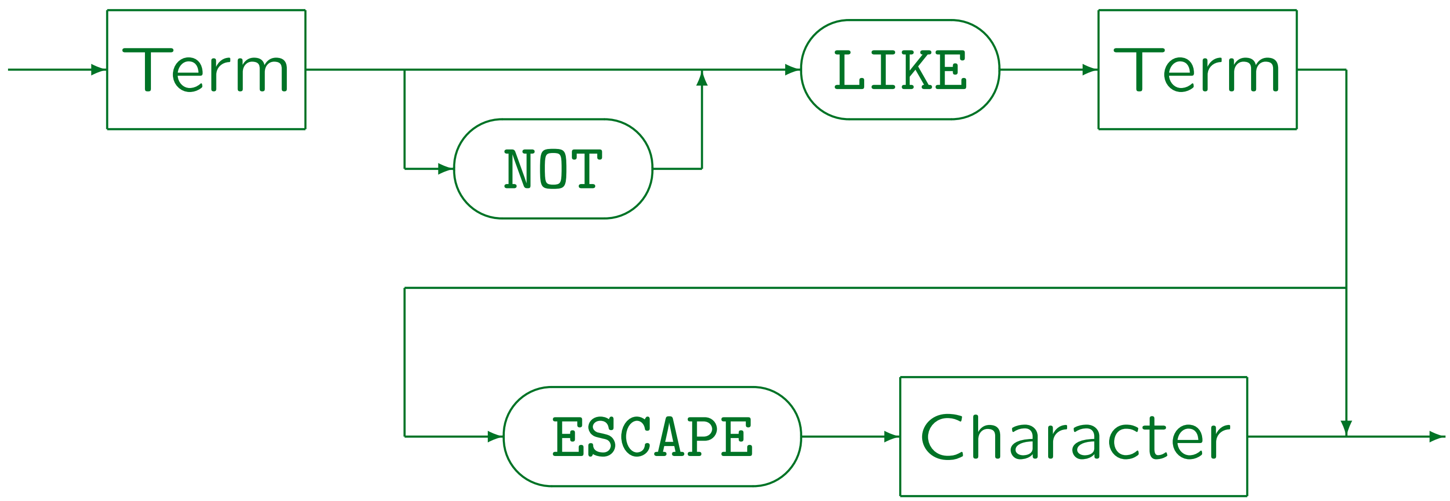
Atomic Formula (Form 2):



- `x BETWEEN y AND z` is equivalent to `x >= y AND x <= z`.
- E.g.: `POINTS BETWEEN 5 AND 8`

LIKE Conditions (1)

Atomic Formula (Form 3):



- E.g.: `EMAIL LIKE '%.pitt.edu'`

  Thius is true for all email addresses that end in "`pitt.edu`".

LIKE Conditions (2)

- The right argument is interpreted as pattern.

  In SQL-86 and DB2, it must be a string constant.

  In Oracle, SQL Server, Access, and MySQL, one can use any string valued term as pattern (especially also another column).

- "%" in the pattern matches any sequence of arbitrary characters (including the empty string).

- "_" matches any single character.

  SQL Server and Access support also character ranges, e.g. `[a-zA-Z]`. MySQL has an additional operator "`RLIKE`" (or "`REGEXP`") that accepts arbitrary regular expressions as patterns.

LIKE Conditions (3)

- To use the characters "%" and "_" without their special meaning in the pattern, an "escape" character is used.

  The escape character removes the special meaning of the following character. E.g. if "\" is the escape character, then "\%" matches only a percent sign, not an arbitrary string.

- The escape character must be declared, e.g.:

```
PROCNAME LIKE '\_%' ESCAPE '\'
```

  This gives all procedure names starting with an "_".

  In MySQL, if no escape character is explicitly declared, "\" is the default escape character. However, this violates the SQL-92 standard.

LIKE Conditions (4)

- `LIKE` uses the non-padded semantics.

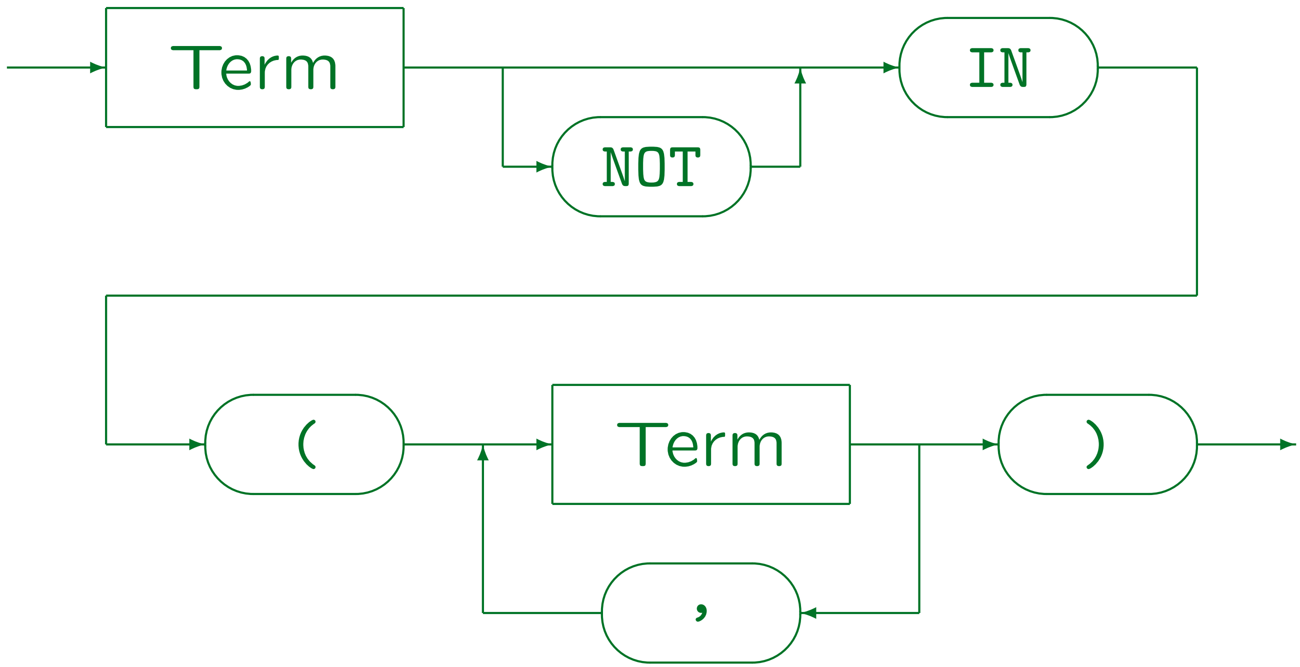
  Oracle, DB2, MySQL, and Access use the non-padded semantics as required by the SQL-92 standard. Note that MySQL removes trailing spaces when strings are stored. All systems fill values with blanks if the column is declared as fixed-length character string.
  In SQL Server, if the stored string contains more spaces at the end than the pattern, it might still match. If the pattern contains more spaces, the match fails. With the Unicode national character set types, the strict non-padded semantics is used.

- E.g. `'a' = 'a '` might be true (in some DBMS), but `'a' LIKE 'a '` is surely false.

- The case sensitivity is the same as for ordinary comparisons.

IN Conditions (1)

Atomic Formula (Form 4):



IN Conditions (2)

- E.g. `CAT IN ('M', 'F')`
- This is equivalent to

```
CAT = 'M' OR CAT = 'F'
```

- The SQL-86 standard allowed only constants in the enumeration of values.

  SQL-92, Oracle, SQL Server, and DB2 allow arbitrary terms, but it is normally better style to use `OR` if the set is not an enumeration of constants.

- Note that although in mathematics, "`(...)`" are used for intervals, here they mean "set".

A bit of Logic (1)

- Conditions used in the `WHERE`-clause are formulas of tuple calculus, which is a variant of predicate logic.
- Predicate logic is studied for about 100 years in mathematics and philosophy.
- Some basic knowledge of logic can actually help in query formulation.
- Here, the notions "inconsistent", "tautology", "implied", and "equivalent" are introduced, as well as some concrete equivalences for the propositional connectives `AND`, `OR`, `NOT`.

A bit of Logic (2)

- A condition is inconsistent if it can never be satisfied, i.e. is always false, no matter what the database state is and no matter which tuples are assigned to the tuple variables.
- E.g., no matter what row stands `R` for, `R.ENO` cannot be two different values at the same time:

  `R.ENO = 1 AND R.ENO = 2` **Wrong!**

- An inconsistent condition as `WHERE`-clause means that the query will never return any result rows.

A bit of Logic (3)

- Database management systems like Oracle do not give warnings for inconsistent conditions.

  Actually, it can be proven that it is impossible to develop an algorithm that detects all inconsistent conditions (if also subqueries or arithmetic operations are allowed).

- The other extreme is a tautology, i.e. a condition that is always true, e.g.:

```
R.ENO < 3  OR  R.ENO > 2
```

- Obviously, such conditions are not useful.

A bit of Logic (4)

- A condition A implies a condition B if, whenever A is true, also B is true.

  The implied condition B is weaker than condition A that implies it. A set of conditions $\{A_1, \ldots, A_n\}$ implies a condition B if, whenever A_1 to A_n are all true, also B is true.

- E.g. "`R.ENO = 2`" implies "`R.ENO <> 1`".

- Therefore, the condition

  ```
  R.ENO = 2  AND  R.ENO <> 1
  ```

  can be safely simplified to `R.ENO = 2`.

  The second part gives nothing new.

A bit of Logic (5)

- Two conditions are called (logically) equivalent if they always yield the same truth value.

  I.e. A and B are equivalent if for all database states and all assignments of rows to the tuple variables, if A is true, then B is true, and if A is false, then B is false. Equivalence means implication in both directions.

- E.g. it is not important whether one writes

```
CAT = 'H' AND ENO = 1
```

or vice versa

```
ENO = 1 AND CAT = 'H'
```

A bit of Logic (6)

- For the correctness of a query, it is not important which one out of several logically equivalent formulation one chooses.
- Of course, some formulations are more complicated than others, and one should choose a simple one.

  For instance, although adding an implied condition as shown above does not change the correctness of the query, points might be taken off in the exam for unnecessary complications.
- Modern DBMSs have good optimizers, such that simple equivalences like A **AND** B vs. B **AND** A are not important for the runtime of a query.

A bit of Logic (7)

- More complicated equivalences might not be detected by the query optimizer, e.g. writing

  `ENO - 2 = 0`

  might prevent that a special access structure for finding rows quickly (B-tree index) is used, which would have been used for the logically equivalent condition

  `ENO = 2`

- However, one gets the same answer in both cases, only the first query might run slightly longer.

Some Equivalences (1)

- A **AND** B $\equiv$ B **AND** A

  This is called commutativity. It holds also for **OR**.

- A **AND** (B **AND** C) $\equiv$ (A **AND** B) **AND** C

  This is called associativity. It means that no parentheses are necessary if one has a sequence of conditions all connected with **AND**. The associative law also holds for **OR**.

- A **AND** (B **OR** C) $\equiv$ (A **OR** B) **AND** (A **OR** C)

  This is the distribution law. It holds also for **AND** and **OR** exchanged.

- **NOT** (**NOT** A) $\equiv$ A

  This means that double negation cancels out.

Some Equivalences (2)

- NOT(A AND B) $\equiv$ (NOT A) OR (NOT B)

  This is De Morgan's Law. It holds also with AND and OR exchanged.

- A AND A $\equiv$ A

  It makes no sense to repeat a condition. This holds also for OR.

- NOT X < Y $\equiv$ X >= Y

  The comparison operators always come in complementary pairs, and it is not necessary to use NOT directly in front of such a condition. Together with De Morgan's law and the double negation rule, one can eliminate NOT from conditions (that use only the six comparison operators). But this might not always make the condition simpler.

Some Equivalences (3)

- X `=` Y $\equiv$ Y `=` X (symmetry)
- X `<` Y $\equiv$ Y `>` X

  And the same for `<=` and `>=`.
  Also, X `<=` Y is equivalent to X `<` Y `OR` X `=` Y.

- X `=` Y `AND` Y `=` Z implies X `=` Z (transitivity)

  And the same for `<`, `<=`, `>`, `>=`. When A implies B, the formulas A and A `AND` B are equivalent. Thus, X `=` Y `AND` Y `=` Z is equivalent to X `=` Y `AND` Y `=` Z `AND` X `=` Z Since certain equality conditions can be evaluated by using an index, it makes sense for a query optimizer to compute implied such conditions.

- X `=` X is a tautology if X cannot be null.

  SQL uses a three-valued logic, see below.

Exercise

- Is there any problem with this query? The task is to list all students who solved an exercise about SQL and an exercise about relational algebra.

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S, RESULTS R,
       EXERCISES E1, EXERCISES E2
WHERE  S.SID = R.SID
AND    R.CAT = E1.CAT AND R.ENO = E1.ENO
AND    R.CAT = E2.CAT AND R.ENO = E2.ENO
AND    E1.TOPIC = 'SQL'
AND    E2.TOPIC = 'Rel. Alg.'
```

Three-Valued Logic (1)

- Consider the following query:

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  EMAIL = 'xyz@acm.org'
```

- What happens if a course has a null value in the column `EMAIL`? It is not printed.
- But it also does not appear in the result of this query (because the value is unknown):

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  NOT (EMAIL = 'xyz@acm.org')
```

Three-Valued Logic (2)

- The condition

  `EMAIL = 'xyz@acm.org'`

  does not evaluate to false if `EMAIL` is null, since then the row would appear in the negated query.

  Of course, it also does not yield true.

- SQL uses a three-valued logic for treating null values. The three truth values are **`true`**, **`false`**, and **`unknown`**.

  Instead of "**`unknown`**", one also often reads "**`null`**".

Three-Valued Logic (3)

- The idea is that tuples which have a null value in an attribute which is important for the query should be "filtered out" — they should not influence the query result.
- The real attribute value is unknown or does not exist, so saying that the result of a comparison with a null value is true or false is equally wrong.
- In SQL, a comparison with a null value always yields the third truth value "`unknown`".

Three-Valued Logic (4)

- A result row is printed only if the WHERE-condition evaluates to "true".
- Thus, the following query gives the empty result:

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  EMAIL = null
```

Actually, the query is illegal in SQL-92, and DB2 refuses it. Oracle, SQL Server, Access, and MySQL accept it and print the empty result.

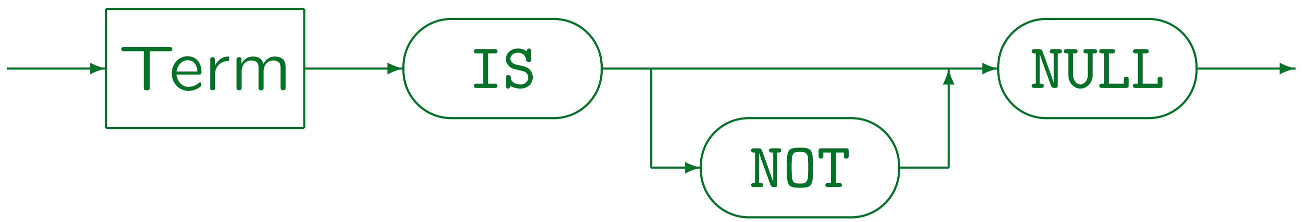
- "AND" / "OR" forward the truth value "unknown", unless the result is clear:
  E.g. "true OR unknown = true".

Three-Valued Logic (5)

P	Q	NOT P	P AND Q	P OR Q
false	false	true	false	false
false	unknown	true	false	unknown
false	true	true	false	true
unknown	false	unknown	false	unknown
unknown	unknown	unknown	unknown	unknown
unknown	true	unknown	unknown	true
true	false	false	false	true
true	unknown	false	unknown	true
true	true	false	true	true

Test for Null (1)

Atomic Formula (Form 5):



- E.g. `EMAIL IS NULL`
- The test for a null value can only be done in this way.

  "`EMAIL = NULL`" does not give the expected result in Oracle and SQL Server, it is a syntax error in SQL-92 and DB2.
  In SQL Server 7, "`EMAIL = NULL`" works after the command "`SET ANSI_NULLS OFF`" (then a two-valued logic is used).

Test for Null (2)

- Exercise: The following query prints all students with an email address in the domain "`.pitt.edu`":

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  EMAIL IS NOT NULL
AND    EMAIL LIKE '%.pitt.edu'
```

  Is the test for null necessary?

- `CHECK`-integrity constraints are satisfied if the condition evaluates to the third truth value "unknown".

  They are only violated if the condition evaluates to false.

Problems of Null Values (1)

- For those accustomed to working with a two-valued logic (all of us), null values can sometimes lead to surprises: Some standard logical equivalences do not hold in SQL.
- E.g. if students with an email address in the domain "`.pitt.edu`" are counted, and students with an outside email address, one would normally assume to get all students.
- But this is not true in SQL — those with a null value in the `EMAIL` column are not counted.

Problems of Null Values (2)

- E.g. `X = X` evaluates to “unknown”, not to “true” if `X` is null.
- Since the null value is used with different meanings, there can be no satisfying semantics for a query language.

  E.g. the meaning “value exists, but unknown” ($\exists X: \ldots$) would allow to use standard logical equivalences.

Terms with Null Values (1)

- Data type functions will normally return null if one of their arguments is null. E.g. if `A` is null, `A+B` will be null.

  In Oracle, `A || B` (the concatenation of strings `A` and `B`) returns `B` if `A` is null (violates the SQL-92 standard).

- `NULL` by itself is not a term (expression), although it can be used in many contexts that otherwise require a term.

Terms with Null Values (2)

- `NULL` has no type, so at least we need a context in which the type is clear:
  - In SQL-92 and DB2, `CAST(NULL AS type)` gives a null value of the specified type.
  - In Oracle, `NULL` often can be used as a term, but e.g. this gives an error:

    ```
    select 1 from dual union select null from dual
    ```

    One must write `TO_NUMBER(null)`.
  - In SQL Server, Access, and MySQL "`NULL`" is handled like a normal term (with arbitrary type).

Overview

1. Lexical Syntax
2. Tuple Variables, Joins
3. Terms, Conditions, Logic, Null Values
4. Subqueries, Nonmonotonic Constructs
5. Aggregations
6. Union, ORDER BY, Outer Join

Example Database (again)

STUDENTS			
SID	FIRST	LAST	EMAIL
101	Ann	Smith	...
102	Michael	Jones	(null)
103	Richard	Turner	...
104	Maria	Brown	...

EXERCISES			
CAT	ENO	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

RESULTS			
SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

Nonmonotonic Behaviour (1)

- SQL queries using only the constructs introduced above compute monotonic functions on the existing tables: If further rows are inserted, one gets at least the same answers as before, and maybe more.
- However, not all queries behave monotonically in this way: E.g. print students who have not yet submitted any homework.

  Currently Maria Brown would be a correct answer. But if a homework result were inserted for her, she would no longer qualify.
- Therefore, this query cannot be formulated with the SQL constructs that were introduced so far.

Nonmonotonic Behaviour (2)

- In the natural language version of queries, formulations like "there is no", "does not exist" indicate nonmonotonic behaviour.
- Furthermore, "for all", "the minimal/maximal", also indicate nonmonotonic behaviour: In this case a violation of the "for all" condition must not exist.

  For some such queries, a formulation with `HAVING` might be natural.
- When formulating queries in SQL, it is important to check whether the query requires that certain tuples do not exist.

NOT IN (1)

- With IN ($\in$) and NOT IN ($\notin$) it is possible to check whether an attribute value appears in a set that is computed by another SQL query.
- E.g. students without any homework result:

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  SID NOT IN (SELECT SID
                   FROM   RESULTS
                   WHERE  CAT = 'H')
```

FIRST	LAST
Maria	Brown

NOT IN (2)

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

STUDENTS			
SID	FIRST	LAST	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

Result of Subquery
SID
101
101
102
102
103

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

NOT IN (3)

- It is possible to use DISTINCT in the subquery:

```
SELECT FIRST, LAST
FROM   STUDENTS
WHERE  SID NOT IN (SELECT DISTINCT SID   ?
                   FROM   RESULTS
                   WHERE  CAT = 'H')
```

- This is logically equivalent, and the effect on the performance depends on the data and the DBMS.

  I would expect that a reasonable optimizer knows that duplicates are not important in this case and that conversely writing DISTINCT might have the effect that the optimizer does not consider certain evaluation stragegies that do not really materialize the result of the subquery.

NOT IN (4)

- 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 on next page).

NOT IN (5)

- E.g. topics ("names") of homeworks that were already solved by at least one student:

```
SELECT TOPIC
FROM   EXERCISES
WHERE  CAT='H' AND ENO IN (SELECT ENO
                           FROM   RESULTS
                           WHERE  CAT='H')
```

- Exercise: Is there a difference to this query (with or without DISTINCT)?

```
SELECT DISTINCT TOPIC
FROM   EXERCISES E, RESULTS R
WHERE  E.CAT='H' AND E.ENO=R.ENO AND R.CAT='H'
```

NOT IN (6)

- In SQL-86, the subquery on the right-hand side of `IN` must have a single output column.

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

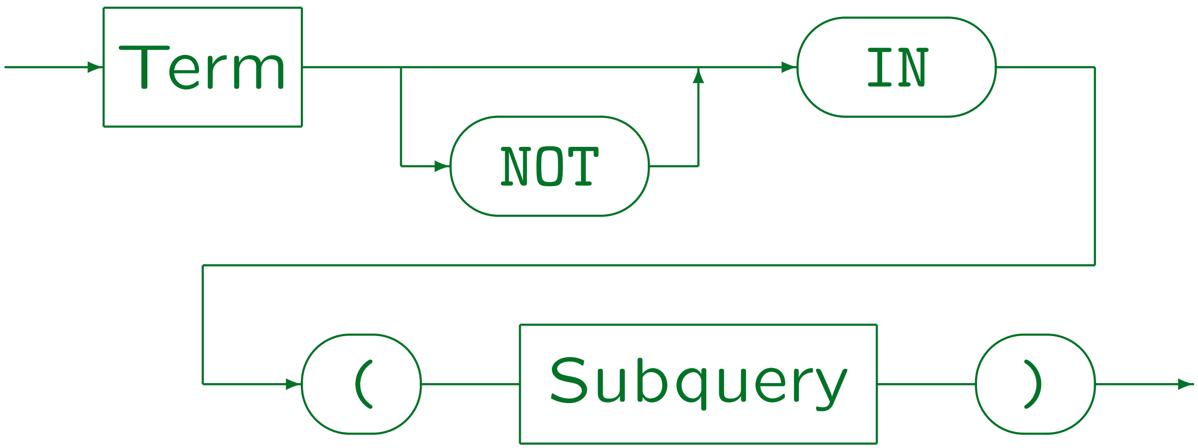
- In SQL-92, comparisons were extended to the tuple level, and therefore it is possible to write e.g.

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

  But is not very portable. E.g. SQL Server and Access do not support it (and MySQL does not permit any subqueries, see below). An `EXISTS` subquery (see below) might be better if one has to compare more than one column. Oracle and DB2 do allow `IN` with multiple columns.

NOT IN (7)

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 ($\texttt{Term}_1$, ..., $\texttt{Term}_n$). Then the subquery must result in a table with exactly n columns.
- MySQL does not support subqueries.
- The column names on the left and right hand side of IN do not have to match, but the data types must be compatible.

NOT EXISTS (1)

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

  This is actually also possible for IN subqueries, but there it is an unnecessary and unexpected complication (bad style).
- This means that the subquery has to be evaluated once for every assignment of values to the accessed tuple variables in the outer query. The subquery can be seen as parameterized.

NOT EXISTS (2)

- Students that have not submitted any homework:

```
SELECT FIRST, LAST
FROM   STUDENTS S
WHERE  NOT EXISTS (SELECT * FROM RESULTS R
                   WHERE  R.CAT = 'H'
                   AND    R.SID = S.SID)
```

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

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

NOT EXISTS (3)

- First, S points to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
101	Ann	Smith	···

- S.SID in the subquery is conceptually replaced by 101 and the following query is executed:

```
SELECT * FROM RESULTS R
WHERE  R.CAT = 'H'
AND    R.SID = 101
```

SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8

- The result is not empty. Thus, the NOT EXISTS condition in the outer query is not satisfied for this S.

NOT EXISTS (4)

- The same happens for the second row in STUDENTS. The subquery is executed for S.SID = 102:

```
SELECT * FROM RESULTS R
WHERE  R.CAT = 'H'
AND    R.SID = 102
```

SID	CAT	ENO	POINTS
102	H	1	9
102	H	2	9

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

NOT EXISTS (5)

- Finally, S points to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
104	Maria	Brown	···

- For S.SID = 104, the result of the subquery is empty:

```
SELECT * FROM RESULTS R
WHERE  R.CAT = 'H'
AND    R.SID = 104
```

no rows selected

- Thus, the NOT EXISTS condition is satisfied for this tuple S. Maria Brown is printed as the query result.

NOT EXISTS (6)

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

```
SELECT FIRST, LAST, R.ENO   Wrong!
FROM   STUDENTS S
WHERE  NOT EXISTS (SELECT * FROM RESULTS R
                   WHERE  R.CAT = 'H'
                   AND    R.SID = S.SID)
```

- 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 ($\sim$ block structure in Pascal).

NOT EXISTS (7)

- Subqueries that access variables from the outer query are called "correlated subqueries".

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

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

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

NOT EXISTS (8)

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

```
SELECT FIRST, LAST      Wrong!
FROM   STUDENTS S
WHERE  NOT EXISTS (SELECT * FROM RESULTS R
                   WHERE  CAT = 'H')
```

Here the join-condition in the subquery was forgotten, and it became a non-correlated subquery.

- If there is at least one homework entry in RESULTS, no matter for what student, the NOT EXISTS will be false, and the query result empty.

NOT EXISTS (9)

- Until now, for attribute references without tuple variable (“unqualified attribute name”), there had to be a unique tuple varible to which it can refer.
- For subqueries, SQL only requires 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 RESULTS R
                   WHERE  CAT = 'H'
                   AND    SID  = S.SID)
```

NOT EXISTS (10)

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

NOT EXISTS (11)

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

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

- References to `X` in the subquery mean `RESULTS X`. The variable declared in the outer query becomes "shadowed": It cannot be accessed in the subquery.

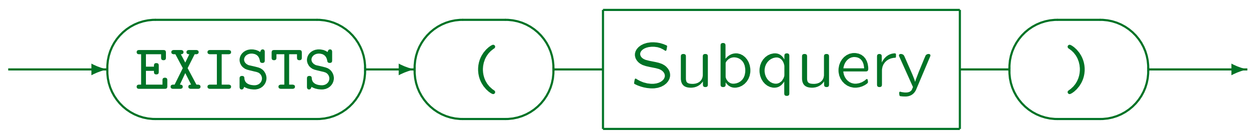
NOT EXISTS (12)

- It is legal to specify a SELECT-list in the subquery, but since for NOT EXISTS the returned columns do not matter, "SELECT *" should be used.
- Some authors say that in some systems SELECT null or SELECT 1 is actually faster than SELECT *.

  "SELECT null" is used by Oracle's programmers (in "catalog.sql"). But this does not work in DB2 (null cannot be used as a term here). Today, resonably good optimizers should know that the column values are not really needed, and the SELECT-list should not matter, not even for performance.

NOT EXISTS (13)

Atomic Formula (Form 7):



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

  [...] means that these parts are optional. SQL-92 also allows `UNION` (see below) in subqueries (as do Oracle, DB2, and SQL Server), SQL-86 does not (and Access really does not support it).

- `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 (14)

- It is possible to use EXISTS without negation.
- Who has submitted at least one homework?

```
SELECT SID, FIRST, LAST
FROM   STUDENTS S
WHERE  EXISTS (SELECT * FROM RESULTS R
               WHERE  R.SID = S.SID
               AND    R.CAT = 'H')
```

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

```
SELECT DISTINCT S.SID, FIRST, LAST
FROM   STUDENTS S, RESULTS R
WHERE  S.SID = R.SID AND R.CAT = 'H'
```

"For all" (1)

- Who got the best result for Homework 1?

```
SELECT FIRST, LAST, POINTS
FROM   STUDENTS S, RESULTS X
WHERE  S.SID = X.SID
AND    X.CAT = 'H' AND X.ENO = 1
AND    NOT EXISTS
       (SELECT * FROM RESULTS Y
        WHERE  Y.CAT = 'H' AND Y.ENO = 1
        AND    Y.POINTS > X.POINTS)
```

- I.e. a result `X` for Homework 1 is selected if there is no result `Y` for this exercise with more points than `X`.

"For all" (2)

- In mathematical logic, there are two quantifiers:
  - $\exists X\ (\varphi)$: There is an X that satisfies φ. (existential quantifier)
  - $\forall X\ (\varphi)$: For all X, φ is true. (universal quantifier)
- In tuple relational calculus, the maximal number of points for Homework 1 is expressed e.g. as follows:

$$\{\texttt{X.POINTS} \mid \texttt{X}\colon \texttt{RESULTS} \wedge \texttt{X.CAT} = \texttt{'H'} \wedge \texttt{X.ENO} = 1 \wedge \forall\, \texttt{Y}\ (\texttt{Y}\colon \texttt{RESULTS} \wedge \texttt{Y.CAT} = \texttt{'H'} \wedge \texttt{Y.ENO} = 1 \longrightarrow \texttt{Y.POINTS} \leq \texttt{X.POINTS})\}$$

"For all" (3)

- The pattern $\forall X(\varphi_1 \rightarrow \varphi_2)$ is very typical:
  For all X, if φ_1 is true, then φ_2 must be true.
- I.e. φ_2 must be true for all X that satisfy φ_1.

  Such a "bounded quantifier" is natural because tuple relational calculus requires that all queries are safe in the sense that values outside the database state do not influence the truth values. This is important in order to make sure that the truth value of a formula for a database state can be determined with finite effort (it suffices to look at all values that really occur in the database relations). The formal definition is a bit complex. SQL solves this problem by immediately binding every tuple variable to a single relation. However, because of this restriction, it needs `UNION`.
- $\varphi_1 \rightarrow \varphi_2$ is logically equivalent to $\neg\varphi_1 \vee \varphi_2$.

"For all" (4)

- SQL has only an existential quantifier ("EXISTS"), but not a universal quantifier.

  However, see ">= ALL" below.

- This is no problem, because "$\forall X\,(\varphi)$" is equivalent to "$\neg\exists X\,(\neg\varphi)$".

  "φ is true for all X" is the same as "φ is false for no X".

- Thus, one type of quantifier suffices.
- Exercise: Show that $\forall X(\varphi_1 \rightarrow \varphi_2)$ is equivalent to $\neg\exists X(\varphi_1 \wedge \neg\varphi_2)$.

"For all" (5)

- The above example is logically equivalent to:

$$\{\texttt{X.POINTS} \mid \texttt{X: RESULTS} \land \texttt{X.CAT} = \texttt{'H'} \land \texttt{X.ENO} = 1 \land \neg\exists\, \texttt{Y}\, (\texttt{Y: RESULTS} \land \texttt{Y.CAT} = \texttt{'H'} \land \texttt{Y.ENO} = 1 \land \texttt{Y.POINTS} > \texttt{X.POINTS})\}$$

- In SQL, this is written as:

```
SELECT X.POINTS
FROM   RESULTS X
WHERE  X.CAT = 'H' AND X.ENO = 1
AND    NOT EXISTS
       (SELECT * FROM RESULTS Y
        WHERE  Y.CAT = 'H' AND Y.ENO = 1
        AND    Y.POINTS > X.POINTS)
```

Nested Subqueries

- Subqueries can be nested to any reasonable depth.
- List the students who solved all homeworks:

```
SELECT FIRST, LAST
FROM   STUDENTS S
WHERE  NOT EXISTS
       (SELECT * FROM EXERCISES E
        WHERE  CAT = 'H'
        AND    NOT EXISTS
               (SELECT * FROM RESULTS R
                WHERE  R.SID = S.SID
                AND    R.ENO = E.ENO
                AND    R.CAT = 'H'))
```

Common Errors (1)

Exercises:

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

```
SELECT DISTINCT S.SID, FIRST, LAST
FROM   STUDENTS S, RESULTS R
WHERE  S.SID <> R.SID AND R.CAT = 'H'
```

- Would this query find exercises that were not yet solved?

```
SELECT DISTINCT E.CAT, E.ENO
FROM   EXERCISES E, RESULTS R
WHERE  E.CAT = R.CAT AND E.ENO = R.ENO
AND    R.SID IS NULL
```

Common Errors (2)

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

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

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

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

Common Errors (3)

- Does this query compute the student with the best result for Homework 1?

```
SELECT DISTINCT S.FIRST, S.LAST, X.POINTS
FROM   STUDENTS S, RESULTS X, RESULTS Y
WHERE  S.SID = X.SID
AND    X.CAT = 'H' AND X.ENO = 1
AND    Y.CAT = 'H' AND Y.ENO = 1
AND    X.POINTS > Y.POINTS
```

- If not, what does it compute?

Common Errors (4)

- As mentioned above, using a non-correlated subquery with NOT EXISTS is normally an error.
- Does this also apply in this case (there is a join condition in the subquery)?

```
SELECT FIRST, LAST
FROM   STUDENTS S
WHERE  NOT EXISTS              Wrong!
       (SELECT *
        FROM   RESULTS R, STUDENTS S
        WHERE  R.SID = S.SID
        AND    R.CAT = 'H' AND R.ENO = 1)
```

Common Errors (5)

- What is the error in this query? It is supposed to find students that have neither submitted a homework nor participated in an exam.

```
SELECT FIRST, LAST          Wrong!
FROM   STUDENTS S
WHERE  SID NOT IN (SELECT SID
                   FROM   EXERCISES)
```

- This query is syntactically correct SQL. Why?
- What is the output of the query?

  Under the assumption that `EXERCISES` is not empty.

Common Errors (6)

- Is there any problem with this query?

  The task is to list all students who did not yet actively participated in the course, i.e. neither submitted a homework nor took the exam.

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S, RESULTS R
WHERE  S.SID = R.SID
AND    NOT EXISTS (SELECT *
                   FROM   RESULTS R
                   WHERE  S.SID = R.SID)
```

ALL, ANY, SOME (1)

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

```
SELECT S.FIRST, S.LAST, X.POINTS
FROM   STUDENTS S, RESULTS X
WHERE  S.SID=X.SID AND X.CAT='H' AND X.ENO=1
AND    X.POINTS >= ALL (SELECT Y.POINTS
                        FROM   RESULTS Y
                        WHERE  Y.CAT = 'H'
                        AND    Y.ENO = 1)
```

ALL, ANY, SOME (2)

- The following is logically equivalent to the above query:

```
SELECT S.FIRST, S.LAST, X.POINTS
FROM   STUDENTS S, RESULTS X
WHERE  S.SID=X.SID AND X.CAT='H' AND X.ENO=1
AND    NOT X.POINTS < ANY (SELECT Y.POINTS
                           FROM   RESULTS Y
                           WHERE  Y.CAT = 'H'
                           AND    Y.ENO = 1)
```

- Again, "for all" can be replaced by "not exists not".

  Of course, also conversely "exists" is equivalent to "not for all not".

ALL, ANY, SOME (3)

- This construct is not strictly necessary, since e.g.

  T_1 `< ANY (SELECT` T_2 `FROM ... WHERE ...)`

  is equivalent to

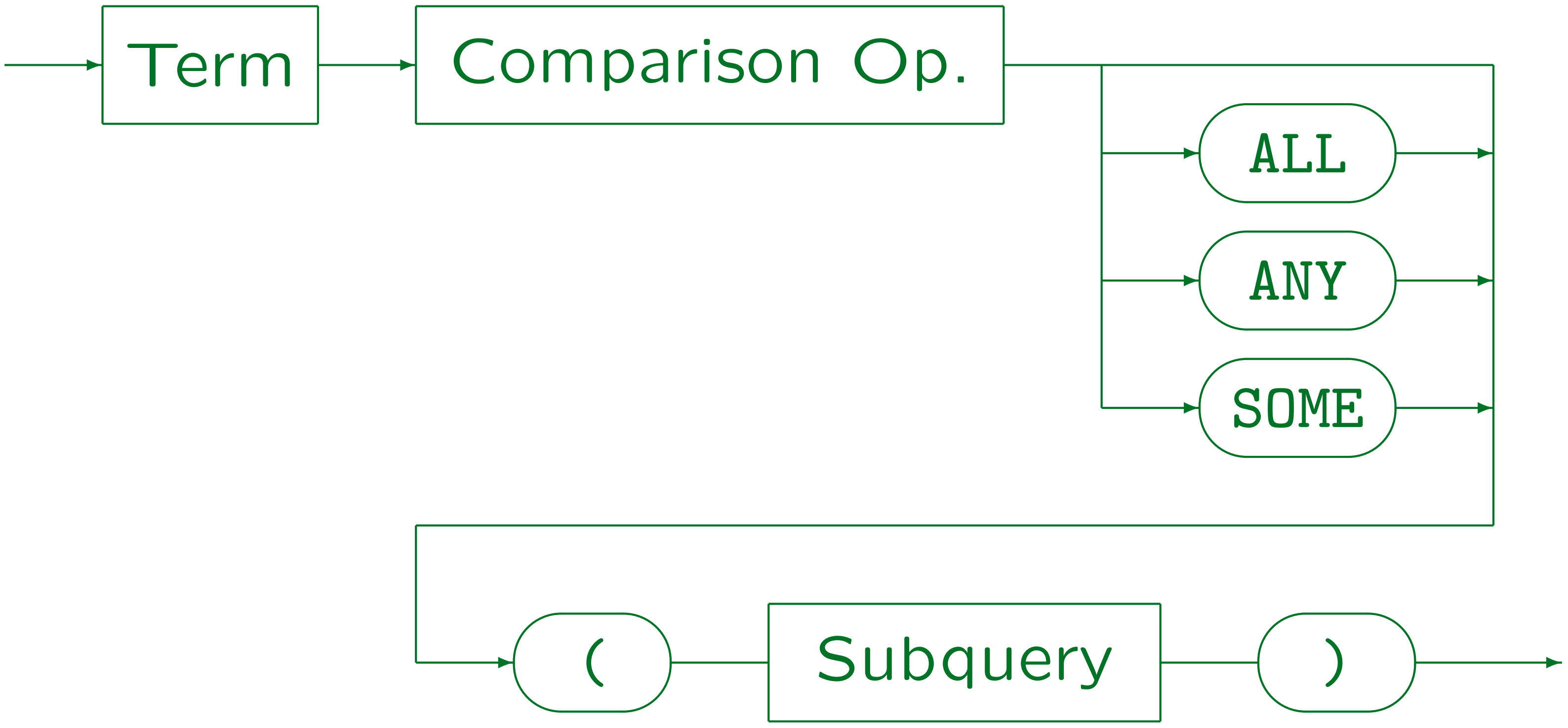
  `EXISTS (SELECT * FROM ... WHERE ... AND` T_1 `<` T_2`)`

  This requires that T_1 explicitly mentions a tuple variable which is not redeclared in the subquery (so that the meaning of T_1 is not changed by moving it into the subquery).

- E.g. Oracle internally does such transformations so that the query optimizer does not have to handle too many different cases (syntactic variants).

ALL, ANY, SOME (4)

Atomic Formula (Form 8):



ALL, ANY, SOME (5)

Syntactic Remarks:

- `ANY` and `SOME` are synonyms.
- "`x IN S`" is equivalent to "`x = ANY S`".
- The subquery must have a single result 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, SQL Server, and Access do not support tuple comparisons.

- If none of the keywords `ALL`, `ANY`, `SOME` are present, the subquery must yield at most one row.

  Since there is also only one column, this means the subquery gives a single data value. If the subqery result is empty, the null value is used.

Single Value Subqueries (1)

- Who got full points for Homework 1?

```
SELECT S.FIRST, S.LAST
FROM   STUDENTS S, RESULTS R
WHERE  S.SID=R.SID AND R.CAT='H' AND R.ENO=1
AND    R.POINTS = (SELECT MAXPT
                   FROM   EXERCISES
                   WHERE  CAT='H' AND ENO=1)
```

- It is only possible to leave out ANY/ALL when the subquery is guaranteed to return at most one row.

  In the example, a key of EXERCISES is specified. But in general, this may depend on the data. The query might run during testing, but later give an error. Use constraints to ensure that the necessary assumtions are really satisfied.

Single Value Subqueries (2)

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

```
(SELECT MAXPT FROM ...) = R.POINTS
```

- In Oracle8 and SQL86, the subquery must be on the right hand side.
- One can even do further computations with the result of a subquery, e.g. (not in SQL86, Oracle8):

```
R.POINTS >= (SELECT MAXPT FROM ...) * 0.9
```

Single Value Subqueries (3)

- If the subquery has an empty result, the null value is used instead.
- E.g. this is a strange way to ask for students that have not yet solved Homework 1:

```
SELECT FIRST, LAST
FROM   STUDENTS S
WHERE  (SELECT 1                    Bad Style!
        FROM   RESULTS R
        WHERE  R.SID = S.SID
        AND    R.CAT = 'H' AND R.ENO = 1) IS NULL
```

- In SQL86 and Oracle8, this is a syntax error.

Subqueries under FROM (1)

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

  In relational algebra, whereever one can write a relation name, one can also write a subquery (relational algebra expression).
- Subqueries under `FROM` are really needed only seldom, and might make the query more complex.

Subqueries under FROM (2)

- Subqueries under FROM are needed e.g. for nested aggregations, see below.
- In the following example, the join of RESULTS and EXERCISES is computed in a subquery (that might result from a view definition, see below):

```
SELECT X.SID, ROUND(X.POINTS*100/X.MAXPT) AS PCT
FROM   (SELECT E.CAT, E.ENO, R.SID, R.POINTS
               E.MAXPT
        FROM   EXERCISES E, RESULTS R
        WHERE  E.CAT=R.CAT AND E.ENO=R.ENO) X
WHERE  X.CAT = 'E' AND X.ENO = 1
```

Subqueries under FROM (3)

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

```
FROM (...) X(CATEGORY, EX_NO, ...)
```

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

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

Subqueries under FROM (4)

- Inside the subquery, one cannot refer to other tuple variables introduced in the same FROM-clause:

```
SELECT S.FIRST, S.LAST, X.ENO, X.POINTS    Wrong!
FROM   STUDENTS S, (SELECT R.ENO, R.POINTS
                    FROM   RESULTS R
                    WHERE  R.CAT = 'H'
                    AND    R.SID = S.SID) X
```

- In addition, subqueries under FROM should only be used if needed. They can make queries much more difficult to understand.

Subqueries under FROM (5)

- A view declaration stores a query under a name in the database:

```
CREATE VIEW HW_POINTS AS
   SELECT   S.FIRST, S.LAST, R.ENO, R.POINTS
   FROM     STUDENTS S, RESULTS R
   WHERE    S.SID=R.SID AND R.CAT = 'H'
```

- Views can be used in queries like stored tables:

```
SELECT ENO, POINTS
FROM   HW_POINTS
WHERE  FIRST='Michael' AND LAST='Jones'
```

- A view is an abbreviation for the subquery (macro).

Subqueries under FROM (6)

- When a view used in a query, the DBMS simply replaces the view name by the query it stands for.

  Views existed already in SQL-86. However, since SQL-86 did not contain subqueries under `FROM`, there were complex restrictions for using views.

- By using views, one can build complex queries step by step.

  If the optimizer is not very good, it might be possible that a query built in this way runs slightly slower than a single "monolithic" query. However, there should be no difference to using subqueries under `FROM`. A performance improvement is only possible if one can formulate the query without such subqueries.

Overview

Aggregations (1)

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

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

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

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

- Aggregation functions are often used for statistical evaluations (e.g. average).

Aggregations (2)

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

  Additional aggregation functions in certain systems:
  Oracle 8i: CORR (correlation, works on a set of pairs),
  COVAR_POP, COVAR_SAMP, linear regression functions,
  STDDEV, STDDEV_POP, STDEV_SAMP, VARIANCE, VAR_POP, VAR_SAMP.
  DB2: CORRELATION, COUNT_BIG, COVARIANCE, regression functions, STDDEV,
  VARIANCE.
  SQL Server: VAR, VARP, STDEV, STDEVP.
  Access: VAR, VARP, STDEV, STDEVP, FIRST, LAST.
  MySQL: STD. However, MySQL supports DISTINCT only for COUNT.

- Any commutative and associative binary operator with a neutral element can be extended to work on sets. E.g. sum is the set-version of $+$.

Aggregations (3)

- Some aggregation functions are sensitive to duplicates (e.g. sum), others are not (e.g. minimum).

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

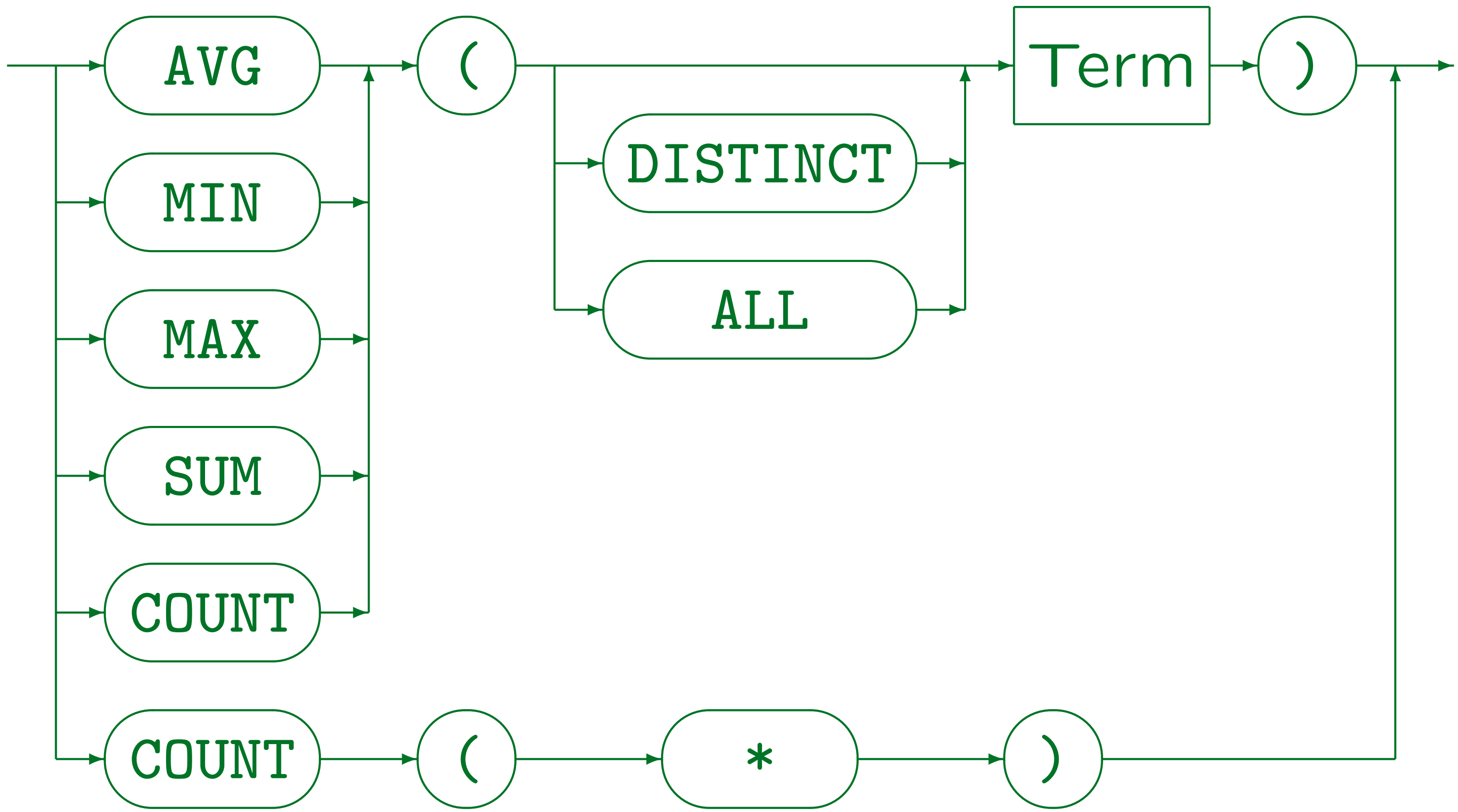
- In SQL, one can request duplicate elimination (input is a set) or not (input is a multiset).

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

- `SUM(DISTINCT X)` and `AVG(DISTINCT X)` are most likely an error. Some students mix up `SUM` and `COUNT`.

Syntax

Aggregation Term:



- MySQL supports DISTINCT only for COUNT (and does not understand ALL).

Null Values in Aggregations

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

  Otherwise, the actual attribute value is not important for `COUNT`, and one probably should use `COUNT(*)`. Of course, if duplicates are eliminated as in `COUNT(DISTINCT CAT)`, the attribute is obviously important.

Empty Aggregations

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

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

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

  Alternative: Use e.g. NVL(SUM(POINTS),0) in Oracle to replace the null.

Aggregation Queries

- There are three different types of queries in SQL:
  - ◇ Queries without aggregation functions and without `GROUP BY` and `HAVING`: See above.
  - ◇ Queries with aggregation functions under `SELECT`, but no `GROUP BY` (called "simple aggregations" below): Result is always exactly one row.
  - ◇ Queries with `GROUP BY`.
- Each type has different syntax restrictions and is evaluated in a different way.

Evaluation (1)

- First, the `FROM`-clause is evaluated.

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

- Second, the `WHERE`-clause is evaluated.

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

- Third, if there is no aggregation, `GROUP BY`, and `HAVING`, the `SELECT`-clause is evaluated by printing the values of the terms/scalar expressions in the `SELECT`-list for every remaining tuple combination.

Evaluation (2)

- When the SELECT-list contains an aggregation term, and there is no GROUP BY, only a single output row is computed by applying the aggregation operators.
- Instead of printing the values of columns as usual, the values are added to a set/multiset that is the input to the aggregation function.

  If the SELECT-list contains multiple aggregations, multiple such sets must be managed.
- If no DISTINCT is used, the aggregated values can be incrementally computed without explicitly storing a temporary set of values (see next slide).

Evaluation (3)

- E.g. consider the query:

```
SELECT SUM(MAXPT), COUNT(*)
FROM   EXERCISES E
WHERE  CAT = 'H'
```

- This is evaluated as:

```
out1 = 0; out2 = 0;
foreach row E in EXERCISES do
    if E.CAT = 'H' then begin
        out1 = out1 + E.MAXPT;
        out2 = out2 + 1;
    end;
print out1, out2;
```

Syntax / Restrictions (1)

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

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

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

  > It is possible that aggregations are first applied to groups of rows, and then the result is input to another aggregation. E.g. what is the average over the total number of points students got for their homeworks? This is done with GROUP BY and subqueries (see below).

Syntax / Restrictions (2)

- Aggregations cannot be used in the WHERE-clause.

  The WHERE-condition is evaluated before aggregations are computed (it determines which tuples enter the aggregation). Conditions with aggregations can be specified under HAVING (see below). Of course, also subqueries nested in the WHERE-clause may contain aggregations.

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

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

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

  ```
  SELECT CAT, ENO, AVG(POINTS)
  FROM   RESULTS
  ```
  Wrong!

Syntax / Restrictions (3)

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

```
SELECT SID
FROM   RESULTS
WHERE  CAT = 'H' AND ENO = 1
AND    POINTS = MAX
```
Wrong! Wrong!

  Aggregations are also not allowed under `WHERE`.

- A subquery is required to find the student(s) with the best result for Homework 1 (see below).

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 average points for each homework:

```
SELECT   ENO, AVG(POINTS)
FROM     RESULTS
WHERE    CAT = 'H'
GROUP BY ENO
```

ENO	AVG(POINTS)
1	8
2	8.5

GROUP BY (2)

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

SID	CAT	ENO	POINTS
101	H	1	10
102	H	1	9
103	H	1	5
101	H	2	8
102	H	2	9

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

GROUP BY (3)

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

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

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

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

GROUP BY (4)

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

  Other attributes can be used under `SELECT` only inside aggregations.

- E.g. this is illegal:

```
SELECT   E.ENO, E.TOPIC, AVG(R.POINTS)   Wrong!
FROM     EXERCISES E, RESULTS R
WHERE    E.CAT='H' AND R.CAT='H' AND E.ENO=R.ENO
GROUP BY E.ENO
```

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

GROUP BY (5)

- Thus, one must group by `E.ENO` and `E.TOPIC`:

```
SELECT   E.ENO, E.TOPIC, AVG(R.POINTS)
FROM     EXERCISES E, RESULTS R
WHERE    E.CAT='H' AND R.CAT='H' AND E.ENO=R.ENO
GROUP BY E.ENO, E.TOPIC
```

E.ENO	E.TOPIC	AVG(POINTS)
1	Rel. Algeb.	8
2	SQL	8.5

- Adding `E.TOPIC` to the `GROUP BY` attributes does not change the groups, but now one can print it.

GROUP BY (6)

- Exercise: Is there any semantical difference between

```
SELECT   TOPIC, AVG(POINTS/MAXPT)
FROM     EXERCISES E, RESULTS R
WHERE    E.CAT='H' AND R.CAT='H' AND E.ENO=R.ENO
GROUP BY TOPIC
```

and the query which additionally groups by `E.ENO`, but does not print it?

```
SELECT   TOPIC, AVG(POINTS/MAXPT)
FROM     EXERCISES E, RESULTS R
WHERE    E.CAT='H' AND R.CAT='H' AND E.ENO=R.ENO
GROUP BY TOPIC, E.ENO
```

GROUP BY (7)

- GROUP BY is evaluated before the SELECT clause. Thus, one cannot refer to new attribute names:

```
SELECT   FLOOR((POINTS/MAXPT)*10+0.5) PCT_RANGE,
         COUNT(*)
FROM     EXERCISES E, RESULTS R
WHERE    E.CAT = R.CAT AND E.ENO = R.ENO
GROUP BY PCT_RANGE
```

Wrong!

- Oracle, SQL Server, DB2, MySQL, and Access support GROUP BY with arbitrary terms. The SQL92 standard permits GROUP BY only with column names.

  I.e. GROUP BY FLOOR(...) works in these systems.
  Portable alternative: Subquery under FROM or using a view.

GROUP BY (8)

- 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 $t.\texttt{A} = u.\texttt{A}$ and $t.\texttt{B} = u.\texttt{B}$. `GROUP BY B, A` means that two tuples t, u belong into the same group if $t.\texttt{B} = u.\texttt{B}$ and $t.\texttt{A} = u.\texttt{A}$.

- Note that it makes no sense to group by a key (if only one table is listed under `FROM`): Then every group will consist of only a single row.

- In the same way, `GROUP BY` is not useful if there can be only a single group.

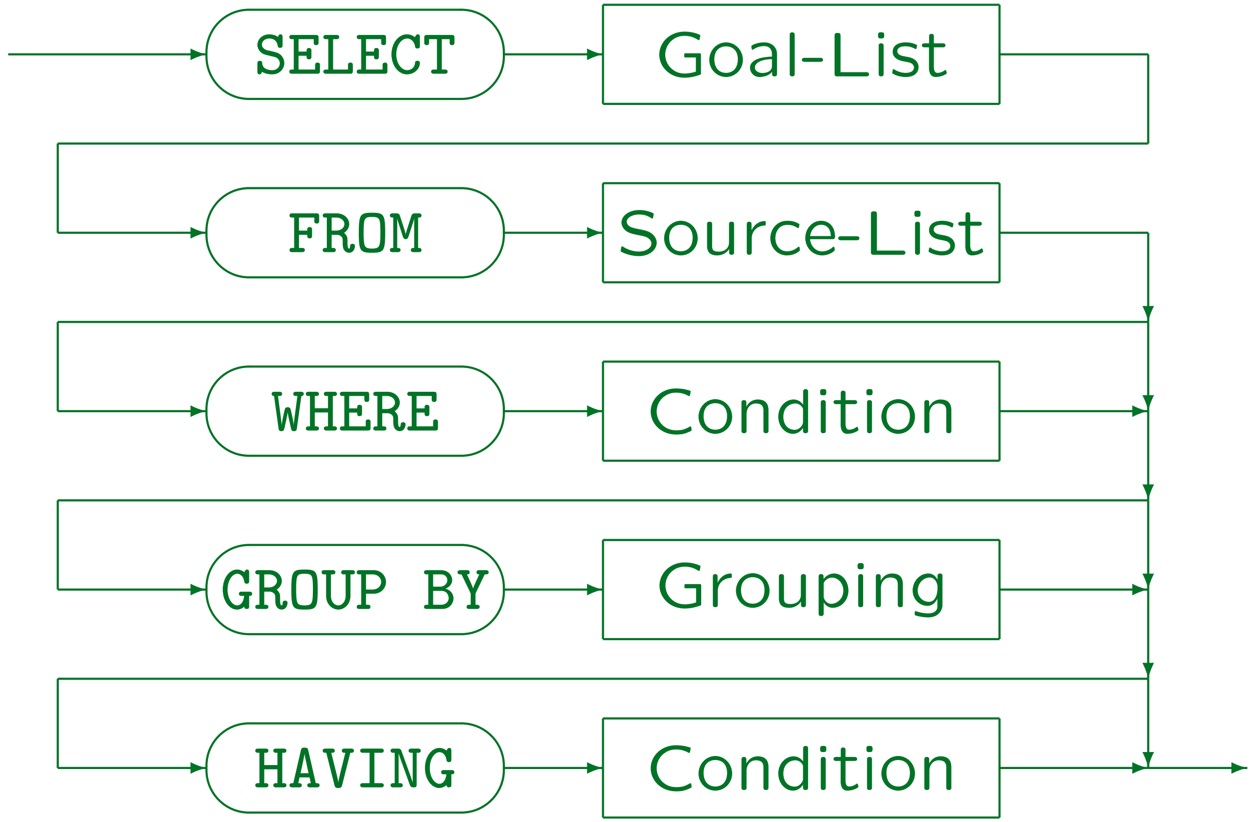
GROUP BY (9)

Warning:

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

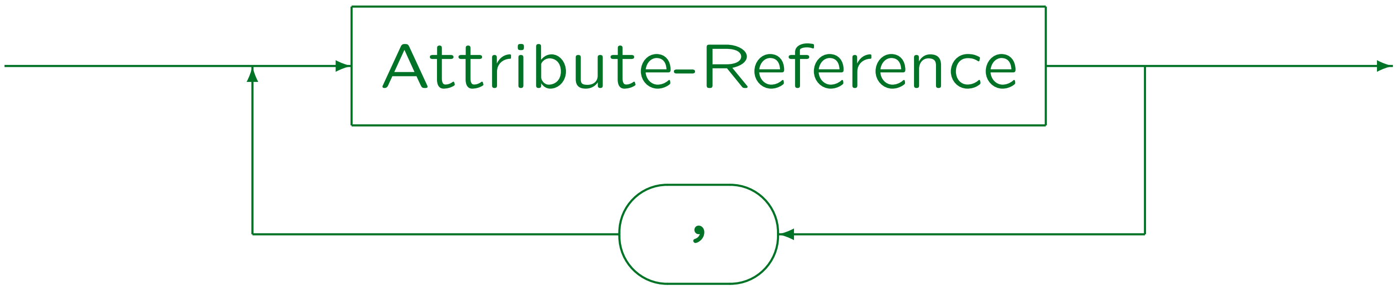
Syntax (1)

SELECT-Expression:



Syntax (2)

Grouping:



- E.g. `GROUP BY TITLE, C.CRN`

- Oracle, SQL Server, DB2, Access, and MySQL support the more general "Term" instead of "Attribute-Reference". Of course, no aggregation functions are permitted under `GROUP BY`.

HAVING (1)

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

HAVING (2)

- Which students got at least 18 homework points?

```
SELECT   FIRST, LAST
FROM     STUDENTS S, RESULTS R
WHERE    S.SID=R.SID AND R.CAT='H'
GROUP BY S.SID, FIRST, LAST
HAVING   SUM(POINTS) >= 18
```

FIRST	LAST
Ann	Smith
Michael	Jones

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

Evaluation

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

Syntax: Restrictions

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

  Inside aggregation functions, i.e. as their arguments, all attributes can be used. E.g. `AVG(A)/B`: The attribute `A` appars inside an aggregation function, `B` outside.
- **HAVING** without **GROUP BY** is legal, but uncommon: The query could only return 0 or 1 output rows.

WHERE vs. HAVING

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

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

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

```
SELECT   FIRST, LAST
FROM     STUDENTS S, RESULTS R
GROUP BY S.SID, R.SID, FIRST, LAST
HAVING   S.SID = R.SID AND SUM(POINTS) >= 18
```

Aggregation Subqueries (1)

- Who has the best result for Homework 1?

```
SELECT S.FIRST, S.LAST, R.POINTS
FROM   STUDENTS S, RESULTS R
WHERE  S.SID=R.SID AND R.CAT='H' AND R.ENO=1
AND    R.POINTS = (SELECT MAX(POINTS)
                   FROM   RESULTS
                   WHERE  CAT='H' AND ENO=1)
```

- For an aggregation query without GROUP BY, it is guaranteed that it will return exactly one row. Thus ANY/ALL is not necessary here.

Aggregation Subqueries (2)

- Since in SQL92, DB2, SQL Server, and Access a subquery returning a single data element can be used as a term, subqueries are also allowed in the SELECT-clause. Oracle 8.0 does not support this.
- This can replace GROUP BY. E.g. print for every student the sum of the homework points (null if none):

```
SELECT FIRST, LAST, (SELECT SUM(POINTS)
                     FROM   RESULTS R
                     WHERE  R.SID = S.SID
                     AND    R.CAT = 'H')
FROM   STUDENTS S
```

Nested Aggregations (1)

- Nested aggregations require a subquery under FROM.
- What is the average number of homework points? (counting only students who submitted homeworks)

```
SELECT AVG(X.HW_PT)
FROM   (SELECT   SID, SUM(POINTS) AS HW_PT
        FROM     RESULTS
        WHERE    CAT = 'H'
        GROUP BY SID) X
```

X	
SID	HW_PT
101	18
102	18
103	5

AVG(X.HW_PT)
13.67

Nested Aggregations (2)

- Oracle also supports nested aggregations written in this way:

```
SELECT   AVG(SUM(POINTS))
FROM     RESULTS
WHERE    CAT = 'H'
GROUP BY SID
```

Only Oracle!

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

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

Aggregating Different Sets (1)

- Subqueries under `FROM` make it possible to aggregate over different sets:

```
SELECT FIRST, LAST, H.PT AS HOMEWORK, M.PT AS MID
FROM   STUDENTS S,
       (SELECT   SID, SUM(POINTS) AS PT
        FROM     RESULTS
        WHERE    CAT = 'H'
        GROUP BY SID) H,
       (SELECT   SID, SUM(POINTS) AS PT
        FROM     RESULTS
        WHERE    CAT = 'M'
        GROUP BY SID) M
WHERE  S.SID = H.SID AND S.SID = M.SID
```

Aggregating Different Sets (2)

- This is also possible with conditional expressions, e.g. in Oracle:

```
SELECT FIRST, LAST,
       SUM(DECODE(R.CAT, 'H', R.POINTS, 0)) HW
       SUM(DECODE(R.CAT, 'M', R.POINTS, 0)) MID
FROM   STUDENTS S, RESULTS R
WHERE  S.SD = R.SID
```

- E.g. the conditional expression

```
DECODE(R.CAT, 'H', R.POINTS, 0)
```

returns `R.POINTS` if `R.CAT = 'H'` and 0 otherwise.

Maximizing Aggregations (1)

- Who has the best results in the homeworks (maximal sum of homework points)?

```
SELECT   FIRST, LAST, SUM(POINTS) AS TOTAL
FROM     STUDENTS S, RESULTS R
WHERE    S.SID = R.SID AND R.CAT = 'H'
GROUP BY S.SID, FIRST, LAST
HAVING   SUM(POINTS) >= ALL(SELECT   SUM(POINTS)
                            FROM     RESULTS
                            GROUP BY SID)
```

- An alternative solution with a view is shown on the next slide.

Maximizing Aggregations (2)

- Total number of HW points for every student:

```
CREATE VIEW HW_TOTALS AS
   SELECT   SID, SUM(POINTS) AS TOTAL
   FROM     RESULTS
   WHERE    CAT = 'H'
   GROUP BY SID
```

- Then one can use this as follows:

```
SELECT S.FIRST, S.LAST, H.TOTAL
FROM   STUDENTS S, HW_TOTALS H
WHERE  S.SID = H.SID
AND    H.TOTAL = (SELECT MAX(TOTAL)
                  FROM   HW_TOTALS)
```

Exercise: Possible Errors (1)

- What do you think about this query? Its task is to list all students who have solved at least two homeworks.

```
SELECT FIRST, LAST
FROM   STUDENTS S
WHERE  2 <= (SELECT COUNT(S.SID)
             FROM   RESULTS R
             WHERE  R.SID = S.SID
             AND    R.CAT = 'H')
```

Exercise: Possible Errors (2)

- And what about this query? Again, the task is to list students who have solved at least two homeworks.

```
SELECT FIRST, LAST
FROM   STUDENTS S, RESULTS R
WHERE  S.SID = R.SID
AND    R.CAT = 'H'
AND    COUNT(R.ENO) >= 2
```

Exercise: Possible Errors (3)

- And what about this query? Here the task is to list the number of homeworks per student.

```
SELECT   S.SID, S.FIRST, S.LAST, SUM(R.ENO)
FROM     STUDENTS S, RESULTS R
WHERE    S.SID = R.SID
AND      R.CAT = 'H'
GROUP BY S.SID, S.FIRST, S.LAST, R.ENO
```

Overview

1. Lexical Syntax
2. Tuple Variables, Joins
3. Terms, Conditions, Logic, Null Values
4. Subqueries, Nonmonotonic Constructs
5. Aggregations
6. Union, ORDER BY, Outer Join

UNION (1)

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

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

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

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

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

UNION (2)

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

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

- SQL distinguishes between
  - ◇ `UNION`: $\cup$ with duplicate elimination, and
  - ◇ `UNION ALL`: concatenation (retains duplicates).

  Duplicate elimination is quite expensive.

UNION (3)

- Print for every student his/her total number of homework points (0 if no homework submitted).

```
SELECT   S.FIRST, S.LAST, SUM(R.POINTS) AS TOTAL
FROM     STUDENTS S, RESULTS R
WHERE    S.SID = R.SID AND R.CAT = 'H'
GROUP BY S.SID, S.FIRST, S.LAST
UNION ALL
SELECT   S.FIRST, S.LAST, 0 AS TOTAL
FROM     STUDENTS S
WHERE    S.SID NOT IN (SELECT SID
                       FROM   RESULTS
                       WHERE  CAT = 'H')
```

UNION (4)

- Assign student grades based on Homework 1:

```
SELECT S.SID, S.FIRST, S.LAST, 'A' GRADE
FROM   STUDENTS S, RESULTS R
WHERE  S.SID=R.SID AND R.CAT='H' AND R.ENO=1
AND    R.POINTS >= 9
    UNION ALL
SELECT S.SID, S.FIRST, S.LAST, 'B' GRADE
FROM   STUDENTS S, RESULTS R
WHERE  S.SID=R.SID AND R.CAT='H' AND R.ENO=1
AND    R.POINTS >= 7 AND R.POINTS < 9
    UNION ALL
...
```

Other Set Operations in SQL

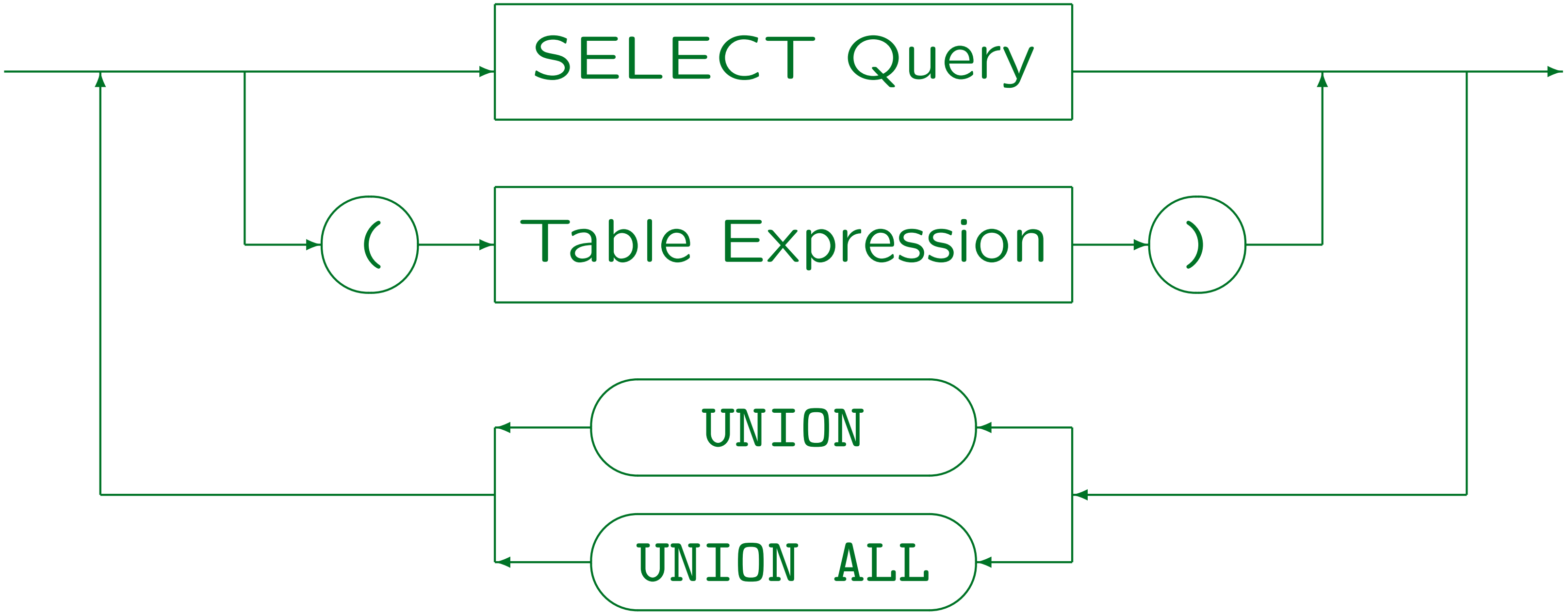
- SQL-86 contained only `UNION [ALL]`.
- The SQL-92 standard also contains `EXCEPT` (set difference, $-$) and `INTERSECT` ($\cap$).

  SQL-86, SQL Server and Access support only `UNION [ALL]`. MySQL does not support any of these operations. DB2 supports all SQL-92 set operators. In Oracle 8.0, the $-$ operator is called `MINUS` instead of `EXCEPT`. `ALL` for `MINUS` and `INTERSECT` is not supported in Oracle.
- These operations add nothing to the expressivity to the language.

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

UNION: Syntax

Table Expression:



- MySQL does not support union. SQL-86 contains `UNION` and `UNION ALL`.
- SQL-92 and DB2 support in addition `INTERSECT`, `INTERSECT ALL`, `EXCEPT`, and `EXCEPT ALL`. Oracle 8 supports `UNION`, `UNION ALL`, `INTERSECT` and `MINUS`.
- In Access, it is not possible to put parentheses around the entire query.

Union vs. Join

Exercise:

- Two alternatives for respresenting the homework, midterm, and final results of the students are:

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

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

- Write SQL queries to translate between the two.

Conditional Expressions (1)

- Whereas using UNION is the portable way to make a case analysis, sometimes a conditional expression suffices, and is more efficient.

  Conditional expressions look differently in each DBMS.

- E.g. Oracle has expressions of the form:

  DECODE(X, X_1, Y_1, X_2, Y_2, ..., Z)

- This is evaluated by comparing X first to X_1, then to X_2, and so on. If X_i is the first value with $X = X_i$, then Y_i is returned. If no X_i matches, Z is returned.

Conditional Expressions (2)

- E.g. print the exercise category in full for the results of Ann Smith (Oracle Version):

```
SELECT   DECODE(CAT, 'H', 'Homework',
                     'M', 'Midterm Exam',
                     'F', 'Final Exam',
                          'Unknown Category'),
         ENO, POINTS
FROM     STUDENTS S, RESULTS R
WHERE    S.SID = R.SID
AND      S.FIRST = 'Ann' AND S.LAST = 'Smith'
ORDER BY DECODE(CAT, 'H', 1, 'M', 2, 'F', 3, 4)
```

Conditional Expressions (3)

- In the SQL-92 standard (and e.g. DB2), this is written as follows:

```
SELECT CASE WHEN CAT='H' THEN 'Homework',
            WHEN CAT='M' THEN 'Midterm Exam'
            WHEN CAT='F' THEN Final Exam'
            ELSE 'Unknown Category' END
       ENO, POINTS
FROM   STUDENTS S, RESULTS R
WHERE  S.SID = R.SID
AND    S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

- Oracle 8i (not 8.0) supports a similar syntax, but requires a comma between the `WHEN` clauses.

Conditional Expressions (4)

- The SQL-92 standard (and DB2, but not Oracle 8i) supports also the following abbreviation which is very similar to Oracle's `DECODE`:

```
SELECT CASE CAT WHEN 'H' THEN 'Homework',
                WHEN 'M' THEN 'Midterm Exam',
                WHEN 'F' THEN Final Exam',
                ELSE 'Unknown Category' END,
       ENO, POINTS
FROM   STUDENTS S, RESULTS R
WHERE  S.SID = R.SID
AND    S.FIRST = 'Ann' AND S.LAST = 'Smith'
```

Conditional Expressions (5)

- A typical application of condtional expressions is to replace a null value by something else.
- In Oracle `NVL(`X`, `Y`)` is equivalent to

  `DECODE(`X`, NULL, `Y`, `X`)`

  I.e. if X is not null, then X is the result.
  If X is null, then Y is the result.
- `COALESCE(`X`, `Y`)` is the same in standard SQL-92.
  There it abbreviates

  `CASE WHEN `X` IS NOT NULL THEN `X` ELSE `Y` END`

Conditional Expressions (6)

- E.g. list the email address of all students, and write "(none)" if the column is null:

```
SELECT FIRST, LAST, NVL(EMAIL, '(none)')
FROM   STUDENTS
```

- Finally note that conditional expressions are normal terms, so they can be input for other datatype functions or e.g. aggregation functions.

Sorting Output (1)

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

  It is much easier to search a specific value in a sorted table. Without "`ORDER BY`" the sequence of output rows means nothing (it depends on the algorithms used in the DBMS and may change between versions).

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

  Whereas sorting is the only formatting command that found its way into the SQL standard, DBMS tools usually offer more options. E.g. to have a pagebreak when the value in a specific column changes, to show negative values in red ink, etc. However, sorting may also be important when an application program retrieves the data.

Sorting Output (2)

- One can specify a prioritized list of sorting criteria.

  The "ORDER BY" list can contain multiple columns. The second column is only used for ordering two tuples which have the same value in the first column, and so on.

- E.g.: Print the homework results sorted by exercise, and for each exercise by points (best result first), and if there is still a tie, alphabetically by name:

```
SELECT   R.ENO, R.POINTS, S.FIRST, S.LAST
FROM     STUDENTS S, RESULTS R
WHERE    S.SID = R.SID AND R.CAT = 'H'
ORDER BY R.ENO, R.POINTS DESC, S.LAST, S.FIRST
```

Sorting Output (3)

- Result of the example query on the previous page:

ENO	POINTS	FIRST	LAST
1	10	Ann	Smith
1	9	Michael	Jones
1	5	Richard	Turner
2	9	Micael	Jones
2	8	Ann	Smith

- E.g. the first two tuples have the same value in the highest priority sort criterion (ENO), and the second criterion (POINTS DESC) determines their sequence.

  It does not matter that according to the criterion of third priority (LAST) the sequence would be the other way round.

Sorting Output (4)

- According to the SQL-92 standard, one can only sort by columns that appear in the output.

  E.g. it is impossible to print a list of student names ordered by total points without printing these points. But tools like SQL*Plus can suppress output columns from the query result.

- However, in all five systems (Oracle 8, DB2, SQL Server, Access, MySQL) one can sort by any term that would be allowed in the `SELECT`-clause.

  In these systems, it is not necessary that the term really appears in the `SELECT`-clause. E.g. one can sort by `UPPER(LAST)`, but print `LAST`. With `DISTINCT`, one can only sort by result columns (in Oracle one can still use them in terms and MySQL has no restriction).

Sorting Output (5)

- Sometimes it is necessary to add columns to database tables to get a sort value, e.g.
  - ◇ The results should be printed in the sequence: Homeworks, Midterm, Final (not alphabetically).
  - ◇ The “University of Pittsburgh” should appear in a list of universities under “P”, not under “U”.
- If the student names were stored as a single string in the form “FIRST LAST”, it would be (more or less) impossible to sort by last name.

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

Sorting Output (6)

- "`DESC`" means descending (inverse order from high to low values), the default is "`ASC`" (ascending).
- It is also possible to refer to columns by number, e.g.: `ORDER BY 2, 4 DESC, 1`

  Column numbers refer to the sequence in the `SELECT`-list. They were important in earlier SQL versions, where one could not explicitly name the result columns. Today, one probably should use column names.
- Null values are all listed first or all listed last in the sort sequence (depending on the DBMS).

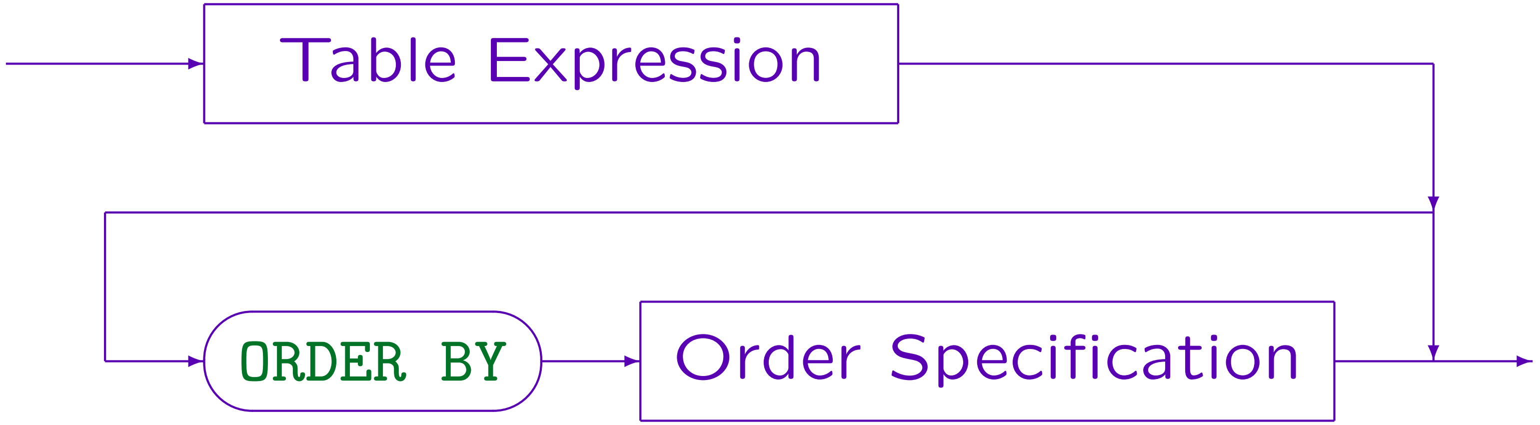
  In Oracle, one can specify `NULLS FIRST` or `NULLS LAST`.

Sorting Output (7)

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

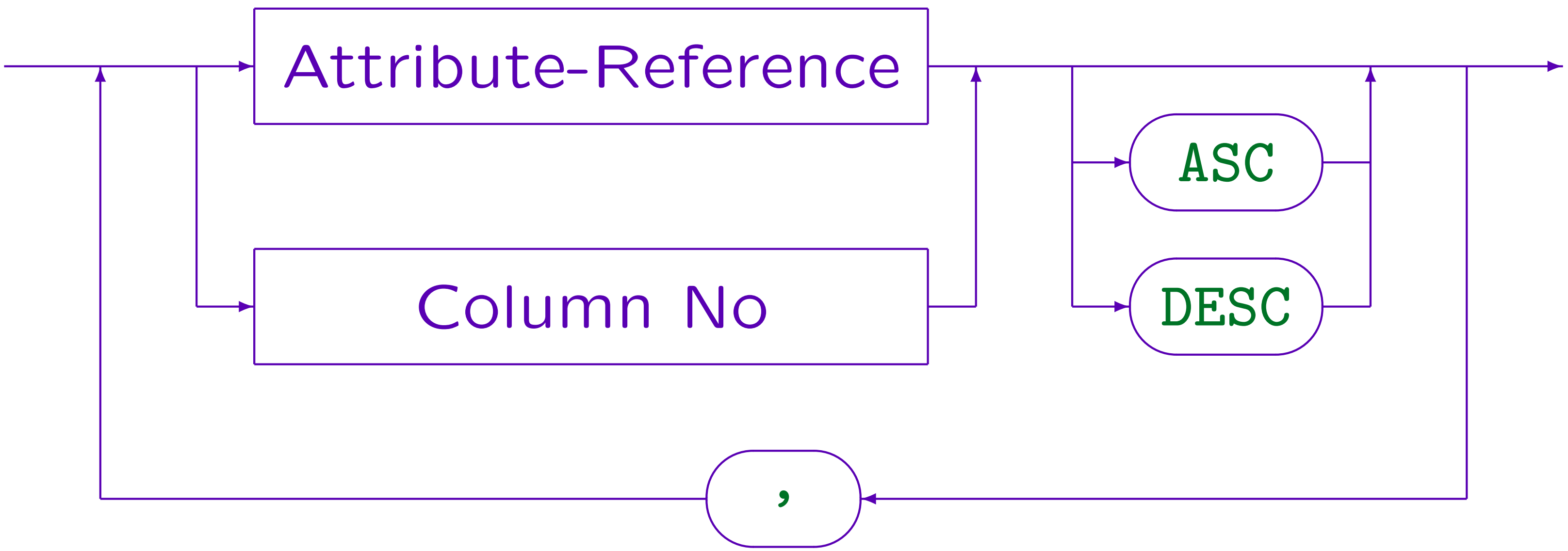
Sorting Output (8)

SQL Query:



Sorting Output (9)

Order Specification:



- Most DBMS permit "Term" instead of "Attribute Reference" (except if DISTINCT or UNION etc. are specified). Then basically the same restrictions apply as for terms in the SELECT-list (there might be additional restrictions for the use of aggregation functions).

Joins in SQL-92 (1)

- An important and useful operation of relational algebra is the join (in several variants).
- In SQL-86, one cannot directly specify a join. One writes a cartesian product (FROM) and then does a selection (WHERE). This is still the usual case.
- E.g. the natural join of RESULTS and EXERCISES is:

```
SELECT R.CAT AS CAT, R.ENO AS ENO, SID,
       POINTS, TOPIC, MAXPT
FROM   RESULTS R, EXERCISES E
WHERE  R.CAT = E.CAT AND R.ENO = E.ENO
```

Joins in SQL-92 (2)

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

```
SELECT SID, ENO, (POINTS/MAXPT)*100
FROM   RESULTS R NATURAL JOIN EXERCISES E
WHERE  CAT = 'H'
```

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

```
R.CAT = E.CAT AND R.ENO = E.ENO
```

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

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

Joins in SQL-92 (3)

- Current systems support the standard only partially:
  - ◇ SQL-92 joins are not supported in Oracle 8.0.
  - ◇ Some types of joins are supported in DB2, SQL Server, and Access, but the above "natural join" is not. A join with explicit condition is possible:

```
SELECT SID, R.ENO, (POINTS/MAXPT)*100
FROM   RESULTS R INNER JOIN EXERCISES E
       ON R.CAT = E.CAT AND R.ENO = E.ENO
WHERE  R.CAT = 'H'
```

  - ◇ MySQL has a natural join, but it behaves not exactly as required in the standard (see below).

Joins in SQL-92 (4)

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

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

- The reason why joins where added to SQL is probably the "outer join": For the outer join, the equivalent formulation in SQL-86 is significantly longer.

Outer Join: Repetition

- The usual join eliminates tuples without partner:

A	B
a_1	b_1
a_2	b_2

$\bowtie$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_2	b_2	c_2

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

A	B
a_1	b_1
a_2	b_2

$⟕$

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_1	b_1	
a_2	b_2	c_2

Rows from the left table are filled with "null" if necessary.
There are also a right outer join and a full outer join.

Outer Join in SQL-92 (1)

- E.g. number of submissions per homework. If there is no submission, the number 0 should be printed:

```
SELECT   E.NO, COUNT(SID)
FROM     EXERCISES E LEFT OUTER JOIN RESULTS R
         ON E.CAT = R.CAT AND E.SNO = R.ENO
WHERE    E.CAT = 'H'
GROUP BY E.ENO
```

- All exercises are present in the result of the left outer join. In exercises without solutions, the attributes of SID and POINTS are filled with null values.
- COUNT(SID) does not count rows where SID is null.

Outer Join in SQL-92 (2)

- Equivalent query without outer join (12 vs. 5 lines):

```
SELECT    E.NO, COUNT(*)
FROM      EXERCISES E, RESULTS R
WHERE     E.CAT = 'H' AND R.CAT = 'H'
AND       E.ENO = R.ENO
GROUP BY E.ENO
UNION ALL
SELECT    E,ENO, 0
FROM      EXERCISES E
WHERE     E.CAT = 'H'
AND       E.ENO NOT IN (SELECT R.RNO
                        FROM   RESULTS R
                        WHERE  R.CAT = 'H')
```

Outer Join in SQL-92 (3)

- E.g. print for every student the number of homeworks he/she has solved (including 0).
- The following query does not work:
  Students without homework are not listed.

```
SELECT   FIRST, LAST, COUNT(ENO)     Wrong!
FROM     STUDENTS S LEFT OUTER JOIN RESULTS R
         ON S.SID = R.SID
WHERE    R.CAT = 'H'
GROUP BY S.SID, FIRST, LAST
```

- The outer join is constructed before the WHERE-condition is evaluated.

Outer Join in SQL-92 (4)

- In general, one must be careful not to eliminate possible join partners after the outer join is done.
- One has to select the homework results before the outer join is done:

```
SELECT   FIRST, LAST, COUNT(R.ENO)
FROM     STUDENTS S LEFT OUTER JOIN
            (SELECT SID, ENO
             FROM   RESULTS
             WHERE  CAT = 'H') R
         ON S.SID = R.SID
GROUP BY S.SID, FIRST, LAST
```

Outer Join in SQL-92 (5)

- One can also put the condition on the right table into the join condition:

```
SELECT   FIRST, LAST, COUNT(R.ENO)
FROM     STUDENTS S LEFT OUTER JOIN RESULTS R
         ON S.SID = R.SID AND R.CAT = 'H'
GROUP BY S.SID, FIRST, LAST
```

- The SQL-92 permits any `WHERE`-condition that refers only to the tuple variables on the left and right side of the join. (But don't abuse this.)

  It seems that DB2 and Access permit no subqueries in the `ON`-clause. More complex conditions must be enclosed in parentheses in Access.

Outer Join in SQL-92 (6)

- Conditions on the left table make little sense in the condition of the left outer join.
- E.g. consider this query:

```
SELECT E.CAT, E.ENO, R.SID, R.POINTS
FROM   EXERCISES E LEFT OUTER JOIN RESULTS R
       ON E.CAT = 'H' AND R.CAT = 'H'
          AND E.ENO = R.ENO
```

- Exercise: Will `E.CAT = 'M'` appear in the output?

  ☐ yes ☐ no

Outer Join in SQL-92 (7)

- MySQL has no subqueries, but sometimes one can use the outer join instead.
- E.g. students who did not submit any homework:

```
SELECT S.SID. S.FIRST, S.LAST
FROM   STUDENTS S LEFT OUTER JOIN RESULTS R
       ON S.SID = R.SID AND R.CAT = 'H'
WHERE  R.CAT IS NULL
```

- Of course, instead of R.CAT one can test any attribute of RESULTS for the null value.

  The test for the null value checks whether the current STUDENTS tuple did not find a join partner.

Join Syntax in SQL-92 (1)

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

Join Syntax in SQL-92 (2)

- The join condition can be specified as follows:
  - ◇ The keyword `NATURAL` in front of the join name.
  - ◇ "`ON` ⟨`Condition`⟩" follows the join.
  - ◇ "`USING` ($A_1, \ldots, A_n$)" follows the join.

    `USING` lists join attributes (e.g. for specifying the natural join). Attributes with the names $A_1, \ldots, A_n$ must appear in both tables and the join condition is $R.A_1 = S.A_1 \wedge \cdots \wedge R.A_n = S.A_n$. `NATURAL` is equivalent to specifying `USING` with all common attribute names.

- Only one of these constructs can be used.
- `CROSS JOIN` and `UNION JOIN` have no join condition.

Join Syntax in SQL-92 (3)

- According to the standard, the NATURAL join and the join with USING produce a table with only one copy of the common attributes.
- Furthermore, the common attributes are listed first and cannot be referenced with a tuple variable.

```
SELECT *
FROM   RESULTS R NATURAL JOIN EXERCISES E
```

- The result columns are CAT, ENO, R.SID, R.POINTS, E.TOPIC, E.MAXPT (in this sequence).

  It is illegal to refer to R.CAT or E.CAT, only CAT can be used (and the same for ENO).

Join Syntax in SQL-92 (4)

- Oracle 8i does not support any SQL-92 joins.
- Inner and outer join with `ON` work in SQL Server, DB2, Access, and MySQL.

  In Access and MySQL, the keyword `INNER` is not optional.
- `USING` is supported in none of the five systems.
- `NATURAL` is supported only in MySQL, but MySQL does not merge the common columns.

  Probably these naming rules which are a bit strange for SQL (but perfectly normal for relational algebra) are the reason why `USING` and `NATURAL` are not implemented in the big commercial systems.

Join Syntax in SQL-92 (5)

- CROSS JOIN is supported only in SQL Server and MySQL, not Access and DB2.

  One can write a comma for CROSS JOIN, so it is not very useful.

- UNION JOIN is supported in none of the five systems.

  However, in SQL-92 (and e.g. Oracle, DB2, SQL Server, not Access), one can write a subquery containing UNION or UNION ALL also in the FROM-clause. So with a bit more keystrokes, one can simulate the union join. By the way, it is a bit strange that e.g. "FROM A NATURAL JOIN B" is legal in SQL-92, but "FROM A UNION B" is not. Also, SQL-92 permits to write "FROM (SELECT * FROM A UNION SELECT * FROM B) X", but the same with "NATURAL JOIN" instead of "UNION" is a syntax error [Date/Darwen, 1997, p. 148].

Join Syntax in SQL-92 (6)

- In the FROM clause, one can also combine joins and the declaration of further tuple variables (separated by "," as usual).
- One can also join the result of joining two tables with a third one (and so on). The syntax is:

```
SELECT ...
FROM   R LEFT JOIN S ON R.A=S.B
         LEFT JOIN T ON S.C=T.D
```

- It is also possible to use parentheses, but then one has to declare a new tuple variable after the (...).

Outer Join in Oracle (1)

- In Oracle, the outer join is specified under `WHERE`.
- Instead of the usual join condition $R.A = S.B$ one writes
  - $R.A = S.B$(+) for the left outer join of R and S,
  - $R.A$(+) $= S.B$ for the right outer join of R and S.

  I.e. the special marker "(+)" is appended to attributes of the table which can be replaced with nulls.

  I.e. this protects the tuples of the other table (not marked with "(+)"). There are many syntactic restrictions which ensure that this is really an outer join. If the join is done on several attributes, all must be marked. It is possible to write also $S.B$(+) $= c$ with a constant c or e.g. $R.A = S.B$(+) $+ 1$.

Outer Join in Oracle (2)

- E.g. number of submissions per exercise (can be 0):

```
SELECT   E.CAT, E.ENO, COUNT(SID)
FROM     EXERCISES E, RESULTS R
WHERE    E.CAT = R.CAT(+) AND E.ENO = R.ENO(+)
GROUP BY E.CAT, E.ENO
```

- As in the SQL-92 outer join, the outer join is constructed before any other conditions in the WHERE-clause are applied.

  No matter in what sequence the conditions are written. But as shown above, one can use a subquery under FROM to do a selection before the outer join.