Part II

Database Query Languages

What’s a “Query” and what isn’t?

The term “query” can be used in (at least) two different meanings

1. (narrow interpretation) the formulation of a (database) **retrieval** request, *i.e.*, a read-only operation on the database,
2. (general interpretation) the formulation of an **arbitrary** database request, *e.g.*, manipulation (update) or data definition operations included.

Throughout the initial chapters of this part, we concentrate on the *narrow* meaning, focussing on retrieval operations. Later, though, when talking SQL, we will also look at the other functionalities of a DBMS API.

9. Declarative Database Languages: Principles

This section's goal

After completing this chapter, you should be able to

- explain the **basic ideas** of (relational) **database languages**:
  - **declarative** vs. navigational access,
  - different language styles (algebra, calculus, SQL, logic, graphical),
- formulate simple queries in different RDBMS languages,
- explain the "use cases" for the different language styles.

Declarative vs. navigational languages

Recall from the introduction to relational databases

- we are interested in *high-level, declarative* query languages, such as SQL, that allow the *specification* of our information needs,
- **not** in (procedural, navigational) programming languages that allow us to code algorithms that retrieve the necessary data!

What’s the salary of employee Jones?

Given a relational representation of Employee information in a table

EMP

EMPNO	ENAME	SAL	...
7369	SMITH	7902	...
7499	ALLEN	7698	...
7521	WARD	7698	...
7566	JONES	7839	...
7654	MARTIN	7698	...
...	...	...	...

Recall: Navigational vs. declarative access

Cont'd: navigation

... you can certainly come up with the following lookup algorithm:

```
$result \leftarrow \emptyset$;
open_file($EMP$);
while $\neg$EOF($EMP$) do
    $e \leftarrow$ get_next_record($EMP$);
    if $e.ENAME$ = "Jones" then
        $result \leftarrow result \cup e$;
    fi
od
return($result$);
```

Cont'd: specification

... but it is certainly much easier to formulate a SQL query:

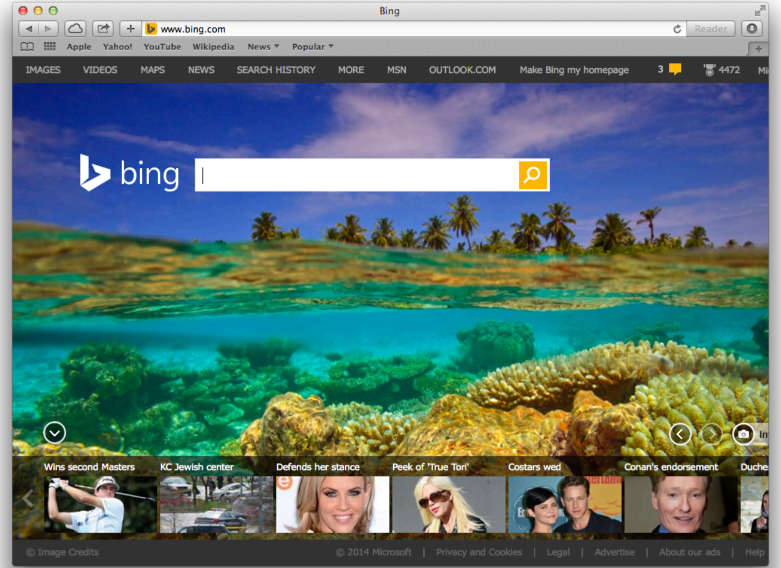
```
SELECT  SAL
FROM    EMP
WHERE   ENAME = 'Jones'
```

And the best thing is: it is very likely to be

- not only *much more convenient* to express,
- but also **much more efficient** to execute!

even better yet: the more complex the query, the bigger the savings!

Do we *really* need more than this...? Why?


Bing
www.bing.com
Reader
Apple Yahoo! YouTube Wikipedia News Popular
IMAGES VIDEOS MAPS NEWS SEARCH HISTORY MORE MSN OUTLOOK.COM Make Bing my homepage 3 4472
bing
Wins second Masters
KC Jewish center
Defends her stance
Peek of 'True Tori'
Costars wed
Conan's endorsement
© Image Credits
© 2014 Microsoft | Privacy and Cookies | Legal | Advertise | About our ads | Help

High-level mathematical formalism

... what mathematical notions are appropriate?

When introducing the relational data model, we noticed the similarity to mathematical set theory, mathematical relations, and first order predicate logic.

1. **Set operators** look like a natural choice: Given two sets, $\mathcal{A}$ and $\mathcal{B}$, we are used to compute their union, difference, intersection and Cartesian product, writing
   $\mathcal{A} \cup \mathcal{B} \quad \mathcal{A} - \mathcal{B} \quad \mathcal{A} \cap \mathcal{B} \quad \mathcal{A} \times \mathcal{B}$.
   All of these work for relations (sets of tuples) as well!

2. **Predicate logic** is another best friend: We're used to define sets via predicates,[33] *e.g.*,
   $\mathcal{E} = \{n \in \mathbb{N} \mid n \mod 2 = 0\}$ or
   $\mathcal{O} = \{n \in \mathbb{N} \mid \exists m \in \mathbb{N} : n = 2m + 1\}$.
   Again, this should be useful with relations!

[33] in Computer Science, such set definitions are called "set comprehensions"

Option 1: Operator-based languages

Relational Algebra is such an operator-based language. Each operator

- takes one or two relations as input argument(s), and
- delivers a relation as output result.

(Compositionality) As an immediate consequence, compound *expressions* may be formed by *nesting*: apply another operator to the result of a previous operator, as in

$$\mathcal{A} \cup (\mathcal{B} - \mathcal{C}).$$

Relational algebra introduces two new operators to **select** tuples that qualify w.r.t. a given condition and to **project** tuples onto a subset of attributes

- $\sigma_P(R)$ returns the subset of R-tuples that match predicate P
- $\pi_{A,B,C}(R)$ retains only attributes A, B, and C of relation R

Σ Five independent basic operators

$\cup$, $-$, $\times$, π, and σ are independent, all other algebraic operators can be derived from these.

Example using Relational Algebra

Example

Get salary of employee Jones.

$$\pi_{\text{SAL}}\left(\sigma_{\text{ENAME} = \text{'Jones'}}(\text{EMP})\right)$$

Example

Employee names and salaries with department names, for those making between 3,000 and 5,000 (see the complex QbE query above).

$$\pi_{\text{ENAME,DNAME,SAL}}\left(\sigma_{3000 < \text{SAL} < 5000}(\text{EMP}) \bowtie \text{DEPT}\right)$$

The operator $\bowtie$ denotes the natural join.

Natural Join in Relational Algebra

In RA, join is a derived operator, defined by other operators

Definition: Natural Join

Let the schema of R be $\text{sch}(R) = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$ and $\text{sch}(S) = \{B_1, \ldots, B_m, C_1, \ldots, C_k\}$, *i.e.*, the attributes B_i are common to both relations, *e.g.*, as a key–foreign key pair.
The **natural join** of R and S, $R \bowtie S$, can be defined as

$$R \bowtie S \equiv \pi_{RS}\left(\sigma_F(R \times S)\right)$$

with $F = (R.B_1 = S.B_1 \land \cdots \land R.B_m = S.B_m)$ and $RS = \{A_1, \ldots, A_n, B_1, \ldots, B_m, C_1, \ldots, C_k\} = \text{sch}(R) \cup \text{sch}(S)$.

Option 2: Predicate logic

Relational Calculus is a query language that defines sets, namely, query results, via predicates.

Simple queries in Relational Calculus

1. All employees named Jones: Q1 $=$
2. The salaries of Jones: Q2 $=$
3. Employee names and salaries, with department names, for those making between 3,000 and 5,000:
   Q3 $=$

N.B. We are using a little notational freedom here, textbook Relational Calculus looks somewhat more involved.

Relational Calculus: How it works

Notice the use of *variables* for tuples (e, d in the examples), hence the name "Tuple Relational Calculus (TRC)".[34] Notation:

- $e \in \text{EMP}$ introduces a (bound) variable, e, that ranges over all tuples in the given relation, EMP;
- $e.\text{SAL}$ (dot notation) is used to access attribute values of tuples;
  - a wide variety of notations can be found for attribute access, *e.g.*, also $e(\text{SAL})$, $e[\text{SAL}]$, $\text{SAL}(e)$, ...
- Quantifiers $\exists e \in \text{EMP}$ and $\forall e \in \text{EMP}$ are allowed in the predicates;
  - the typical textbook would not use bounded quantifiers, though. Instead of $\exists e \in \text{EMP} : \ldots$ one would use $\exists e : \text{EMP}(e) \wedge \ldots$
- Result tuples are constructed component-wise. In the classical textbook TRC, instead of $\{(e.\text{ENAME}, d.\text{DNAME}) \mid \ldots\}$ you would have to use an additional variable, t, and write $\{t \mid \cdots \wedge t.\text{ENAME} = e.\text{ENAME} \wedge t.\text{DNAME} = d.\text{DNAME}\}$.

[34]there is also a Domain RC variant, where variables denote attribute values

Predicate logic as a query language

There is an alternative style of logic-based query languages. Here we also use predicates, but not to define relations via set comprehensions as in "$Q = \{t \mid P(t)\}$", rather, relations themselves are considered as (stored extensions of) predicates.

Relations as stored extensions of predicates

EMP			
EMPNO	ENAME	SAL	...
...	...	...	...
7566	JONES	7839	...
7654	MARTIN	7698	...
...	...	...	...

A relation/table, such as then one above records the fact that predicate EMP holds for certain parameter values

$$\text{EMP}(7566,\text{JONES},7839,\ldots) \equiv \textbf{true}$$
$$\text{EMP}(7654,\text{MARTIN},7698,\ldots) \equiv \textbf{true}$$

and so on.

Rule-based querying

We can now query an RDB by defining new (result) relations via **derivation (or inference) rules** using the stored relations as input.[35]

Get all employees named Jones

$Q1(enum, ename, sal, \ldots) \Longleftarrow \mathrm{EMP}(enum, ename, sal, \ldots).$

Get Jones' salaries

$Q2(sal) \Longleftarrow \mathrm{EMP}(enum, ename, sal, \ldots) \land ename = \text{"Jones"}$, or even
$Q2'(sal) \Longleftarrow \mathrm{EMP}(enum, \text{"Jones"}, sal, \ldots).$

Employee names and salaries, with department names, for those making between 3,000 and 5,000

$Q3(en, dn, s) \Longleftarrow \mathrm{EMP}(eno, en, sal, dno, \ldots) \land \mathrm{DEPT}(dno, dn, \ldots) \land 3000 < sal < 5000.$

[35] Notice the use of variables for parameter/attribute values.

A comparison of languages

Let us compare (some of) the other languages sketched above. The design of SQL has in fact been influenced, at least, by RA and TRC.

We will compare basic queries in those three languages, using simple examples. A detailed discussion is deferred until after the presentation of all these languages. The **five basic operators of relational algebra**, namely

- union ($\cup$),
- difference ($-$),
- product ($\times$),
- selection (σ) , and
- projection (π)

will be our guideline, and, of course, we will also look at the ubiquitous join.

Selection: Predicate-oriented search

1. Selection: Picking some tuples based on a search predicate

Let P be some predicate that can be evaluated on a sinlge tuple $r \in R$ of a relation R[36] (*i.e.*, it references attributes, uses constants and comparison operators, combines parts with logical and, or, not operators). The subset of R satisfying P can be retrieved, by

in RA: $\sigma_P(R)$

in TRC: $\{r \in R \mid P(r)\}$

in SQL `SELECT DISTINCT * FROM R WHERE P`

N.B. As we will see later, there are some subtle differences as to what kind of predicates are permitted (*e.g.*, quantifiers are *only* allowed in TRC), but apart from that, the differences are of a lexical nature.

[36] "Relation R" here and in the sequel might as well be the result of some subexpression!

Projection: Hiding unnecessary details

2. Projection: Keep only a specified set of attributes

Let $L \subseteq \text{sch}(R)$ be a subset of relation R's attributes. If we're not interested in the values of other attributes, we can restrict tuples to the components in L

in RA: $\pi_L\,(R)$

in TRC: $\{r(L) \mid r \in R\}$

in SQL `SELECT DISTINCT L FROM R`

N.B. Duplicate removal in SQL is essential here, at least if no candidate key of R is preserved in the projection!

Relational product: Combining data of different relations

3. Product: Generate all combinations of *R*- and *S*-tuples

Let *R* and *S* be relations without common attributes:[37] $\mathcal{R} = \text{sch}(R)$, $\mathcal{S} = \text{sch}(S)$, $\mathcal{R} \cap \mathcal{S} = \emptyset$. Taking the product of *R* and *S* generates a relation *RS* with the attributes $\mathcal{R} \cup \mathcal{S}$. Each tuple of *R* is combined with each tuple of *S*

in RA: $R \times S$

in TRC: $\{rs \mid \exists r \in R : \exists s \in S : rs(\mathcal{R}) = r \wedge rs(\mathcal{S}) = s\}$

in SQL `SELECT DISTINCT * FROM R, S`

N.B. In SQL, an equivalent, more explicit formulation could be “`SELECT DISTINCT R.*, S.* FROM R, S`”.

[37] If *R* and *S* share common attributes, we will have to rename one “copy” of each shared attribute, such that the resulting schema has unique names (see later).

Union, Difference: Easy & obvious—relations are sets

4./5. Union & Difference: The usual set operations

Let R and S be two relations over the same schema (*i.e.*, they have all attributes in common). Union collects all tuples in R, S, or R and S. Differences retains those tuples from the first argument that are not in the second argument relation

in RA: $R \cup S$ or $R - S$

in TRC: $\{t \mid t \in R \vee t \in S\}$ or $\{r \in R \mid r \notin S\}$

in SQL
```
SELECT DISTINCT * FROM R
UNION or EXCEPT
SELECT DISTINCT * FROM S
```

N.B. DISTINCT could be omitted here, since duplicate elimination is automatically performed before set operations in SQL.

Natural Join: Key-foreign key relationships

We have already seen that Natural Join is not a basic (algebra) operator, since it can be defined using other operators. In fact, the five algebra operators seen above are sufficient to define all the others. Nontheless, we have a look at how Joins are expressed in the three languages here

Join: Combine data that belongs together

Let R and S be relations with attributes sch$(R) = \mathcal{R} \cup \mathcal{X}$, sch$(S) = \mathcal{S} \cup \mathcal{X}$, *i.e.*, R and S share the attributes $\mathcal{X}$. Often, $\mathcal{X}$ will be key in one and foreign key in the other relation, but not necessarily so. The Natural Join brings together matching R and S tuples, *i.e.*, those with same $\mathcal{X}$-values

in RA: $R \bowtie S$

in TRC: $\{rs \mid \exists r \in R : \exists s \in S : rs(\mathcal{R} \cup \mathcal{X}) = r \wedge rs(\mathcal{S} \cup \mathcal{X}) = s\}$

in SQL `SELECT DISTINCT * FROM R NATURAL JOIN S`

Final remark on SQL vs. RA and TRC

From this initial comparison, it may look as if SQL mainly drew from Relational Algebra. This is not the case! Just to give one example...

- SQL allows for so-called “Alias names”, to be introduced in the FROM clause. They can be used as abbreviations for relation names and/or to disambiguate queries that reference the same relation more than once:

```
SELECT  DISTINCT r.*, s.*
  FROM  LongRelationName1 AS r,
        AnotherVeryLongRelationName AS s
 WHERE  r.ATTR1 = s.ATTR2 …
```

- These alias names could also be called “tuple variables”: a concept that is clearly absent from (any) algebra, but exactly the same as in TRC.

10. The Relational Database Query Language SQL

After completing this part, you should be able to

- write advanced SQL queries including, *e.g.*, multiple **tuple variables** over different/the same relation,
- use **aggregation**, **grouping**, UNION, and several types of nested subqueries,
- be comfortable with the various **join variants,**
- evaluate the **correctness** and **equivalence** of SQL queries,
- judge the **portability** of certain SQL constructs;
- disinguish online *transaction* processing (OLTP) from online *analytical* processing (OLAP) requirements,
- formulate basic OLAP queries using appropriate SQL extensions;
- formulate *recursive* queries using the corresponding SQL extension.

10.1 Basic SQL query syntax

SQL's SFW-block (syntax to be extended)

$$
\begin{array}{rl}
\texttt{SELECT} & [\texttt{DISTINCT}]\ [r_i.]\{*\mid\langle\text{Attribute}\rangle_j[\text{AS}\langle\text{NewName}\rangle_j],\ldots\}^* \\
\texttt{FROM} & \{\langle\text{RelationName}\rangle_1[[\text{AS}]r_i],\ldots\}^* \\
[\texttt{WHERE} & \langle\text{Condition}\rangle]
\end{array}
$$

- $[[\text{AS}]r_i]$: SQL's "alias names" are tuple variables (*cf.* TRC),

  … if you omit them, then $\langle\text{RelationName}_i\rangle$ serves as a tuple variable.
- $[\text{AS}\langle\text{NewName}\rangle_j]$: SQL allows for *renaming* in the projection list.
- (Coarse) semantics:
  1. take the product of the tables in the FROM clause;
  2. apply selection with WHERE-predicate $\langle\text{Condition}\rangle$, if present;
  3. finally project according to SELECT clause, including duplicate elimination, if DISTINCT is present, without otherwise;

     : take all attributes, r_i.: all attributes of r_i.

Attribute references (1)

- In general, **attributes are referenced** in the form

$$R.A$$

- If an attribute may be associated to a tuple variable in an unambiguous manner, the variable may be omitted.

Example

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

— Here, FIRST, LAST can only refer to S.
— CAT, ENO, POINTS can only refer to R.
— SID on its own would be ambiguous (may refer to S or R).

- If an explicit tuple variable is declared, then the implicit tuple variable ⟨RelationName⟩ is **not declared**, *e.g.*, STUDENTS.SID in the above WHERE clause would yield an error.

Attribute references (2)

Consider this query.

Erroneous SQL!

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

- Although forced to be equal by the join condition, SQL requires the user to specify unambiguously which of the ENO attributes (bound to R or E) is meant in the SELECT clause.
- The ambiguity rule is **purely syntactic** and does not depend on the query semantics.

Expressions in the SELECT & WHERE clause

SQL allows for expressions in the SELECT clause…

For each RESULT, compute percentage of POINTS achieved

```
SELECT  SID, R.CAT, R.ENO, (100*POINTS/MAXPT) as PCT
FROM    RESULTS R NATURAL JOIN EXERCISES E
```

… and in the WHERE clause as well.

Select RESULTs with more than 80% POINTS achieved

```
SELECT  R.*
FROM    RESULTS R NATURAL JOIN EXERCISES E
WHERE   POINTS > 0.8*MAXPT
```

N.B. The kind of expressions depends on the collection of basic data types supported.

10.2 Joins: Traditional vs. Modern Syntax

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.Alg.	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

Expressing joins in traditional SQL (SQL:1989)

Consider a query with two tuple variables:

```
SELECT  $A_1, \ldots, A_n$
FROM    STUDENTS S, RESULTS R
WHERE   $C$
```

S will range over 4 tuples in STUDENTS, R will range over 8 tuples in RESULTS. In principle, all $4 \cdot 8 = 32$ combinations will be considered in condition C:

Naive(!) implementation

```
foreach S $\in$ STUDENTS do
    foreach R $\in$ RESULTS do
        if $C$ then
            print $A_1, \ldots, A_n$
        fi
    od
od
```

This formulation of joins in SQL resembles the (derived) join definition:

$$\bowtie \;\equiv \pi \circ \sigma \circ \times,$$

where $\circ$ is operator composition (from right to left).

Implementing joins

- A good DBMS will use a **better evaluation algorithm** (depending on the condition *C*).
  This is the task of the ***query optimizer.*** *For example, if C contains the join condition* `S.SID = R.SID`*, the DBMS might loop over the tuples in* `RESULTS` *and find the corresponding* `STUDENTS` *tuple by using an* ***index*** *over* `STUDENT.SID` *(many DBMS automatically create an index over the key attributes).*

- In order to understand the **semantics** of a query, however, the simple nested `foreach` algorithm suffices.
  The query optimizer may use any algorithm that produces the ***exact same output****, although possibly in different tuple order.*

Explicit join conditions in the WHERE clause (1)

In that join syntax, **join conditions** needs to be explicitly specified in the WHERE clause.

Example

```
SELECT   DISTINCT 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’
```

Output of this query?

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

Explicit join conditions in the WHERE clause (2)

Guideline: it is almost always an **error** if there are two tuple variables which are **not linked** (directly or indirectly) via join conditions.

In this query, all three tuple variables 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’
```

The tuple variable connection works as follows:

$$\begin{tikzcd} \boxed{S} \arrow[r, dash, "\texttt{S.SID = R.SID}"'] & \boxed{R} \arrow[r, dash, "\begin{array}{c}\texttt{R.CAT = E.CAT}\\ \texttt{AND R.ENO = E.ENO}\end{array}"'] & \boxed{E} \end{tikzcd}$$

The conditions correspond to **key–foreign key–relationships** between tables. Omission of a join condition will usually lead to numerous duplicates in the query result.

The use of `DISTINCT` *does not fix the error in such a case!*

Join graph (1)

✎ Formulate the following query in SQL

"Which are the topics of all exercises solved by Ann Smith?"

To formulate this query,

- consider that Ann Smith is a student, requiring a tuple variable, S say, over STUDENTS and the identifying condition `S.FIRST = 'Ann' AND S.LAST = 'Smith'`.
- Exercise topics are of interest, so a tuple variable E over EXERCISES is needed, and the following piece of SQL can already be generated:

  ```
  SELECT DISTINCT E.TOPIC
  ```

  Several exercises may have the same topic (hence the `DISTINCT`).

Join graph (2)

- Note: S and E are still **unconnected.**
- The **connection graph** (join graph) of the tables in a database schema (edges correspond to foreign key relationships) helps in understanding the connection requirements:

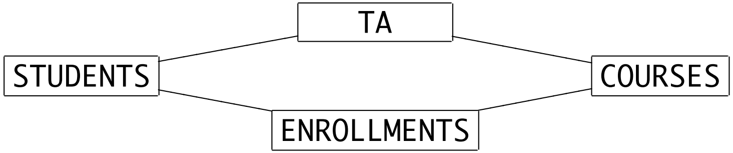
$$\begin{tikzcd}\boxed{\texttt{STUDENTS}} \arrow[r, dash] & \boxed{\texttt{RESULTS}} \arrow[r, dash] & \boxed{\texttt{EXERCISES}}\end{tikzcd}$$

  ▶ We see that the S—E connection is **indirect** and needs to be established via a tuple variable R over RESULTS:

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

Join graph (3)

- It is not always that trivial. The connection graph may contain **cycles**, which makes the selection of the “right path” more difficult (and error-prone).
- Consider a course registration database that also contains TA (“*Hiwi*”) assignments:



Unnecessary Joins (1)

Do not join **more** tables than needed.

- Query will run slowly if the optimizer overlooks the redundancy.

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
```

Will the following query produce the same results?

```
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
```

Joins—Modern syntax according to SQL-2 (SQL:1992)

SQL-92 supports the following join types (parts in [ ] optional)

`[INNER] JOIN`	Usual join.
`LEFT [OUTER] JOIN`	Preserves rows of left table.
`RIGHT [OUTER] JOIN`	Preserves rows of right table.
`FULL [OUTER] JOIN`	Preserves rows of both tables.
`CROSS JOIN`	Cartesian product.
`UNION JOIN`	Pads columns of both tables with `NULL`.[38]

✎ **Semi-join is easily expressed in the** `SELECT` **clause…**

[38] *SQL-92 Intermediate Level*: rarely found implemented in DBMSs.

Join Syntax in SQL-92 (2)

- The **join predicate** may be specified as follows:
  - Keyword NATURAL prepended to join operator name.
  - ON⟨*Condition*⟩ appended to join operator name.
  - `USING` $(A_1, \ldots, A_n)$ appended to join operator name.

  USING specified columns A_i appearing in both join inputs R and S. The effective join predicate then is $R.A_1 = S.A_1$ AND $\cdots$ AND $R.A_n = S.A_n$.

- `CROSS JOIN` and `UNION JOIN` have no join predicate.

✎ `UNION JOIN` **not implemented in today's DBMS products.**

Simulate R `UNION JOIN` S.

Remarks on outer join and selection

✎ Will tuples with CAT = ’M’ appear in the output?

```
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
```

- Conditions filtering the **left table** make little sense in a **left outer join predicate.**

  … The left outer join semantics will make the “filtered” tuples appear anyway (as join partners for unmatched RESULTS tuples).

10.3 Duplicate Elimination

A core difference between SQL and relational algebra is that **duplicates have to be eliminated explicitly** in SQL.

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 a query might yield unwanted duplicate tuples, the DISTINCT modifier may be applied to the SELECT clause to request explicit duplicate row elimination.

Example

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

CAT	ENO
H	1
H	2
M	1

To emphasize that there will be duplicate rows (and that these are wanted in the result), SQL provides the ALL modifier.[39]

[39] SELECT ALL is the default.

Duplicate Elimination (3)

Sufficient condition for superfluous DISTINCT:

1. Let $\mathcal{K}$ be the set of attributes selected for output by the SELECT clause.
2. Add to $\mathcal{K}$ attributes A such that $\quad A = c \quad$ (constant c) appears in the WHERE clause.
   Here we assume that the WHERE *clause specifies a conjunctive condition.*
3. Add to $\mathcal{K}$ attributes A such that $\quad A = B \quad$ $(B \in \mathcal{K})$ appears in the WHERE clause. If $\mathcal{K}$ contains a key of a tuple variable, add all attributes of that variable.
   Repeat 3 until $\mathcal{K}$ stable.
4. If $\mathcal{K}$ **contains a key of every tuple variable** listed under FROM, then DISTINCT is superfluous.

Duplicate Elimination (4)

Assume (FIRST, LAST) is an alternative key for STUDENTS.

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

1. Initialize $\mathcal{K} \leftarrow \{$S.FIRST, S.LAST, R.ENO, R.POINTS$\}$.
2. $\mathcal{K} \leftarrow \mathcal{K} \cup \{$R.CAT$\}$ because of the conjunct R.CAT = 'H'.
3. $\mathcal{K} \leftarrow \mathcal{K} \cup \{$S.SID, S.EMAIL$\}$ because $\mathcal{K}$ contains a key of STUDENTS (S.FIRST, S.LAST).
3. $\mathcal{K} \leftarrow \mathcal{K} \cup \{$R.SID$\}$ because of the conjunct S.SID = R.SID.
4. $\mathcal{K}$ contains a key of STUDENTS (see above) and RESULTS (R.SID, R.CAT, R.ENO), thus DISTINCT is superfluous.

N.B. If FIRST, LAST were no key of STUDENTS, the test would (rightly) fail.

Instead of a(n intermediate) summary: Some SQL traps

- **Missing join conditions** (very common).
- **Unnecessary joins** (may slow query down significantly).
- **Self joins:** incorrect treatment of multiple tuple variables which range over the same relation (missing (in)equality conditions).
- **Unexpected duplicates**, often an indicator for faulty queries (adding `DISTINCT` is no cure here).
- **Unnecessary** `DISTINCT`.
  Although today's query optimizer are probably more "clever" than the average SQL user in proving the absence of duplicates.

10.4 Advanced SQL query syntax

Non-monotonic constructs

- SQL queries using only the constructs introduced above compute **monotonic functions** on the database state: if further rows gets **inserted**, these queries yield a **superset** of rows.
- However, not all queries behave monotonically in this way (remember: “*Find students who have not yet submitted any homework.*”)
  In the current DB state, `Maria Brown` would be a correct answer. `INSERT INTO RESULTS VALUES (104, ’H’, 1, 8)` would invalidate this answer.
- Obviously, such queries *cannot* be formulated with the SQL constructs introduced so far.

Non-monotonic behaviour

- In natural language, queries containing formulations like *"there is no"*, *"does not exists"*, *etc.*, indicate non-monotonic behaviour (**existential quantification**).
- Furthermore, *"for all"*, *"the minimum/maximum"* also indicate non-monotonic behaviour: in this case, a violation of a **universally quantified** condition must not exist.
- In an equivalent SQL formulation of such queries, this ultimately leads to a test whether a certain **query yields a (non-)empty result.**

NOT IN (1)

With IN ($\in$) and NOT IN ($\notin$) it is possible to check whether an attribute value appears in a set of values computed by another SQL **subquery.**

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 evaluation of the **main query** starts.

Example

STUDENTS

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

Subquery result

SID
101
101
102
102
103

Then, for every STUDENTS tuple, a matching SID is searched for in the subquery result. If there is none (NOT IN), the tuple is output.

NOT IN (3)

Since the **(non-)existence of particular tuples** does *not* depend on multiplicity, we may equivalently use DISTINCT in the subquery.

Example

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

The effect on the performance depends on the DBMS and the data (sizes).

A reasonable optimizer will know about the NOT IN semantics and will decide on duplicate elimination/preservation itself, esp. because IN (NOT IN) may efficiently be implemented via **semijoin** and **antijoin** if duplicates are eliminated.

NOT IN (4)

Topics of homeworks that were solved by at least one student.

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

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 (5)

On the SELECT clause of nested subqueries ...

- In SQL-89, the subquery is required to deliver a **single output column.** This ensures that the subquery is a set (or multiset) and not an arbitrary relation.

- In SQL-92, comparisons were extended to the tuple level.[40] It is thus valid to write, *e.g.*:

```
                    ⋮
WHERE (A,B) NOT IN (SELECT C,D FROM …)
```

[40] However, also see EXISTS below.

`NOT EXISTS` (1)

- The construct `NOT EXISTS` enables the main (or outer) query to check whether the **subquery result is empty.**
- In the subquery, tuple variables declared in the `FROM` clause of the outer query may be referenced.
  You may also do so for `IN` *subqueries but this yields unnecessarily complicated query formulations (bad style).*
- In this case, the outer query and subquery are **correlated.** In principle, the **subquery** has to be evaluated for every assignment of values to the outer tuple variables. (The subquery is "*parameterized*".)

NOT EXISTS (2)

Students who 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)
```

Tuple variable S loops over the four rows in STUDENTS. Conceptually, the subquery is evaluated four times (with S.SID bound to the current SID value).

Again: the DBMS is free to choose a more efficient equivalent evaluation strategy (*cf.* **query unnesting**).

NOT EXISTS (3)

- "First," S is bound to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
101	Ann	Smith	...

- In the subquery, S.SID is "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

… Since the result is non-empty, the NOT EXISTS in the outer query is not satisfied **for this** S.

NOT EXISTS (4)

- "Finally," S is bound to the STUDENTS tuple

SID	FIRST	LAST	EMAIL
104	Maria	Brown	...

- In the subquery, S.SID is "replaced by" 104 and the following query is executed:

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

SID	CAT	ENO	POINTS
(no rows selected)			

... Since the result is empty, the NOT EXISTS in the outer query is satisfied and Maria Brown is output.

NOT EXISTS (5)

While in the subquery tuple variables from outer query may be referenced, the **converse is illegal**.

Wrong!

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

N.B. Compare this to **variable scoping** (global/local variables) in block-structured programming languages (Java, C). Subquery tuple variables declarations are "local."

NOT EXISTS (6)

Non-correlated subqueries with NOT EXISTS are almost always an indication of an error.

Wrong!

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

- If there is at least one tuple in RESULTS, the overall result will be empty.

N.B. Non-correlated subqueries evaluate to a relation **constant** and may make perfect sense (*e.g.*, when used with (NOT) IN).

NOT EXISTS (7)

Note: it is legal SQL syntax to specify an arbitrarily complex `SELECT` clause in the subquery, however, this does not affect the existential semantics of `NOT EXISTS`.

`SELECT *` … documents this quite nicely. Some SQL developers prefer `SELECT 42` … or `SELECT null` … or similar SQL code.

Again, the query optimizer will know the `NOT EXISTS` semantics such that the exact choice `SELECT` clause is of no importance.

NOT EXISTS (8)

It is legal SQL syntax 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')
```

Can we reformulate the above without using EXISTS?

Universal quantification: “For all” (1)

- SQL does *not* offer a universal quantifier, only the existential quantifier `EXISTS`.[41]
- Of course, this is no problem because $\forall X : \varphi \Leftrightarrow \neg\exists X : \neg\varphi$.
- In TRC, the query asking for the maximum number of points for homework 1 reads

$$\begin{array}{l}\{\mathtt{X.POINTS} \mid \mathtt{X} \in \mathtt{RESULTS} \wedge \mathtt{X.CAT} = \mathtt{'H'} \wedge \mathtt{X.ENO} = 1 \wedge \\ \qquad \forall \mathtt{Y} : (\mathtt{Y} \in \mathtt{RESULTS} \wedge \mathtt{Y.CAT} = \mathtt{'H'} \wedge \mathtt{Y.ENO} = 1) \\ \qquad\quad \Rightarrow \mathtt{Y.POINTS} \leqslant \mathtt{X.POINTS}\}\end{array}$$

or, equivalently, now:

$$\begin{array}{l}\{\mathtt{X.POINTS} \mid \mathtt{X} \in \mathtt{RESULTS} \wedge \mathtt{X.CAT} = \mathtt{'H'} \wedge \mathtt{X.ENO} = 1 \wedge \\ \qquad \neg\exists \mathtt{Y} : (\mathtt{Y} \in \mathtt{RESULTS} \wedge \mathtt{Y.CAT} = \mathtt{'H'} \wedge \mathtt{Y.ENO} = 1 \\ \qquad\quad \wedge \mathtt{Y.POINTS} > \mathtt{X.POINTS})\}\end{array}$$

[41] However, see `>= ALL` below

“For all” (2)

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)
```

In natural language: *“A result* X *for homework 1 is selected, if there is no result* Y *for this exercise with more points than* X.*”*

Nested Subqueries

Subqueries may 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'))
```

Read: “List those students S, where there is no homework E for which there exists no entry for (S,E) in the RESULTS.”

Common errors (1)

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 the query compute?

Common errors (2)

Subqueries bring up the concept of **variable scoping** (just like in programming languages) and related pitfalls.

Return those students who did not solve homework 1.

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

Common errors (3)

Find those students who have neither submitted a homework nor participated in any exam.

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

What is the error in this query?

1. Is this syntactically correct SQL?
2. What is the output of this query?
3. If the query is faulty, correct it.

ALL, ANY, SOME (1)

SQL allows to compare a **single value** with all values in a set (computed by a subquery).
Such comparisons may be **universally** (ALL) or **existentially** (ANY) quantified.

Which student(s) got the maximum number of points for homework 1?

```
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)
```

N.B. The use of >= is important here!

ALL, ANY, SOME (2)

The following is equivalent to the above query:

Using 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 NOT X.POINTS < ANY (SELECT Y.POINTS
                                 FROM   RESULTS Y
                                 WHERE  Y.CAT = 'H'
                                        AND Y.ENO = 1)
```

Note that ANY (ALL) do *not* extend SQL's expressiveness, since, *e.g.*

$$A < \texttt{ANY (SELECT } B \texttt{ FROM} \cdots \texttt{ WHERE} \cdots\texttt{)}$$
$$\equiv$$
$$\texttt{EXISTS (SELECT 1 FROM} \cdots \texttt{ WHERE} \cdots \texttt{ AND } A < B\texttt{)}$$

ALL, ANY, SOME (3)

Syntactical remarks on comparisons with subquery results:

1. ANY and SOME are synonyms.
2. *x* IN *S* is equivalent to *x* = ANY *S*.
3. The subquery must yield a **single result column**.
4. If none of the keywords ALL, ANY, SOME are present, the subquery must **yield at most one row** (single value subquery).
   With 3, this ensures that the comparison is performed between atomic (non-set) values. An empty subquery result is equivalent to NULL.

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)
```

Comparisons with subquery results (note: no ANY, SOME, ALL) are possible, iff the subquery returns **at most one row.**

[Why is this guaranteed here?]

Use (non-data dependent) constraints to ensure this condition, the DBMS will yield a runtime error, if the subquery returns two or more rows!

Single value subqueries (2)

If the subquery has an **empty result**, the **null value** is returned.

Bad style!

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

Rather, use ...?

Orthogonal SQL: Nesting in the FROM clause

- Since the result of a SQL query is a **table**, it seems most natural to use a subquery result wherever a table might be specified, *i.e.*, in the FROM clause.
- This principle of (query) language construction is known as **orthogonality**: language constructs may be combined in arbitrary fashion as long as the semantic/typing/...rules of the language are obeyed.
  Relational algebra is an orthogonal query language.
- SQL versions prior to SQL-92 were not orthogonal in this sense.
- Apart from stepwise querying, like in algebra, one other use of subqueries under FROM are **nested aggregations** (see further below).

Nested subqueries under FROM (1)

In the following example, the join of RESULTS and EXERCISES is computed in a subquery (this might result from a **view definition**, see below).

Points (in %) achieved in homework exercise 1.

```
SELECT  X.SID, (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) AS X
WHERE   X.CAT = 'H' AND X.ENO = 1
```

N.B. Inside the subquery, tuple variables introduced in the same FROM clause **may not be referenced!**

Nested subqueries under FROM (2)

A **view declaration** registers a query[42] under a given name in the DB.

View: homework points

```
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'
```

Subsequently, queries may use views just like stored tables.

Querying the view

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

Views may be thought of as **subquery macros** that get substituted for the view's name in the FROM clause, yielding a nested subquery.

[42] Not a query result!

Aggregation functions

- **Aggregation functions** are functions from a set (or multiset, list, ...) to a single value, *e.g.*,
$$\min \{42, 57, 5, 13, 27\} = 5 \quad .$$
- Aggregation functions are used to summarize an entire set of values.
  In the DB literature, aggregation functions are also known as **group functions** or **column functions:** the values of an entire column (or partitions of these values) form the input to such functions.
- Typical use: statistics, data analysis, report generation.

Aggregation functions in SQL (1)

- SQL-92 defines five main aggregation functions
  `COUNT`, `SUM`, `AVG`, `MAX`, and `MIN`
  - Some DBMS define further aggregation functions, such as: `CORRELATION`, `STDDEV`, `VARIANCE`, `FIRST`, `LAST`, ...
- Any **commutative** and **associative** binary operator with a neutral element can be extended ("*lifted*") to work on set-valued arguments (*e.g.*, `SUM` corresponds to +).

✎ Commutative and associative, neutral element?

Why do we require these properties of the operators?

Aggregation functions in SQL (2)

- Note: some aggregation functions are sensitive to **duplicates** (*e.g.*, SUM, COUNT, AVG), some are insensitive (*e.g.*, MIN, MAX).
- For the first type, SQL allows to explicitly request to ignore duplicates, *e.g.*: "··· `COUNT(DISTINCT A)` ···"
- **Simple aggregations** feed the value set of an **entire column** into an aggregation function.
  Below, we will discuss partitioning (or **grouping**) of columns.

How many students in the current database state?

```
SELECT   COUNT(*)
FROM     STUDENTS
```

COUNT(*)
4

Best and average result for homework 1?

```
SELECT MAX(POINTS), AVG(POINTS)
FROM   RESULTS
WHERE  CAT = 'H' AND ENO = 1
```

MAX(POINTS)	AVG(POINTS)
10	8

Examples: Simple aggregations ...

How many students have submitted a homework?

```
SELECT   COUNT(DISTINCT SID)
FROM     RESULTS
WHERE    CAT = 'H'
```

COUNT(DISTINCT SID)
3

What is the total number of points student 101 got for her homeworks?

```
SELECT   SUM(POINTS) AS "Total Points"
FROM     RESULTS
WHERE    SID = 101 AND CAT = 'H'
```

Total Points
18

… Examples: Simple aggregations

✎ **What average percentage of the maximum points did the students reach for homework 1?**

✎ **Homework points for student 101 plus 3 bonus points.**

Aggregation queries and SQL semantics

Basically, there are **three different types of queries** in SQL:

1. Queries without aggregation functions and without `GROUP BY` and `HAVING`. (Discussed above.)
2. Queries with aggregation functions in the `SELECT` clause but no `GROUP BY` (simple aggregations). Yield exactly one row.
3. Queries with `GROUP BY`.

Each type has different syntax restrictions and is **evaluated** in a different way.

Notice again: when speaking of "evaluation", we refer to the SQL *semantics* here. A DBMS is free to implement these semantics as it sees fit.

Possible evaluation (1)

1. First, evaluate the FROM clause.
   Conceptually, form all possible tuple combinations of the source tables (Relational product).

2. Evaluate the WHERE clause.
   The Relational product produced in 1 is filtered (restricted) and only those tuple combinations satisfying the filter condition remain.

3. **If no aggregation**, GROUP BY or HAVING: evaluate the SELECT clause.
   Evaluate projection list (terms, scalar expressions) for each tuple combination produced in 2 and print resulting tuples.

Possible evaluation (2)

4. **For simple aggregation:** add column values received from phase 2 to sets/multisets that will be the input to the aggregation function(s).
   - If no `DISTINCT` is used or if the aggregation function is **idempotent** (`MIN`, `MAX`), the aggregation results may be **incrementally** computed, *no* temporary sets need to be maintained (see next slide).
   - Print the single row of aggregated value(s).

Possible evaluation (3)

Simple aggregation, no DISTINCT

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

Possible evaluation strategy (no intermediate storage required)

```
$agg_1 \leftarrow 0$;        /* neutral element for + */
$agg_2 \leftarrow 0$;        /* neutral element for +1 */
foreach E $\in$ EXERCISES do
    if E.CAT = 'H' then
        $agg_1 \leftarrow agg_1 + $ E.MAXPT;                /* incrementally maintain SUM */
        $agg_2 \leftarrow agg_1 + 1$;                       /* incrementally maintain COUNT */
    fi
od
print $agg_1, agg_2$
```

Restrictions

- Aggregations may not be nested (makes no sense).
- Aggregations may not be used in the WHERE clause.

Wrong!

```
··· WHERE SUM(A) > 100 ···
```

- If an aggregation function is used and no GROUP BY is used (simple aggregation), **no attributes** may appear in the SELECT clause.

Wrong!

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

... but see GROUP BY below.

Null values and aggregations

- Usually, null values are **ignored** (filtered out) before the aggregation operator is applied.
  - Exception: `COUNT(*)` counts null values (`COUNT(*)` counts rows, not attribute values).
- If the aggregation input set is empty, aggregation functions yield `NULL`.
  - Exception: `COUNT` returns `0`.

  This seems counter-intuitive, at least for `SUM` (where users might expect `0` in this case). However, this way a query can detect the difference between two types of empty input: (1) all column values `NULL`, or (2) no tuple qualified in `WHERE` clause.

Grouping: GROUP BY

SQL's GROUP BY construct **partitions** the tuples of a table into **disjoint groups.** Aggregation functions may then be applied for each **tuple group** separately.

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

All tuples agreeing in their ENO values (*i.e.*, belonging to the same homework) form a group for aggregation.

Inner workings of GROUP BY

(After evaluation of the FROM and WHERE clauses,) incoming tuples are **partitioned** into groups based on **value equality** of GROUP BY attributes. The intermediate result can be thought of as a *nested relation*:

ENO-based grouping/nesting formed by the above example query

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

- Aggregations are subsequently done on a per-group basis (yielding as many rows as groups).
- This construction can *never* produce empty groups (a COUNT(*) will never result in 0).

Output of GROUP BY queries

Contents of SELECT clause in the presence of GROUP BY:

- Since only the GROUP BY attributes have an **atomic, unique value for every group**, only these attributes may be used in the SELECT clause.
  A reference to any other attribute is illegal.

- The other attributes may be subject to **aggregation**, though.

Wrong! (Because of reference to 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
```

N.B. This query is illegal, even though E.ENO is key and thus E.TOPIC would be unique! Again, SQL uses *purely syntactic* constraints.

Adding output columns

Grouping by E.ENO *and* E.TOPIC is possible and will yield the desired result.

Example

```
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.Alg.	8
2	SQL	8.5

The DBMS now has a simple **syntactic clue** that the value of E.TOPIC will be unique.

Adding more grouping columns

✎ **Is there any semantic difference between these queries?**

1.
```
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
```

2.
```
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 subtleties

- The ordering of attributes in the GROUP BY clause is not important.
- Grouping makes no sense, if the GROUP BY attributes contain a **key** (if only one table is listed in the FROM clause): **each group will contain a single row** only.
- Duplicates should be eliminated with DISTINCT, although such elimination could also be realized via GROUP BY

✎ **Grouping without aggregation:** DISTINCT**...?**

This is an **abuse** of GROUP BY and should be avoided.

Group-based filtering

- Remember: aggregation functions may not be used in the WHERE clause.
- With GROUP BY, however, it may make sense to **filter out entire groups** based on some aggregated group property.
  For example, only groups of size greater than *n* tuples may be significant.
- This is possible with SQL's HAVING clause.
  - The condition in the HAVING clause may reference aggregation functions and the GROUP BY attributes.

Group-based filtering: Example

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

N.B. The WHERE clause refers to *single tuples*, the HAVING condition applies to *entire groups* (in this case: all tuples containing the homework results of a student).

Conceptual "execution order" of SQL's SFW clauses

Flashback: (Coarse) semantics in the "pre-GROUP BY/HAVING" era

1. take the product of the tables in the FROM clause;
2. apply selection with WHERE-predicate;
3. finally project according to SELECT clause.

This should now read

1. evaluate nested queries in FROM clause;
2. compute joins/products of tables/subexpressions in FROM clause;
3. select *rows* according to WHERE clause (incl. proper evaluation of nested subqueries);
4. group result according to GROUP BY columns/expressions;
5. select *groups* according to HAVING condition;
6. project (incl. dup. elimination & aggregations) according to SELECT.

Conditions in WHERE vs. HAVING

If a condition refers to GROUP BY attributes only (but not aggregations), it may be placed under WHERE *or* HAVING.

✎ **Somewhat strange use of HAVING condition**

1.
```
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
```

2.
```
SELECT    FIRST, LAST
FROM      STUDENTS S, RESULTS R
WHERE     S.SID = R.SID
GROUP BY  S.SID, FIRST, LAST
HAVING    SUM(POINTS) >= 18
```

How many groups are produced for these two queries?

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)
```

- The aggregate in the subquery is guaranteed to yield exactly one row as required.
- Remember: our earlier solution to this problem was using `ANY/ALL`.

Aggregation subqueries (2)

In SQL-92, aggregation subqueries may be placed into the SELECT clause. This may replace GROUP BY.

The homework points of the individual students.

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

N.B. Again, the subquery can be (and typically will be) correlated!

Nested Aggregations

Nested aggregations require a subquery in the FROM clause.

What is the average number of homework points (excluding those students who did not submit anything)?

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

X

SID	HW_POINTS
101	18
103	18
103	5

AVG(X.HW_POINTS)
13.67

Maximizing Aggregations (1)

Who has the best overall homework result (maximum 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
                              WHERE     CAT = ’H’
                              GROUP BY  SID)
```

N.B.

- Conditions in the HAVING clause can contain nested subqueries!
- Alternatively, we could use a view to solve this problem (next slide).

Maximizing Aggregations (2)

View: total number of homework points for each student.

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

Alternative formulation of query on previous slide.

```
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)
```

N.B. This (use of views) is a standard way of nesting aggregation functions in SQL.

Conditional expressions: Case analysis

We have seen earlier that UNION is a common way (in algebra and in SQL) to deal with case analysis.

- In SQL, it is possible to combine (*on the outermost nesting level only*) the results of two queries by UNION.
- UNION is strictly needed, since there is no other method to construct one result column that draws from different tables/columns.
  This is necessary, for example, if specializations of a concept ("subclasses") are stored in separate tables. For instance, there may be GRADUATE_COURSES and UNDERGRADUATE_COURSES tables (both of which are specializations of the general concept COURSE).[43]

[43] see our discussion of how to map generalization hierarchies to the relational model

SQL's UNION operator

- The UNION operand subqueries must return tables with the *same number of columns* and *compatible data types*.
  Columns correspondence is by column **position** (1st, 2nd, ...). Column names need not be identical (IBM DB2, for example, creates artificial column names 1, 2, ..., if necessary. Use column renaming via AS if column names matter.
- SQL distinguishes between
  - UNION: like RA $\cup$ with **duplicate elimination**, and
  - UNION ALL: **concatenation** (duplicates retained).
- Other SQL-92 set operations: EXCEPT ($-$), INTERSECT ($\cap$).[44]

[44]These do *not* add to the expressivity of SQL. Proof?

Examples: UNION for case analysis (1)

Total number of homework points for every students (or 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')
```

Examples: UNION for case analysis (2)

Assign student grades based on homework 1.

```
   SELECT  S.SID, S.FIRST, S.LAST, 'A' AS 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' AS 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
   ...
```

See the same example in our Relational Algebra discussion.

Conditional expressions (1)

While UNION is the **portable way** to conduct a case analysis, sometimes a **conditional expression** suffices and is more efficient.

- Here, we will use the SQL-92 (and, *e.g.*, DB2) syntax. Conditional expression syntax varies between DBMSs. Oracle uses DECODE($\cdots$), for example.

Print the full exercise category name for the results of Ann Smith.

```
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'
```

Conditional expressions (2)

- A typical application of
  a conditional expression is to **replace a null value** by another (non-null) value Y:

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

- In SQL-92, this may be abbreviated to

  ```
  ··· COALESCE (X, Y)···
  ```

List the e-mail addresses of all students.

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

- Conditional expressions are regular terms, so they may be input for other functions, comparisons, or aggregate functions.

Sorting the output

- If query output is to be read by humans, enforcing a certain **tuple order** greatly helps in interpreting the result.
  Without such an ordering, the sequence of output rows is meaningless, depends on the internal algorithms selected by the query optimizer to evaluate the query and may change from version to version or even query to query.
- In a SQL RDBMS, however, the query logic and the subsequent output formatting are completely **independent** processes.
  DBMS front-ends offer a variety of formatting options (page breaks, colorization of column values, *etc.*).

SQL's ORDER BY clause

Specify a **list of sorting criteria** in an ORDER BY clause.

An ORDER BY clause may specify multiple attribute names. The second attribute is used for tuple ordering, if they agree on the first attribute, and so on (**lexicographic ordering**).

Homework results sorted by exercise (best result first). In case of a tie, sort alphabetically by student 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
```

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

Notes on sorting

- In some application scenarios it is necessary to **add columns** to a table to obtain suitable **sorting criteria.** Some examples:
  - Print homework results in the order homeworks (`CAT = 'H'`), midterm exam (`'M'`), and final exam (`'F'`).
  - In a list of universities, `'Uni Konstanz'` should be listed under K, not U.
  - If the students names were stored in the form `'Ann␣ Smith'`, sorting by last name is more or less impossible.[45]
- **Null values** are all listed first or all listed last in the sort sequence (IBM DB2: all first).
- Since the effect of ORDER BY is purely "cosmetic", ORDER BY may *not* be applied to a subquery.
  - This also applies if multiple queries are combined via UNION. Place ORDER BY at the bottom of the query to sort all tuples.

---

[45]This is related to an important **DB design time question:** "*What do I need to do with the query outputs?*"

10.5 Recursive Queries in SQL

Since 1999, the SQL standard ("SQL-3") includes a *recursive* query facility.

Syntax

```
WITH [RECURSIVE] ⟨rectable⟩ (⟨attr⟩₁, ⟨attr⟩₂, . . . , ⟨attr⟩ₙ)
     AS ( ⟨SFW-Statement⟩₁                        -- initialization
               UNION ALL
          ⟨SFW-Statement⟩₂ )                      -- recursive step

SELECT [DISCTINCT] ⟨att-list⟩, ⟨aggregations⟩
FROM ⟨rectable⟩
[WHERE ...] [GROUP BY ...] [HAVING ...] [ORDER BY ...]
```

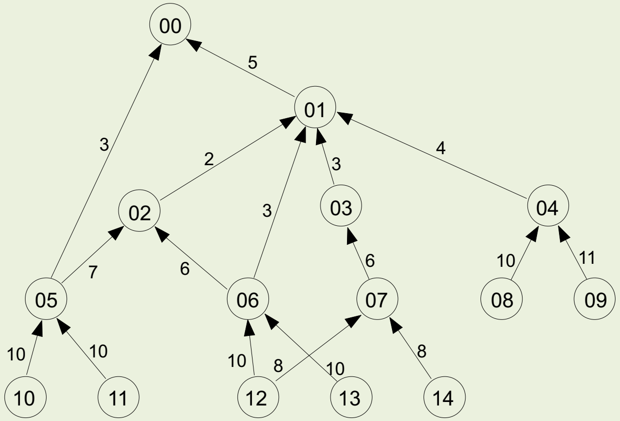
- ⟨SFW-Statement⟩$_2$ must not contain a `DISTINCT` clause
- must use `UNION` <u>`ALL`</u>
- ⟨SFW-Statement⟩$_2$ will be a join query involving ⟨rectable⟩
- recursion stops when no new rows found (user's responsibility!)

“Bill of materials”

Example

Constr		
sup	*sub*	*qty*
00	01	5
00	05	3
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6
03	07	6
04	08	10
04	09	11
05	10	10
05	11	10
06	12	10
06	13	10
07	12	8
07	14	8

Graph representation:



“Bill of Materials” in SQL

Get all parts, including quantity, necessary for Part 01.

First attempt:

```
WITH RecRel (sup, sub, qty) AS
        ( SELECT * FROM Constr WHERE sup = 01
          UNION  ALL
          SELECT Constr.* FROM RecRel, Constr WHERE RecRel.sub = Constr.sup )

SELECT * FROM RecRel
```

N.B. This is not yet it! (Why?)

Result of this SQL query

Not what we wanted...!

result	sup	sub	qty	
	01	02	2	
	01	03	3	init
	01	04	4	
	01	06	3	
	02	05	7	
	02	06	6	
	03	07	6	
	04	08	10	1. rec. step
	04	09	11	
	06	12	10	
	06	13	10	
	05	10	10	
	05	11	10	
dup $\Longrightarrow$	06	12	10	2. rec. step
dup $\Longrightarrow$	06	13	10	
	07	12	8	
	07	14	8	

Improving on the SQL solution

- Removal of the two identical duplicates in the result of our first attempt can be achieved by adding DISTINCT to the final SFW.
- This does not solve the problem, though.
- We need to *aggregate* the quantities "along the way"...

Solution

```
WITH RecRel (sup, sub, qty) AS
     ( SELECT * FROM Constr WHERE sup = 01
       UNION  ALL
       SELECT RecRel.sup, Constr.sub, RecRel.qty * Constr.qty
         FROM RecRel, Constr WHERE RecRel.sub = Constr.sup )

SELECT sup, sub, SUM(qty) AS total FROM RecRel GROUP  BY sup, sub
 [ ORDER  BY sup, sub ]
```

Result now

This looks much better!

result	sup	sub	total
	01	02	2
	01	03	3
	01	04	4
	01	05	14
	01	06	15
	01	07	18
	01	08	40
	01	09	44
	01	10	140
	01	11	140
	01	12	294
	01	13	150
	01	14	144

Final example

Restrict depth of recursion to 2 for the previous query

```
WITH RecRel (lvl, sup, sub, qty) AS
        ( SELECT 1, Constr.* FROM Constr WHERE sup = 01
          UNION  ALL
          SELECT RecRel.lvl + 1, RecRel.sup, Constr.sub, RecRel.qty * Constr.qty
              FROM RecRel, Constr
          WHERE RecRel.sub = Constr.sup AND RecRel.lvl < 2)

SELECT sup, sub, SUM(qty) AS total FROM RecRel GROUP  BY sup, sub
 [ ORDER  BY sup, sub ]
```

11. SQL: More Than a Query Language

After completing this chapter, you should be able to:

- enumerate and explain some the functionality of SQL that goes beyond **querying**, to make SQL a full-scale database language (QL, DDL, DML, ...),
- **define database schemas** (domains, tables, keys, ...) in SQL,
- write statements in SQL that **modify the current database state**,
- explain various types of integrity constraints and how to deal with them during updates,
- make use of view definitions and explain problems w.r.t. view updates,
- define access control strategies and grant/revoke privileges,
- work with programming language interfaces to SQL databases.

Database Language Functionalities

The SQL standard defines much more than “just” the query operators of the language. A complete database language also has to offer statements for

- DDL: the definition and modification of database schemas, *e.g.*,
  - named schemas and subschemas,
  - named tables, with attributes, domains, keys, foreign keys, other constraints, views,
- DML: the manipulation of stored database contents,
  - insertion, updating, deletion of rows
- Misc: the administration of the DBMS, *e.g.*,
  - creating users and assigning roles/permissions to them,
  - allocating storage space and assigning tables to containers,
  - index creation and other physical schema maintenance tasks.

11.1 DDL: Data Definition Capabilities of SQL

The basic task of a **D**ata **D**efinition (sub-) **L**anguage is to be able to declare the elements of a database schema, *i.e.*, to communicate the result of the database design process to the DBMS.

- A SQL database is structured into named **schemas**,
- each schema contains a set of **tables**,
- for each table, we specify its **attributes** with their **domains**,
- (optionally) the **primary key** and alternative keys,
- (optionally) the **foreign keys**, if present;
- in addition, we can specify further **integrity constraints**
- … *more functionality to be added later* …

CREATE TABLE

SQL's CREATE TABLE (simplified syntax)

```
CREATE  [TEMPORARY] TABLE ⟨TableName⟩
     (  ⟨AttrName⟩ ⟨Domain⟩ [⟨AttrConstraints⟩],
        ...
     )  [⟨TableConstraints⟩];
```

- TEMPORARY: table exists only within the creating transaction
- ⟨AttrConstraints⟩ define integrity constraints on a single attribute, *e.g.*,
  - [NOT] NULL: whether or not null values are permitted,
  - PRIMARY KEY or UNIQUE for single-attribute (candidate) keys,
  - REFERENCES for single-attribute foreign keys,
  - CHECK conditions on this attribute's values.
- ⟨TableConstraints⟩ may refer to multiple attributes of the table.

Example in our Homework database

Example

```
CREATE TABLE EXERCISES (
   CAT      CHAR(1),
   ENO      DECIMAL(3),
   TOPIC    VARCHAR(30)  NOT NULL,
   MAXPT    DECIMAL(2)   NOT NULL CHECK (MAXPT > 0)
   PRIMARY KEY (CAT, ENO)
)
CREATE TABLE RESULTS (
   SID      DECIMAL(3)   REFERENCES STUDENTS,
   CAT      CHAR(1),
   ENO      DECIMAL(3),
   POINTS   DECIMAL(2)   NOT NULL
   PRIMARY KEY (SID, CAT, ENO)
   FOREIGN KEY (CAT, ENO) REFERENCES EXERCISES
   CHECK    (POINTS <= (SELECT MAXPT FROM EXERCISES E
                       WHERE  E.ENO=RESULTS.ENO AND E.CAT=RESULTS.CAT))
)
```

Remarks

- Constraints can be **named**, *e.g.*, `CONSTRAINT MAX_GT_ZERO CHECK (MAXPT > 0)`, such that checking this constraint can be switched on/off (see below).
- Attributes belonging to the primary key are automatically `NOT NULL`.
- Key attributes referenced by a foreign key need not have the same name as in the referencing (foreign key) relation, *e.g.*, `REFERENCES STUDENTS(StudID)`.
- In addition to constraints, SQL also allows for the specification of **default** values for attributes.

- **Note** PostgreSQL does not support subqueries in CHECK constraints as shown on the previous slides. Instead, triggers (*cf.* Slide 696) need to be written.

Attribute domains

The *type* of (atomic) attributes can be declared either

- as one of the SQL built-in basic types[46], or
- as a *named domain* that has been declared before, via

CREATE DOMAIN

CREATE DOMAIN ⟨DomainName⟩ [AS] ⟨DataType⟩
[⟨DefaultClause⟩] [⟨ConstraintClause⟩]

... useful to specify, in one place, integrity constraints that apply in several places and/or to enforce name-equivalence in type checking.

Example

```
CREATE DOMAIN ExcPts AS DECIMAL(2)
CHECK (VALUE IS NOT NULL) AND (VALUE > 0)

CREATE TABLE EXERCISES ( ..., MAXPT ExcPts, ...)
```

[46] some of which can be seen in the examples above

SQL's basic data types

The standard defines a large collection of basic data types. Some of them resemble primitive types that can be found in (almost) any programming language, others are more database specific. For instance,

- Numeric data types: `integer`, `smallint`, `real`, `float`[(*n*)], `decimal`[(*n*, *k*)],
- Character data types: `char`(*n*), `varchar`(*n*), `long`,
- Byte strings: `long raw`, `bit`,
- Others: `date`, `time`, `money`,
- also: various national character sets, character sets with user-defined collation sequences,
- Since SQL-1999:
  - BLOB – binary large object,
  - CLOB – character large object,
  - BOOLEAN – with three-valued logic

Data dictionary

Apart from declaring the appropriate tables as part of the database schema, a `CREATE TABLE` statement also adds rows to various tables in the *catalog* or *data dictionary* schema that is available in every SQL database.

- These tables can be accessed (queried) like any other table using SQL as part of the schema "`INFORMATION_SCHEMA`".
- The standard defines a couple of views, *e.g.*,
  - `COLUMNS`, `REFERENTIAL_CONSTRAINTS`, `TABLES`, `USERS`, `VIEWS`, ...

  that are based upon the data in the "`DEFINITION_SCHEMA`" schema.
- The `DEFINITION_SCHEMA` itself is not visible to SQL statements.
- Each RDBMS may add its own tables/views.

Modifying existing schemas

The schema of an existing SQL table may be modified (this is often called "schema evolution" in the literature).

We need cellphone numbers of students as well.

```
ALTER TABLE Students ADD (CELLPHONE NUMERIC(12)).
```

N.B.

- Newly added attributes are "appended" at the end of existing tuples, and they are filled with NULL values.
- NOT NULL can be specified as well, but only if the table is *empty*.
- `ALTER TABLE ... MODIFY ...` allows to change the declaration of an attribute (data type or domain, NULL- or DEFAULT-clause, integrity constraints), but only if the table is empty or all rows contain NULLs in this column.
- Attributes (columns) may also be DROPped.

Dropping tables

SQL's DROP TABLE: Syntax

DROP TABLE ⟨TableName⟩

The specified table is removed from the database schema, *i.e.*,

- any rows are removed,
- the table definition is purged from the schema.

N.B. as a side-effect, the current transaction is *implicitly committed*. Hence, there is no way to UNDO this action!

11.2 DML: Data Manipulation in SQL

The **D**ata **M**anipulation (sub-) **L**anguage is used to modify the state of the database, *i.e.*, to add, modify, and remove rows to/in/from tables.
SQL contains three primitives for DML purposes:

- `INSERT`,
- `UPDATE`,
- `DELETE`.

All of them offer *some form of set-orientation*, so, not only retrieval, but also update is set-oriented in SQL databases.

INSERT: Adding rows to a table

New tuples can be inserted into a relation by either

1. explicitly specifying values for the attributes, or
2. generating the new values from a query.

Syntax

$$\texttt{INSERT INTO}\ \langle\text{TableName}\rangle\ [(\langle\text{AttrName}\rangle,\ldots)]$$
$$\left\{\begin{array}{l}\texttt{VALUES}\ \ [\text{ROW}](\langle\text{Value}\rangle,\ldots),\ldots \\ \langle\text{Query}\rangle\end{array}\right\}$$

- If no attribute list is given, values need to be provided for all attributes defined in the table's schema (and given in the order specified in the `CREATE TABLE` statement).
- With attribute names listed, `NULL`able attributes can be omitted and the order of names and values must match.
- More than one row can be inserted with `VALUES(...)`.

Examples

Insert new students (values for NULLable attribute EMAIL not known yet).

```
INSERT INTO Students(SID, FIRST, LAST)
   VALUES ROW (109,'John','Doe'),
          ROW (110,'James','Wright')
```

Populate a properly defined new table TotalResults

```
INSERT INTO TotalResults(SID, CAT, TOTALPOINTS)
       (SELECT   SID, CAT, SUM(POINTS)
        FROM     RESULTS
        GROUP BY SID, CAT)
```

N.B. obviously, the schema or attribute list of the table and the SELECT clause must match.

DELETE: Removing rows from tables

Tuples matching a given search criterion can be removed from a relation. Notice the set-oriented flavor of this update.

Syntax

```
DELETE FROM ⟨TableName⟩ [WHERE ⟨SearchCondition⟩]
```

Where ⟨SearchCondition⟩ can be an arbitrarily complex condition (like in "SELECT * FROM ⟨TableName⟩ WHERE ⟨SearchCondition⟩"). If the search condition is omitted, *all* tuples are deleted.

Remove students without any results

```
DELETE FROM Students
       WHERE SID NOT IN (SELECT SID FROM Results)
```

UPDATE: Changing attribute values in existing rows

Syntax

UPDATE ⟨TableName⟩

$$\text{SET}\left\{\begin{array}{l}\langle\text{AttrName}\rangle = \langle\text{NewValue}\rangle, \ldots\\ (\langle\text{AttrName}\rangle, \ldots) = (\langle\text{SubQuery}\rangle), \ldots\\ \text{ROW} = (\langle\text{SubQuery}\rangle)\end{array}\right\}$$

[WHERE ⟨SearchCondition⟩]

- ⟨NewValue⟩ can be an appropriately typed value expression or a subquery computing the new value,
- the SELECT clause of the ⟨SubQuery⟩ in the second form must match the attribute list on the lhs of the assignment,
- the SELECT clause of the ⟨SubQuery⟩ in the third form must match the schema of the updated table,
- the first and second forms can be mixed in one UPDATE.

Examples

Give 2 more maximum points to all SQL exercises.

```
UPDATE Exercises
   SET MAXPT = MAXPT + 2
   WHERE TOPIC = 'SQL'
```

Set maximum points of midterm exercise 1 to the maximum points achieved by any student.

```
UPDATE Exercises
   SET MAXPT = (SELECT MAX(POINTS) FROM Results
                 WHERE  CAT='M' AND ENO=1)
   WHERE CAT = 'M' AND ENO = 1
```

Set-orientation vs. state changes

While set-orientation and (read-only) queries go together well, **set-oriented updates** to the database state pose **theoretical challenges**.

- Bulk retrieval, *i.e.*, computing a whole **set** of rows as a query result, works fine, and the order of rows in the result set is immaterial.
- A (naive) implementation could *iterate* (in any order) over the input set to collect the result tuples.
- Doing the same for *set-oriented updates* may result in ill-defined semantics!
  - Different orders of iteration may yield different results!
  - None of these results might coincide with the *intended* semantics.

Example: Iteration vs. updates

Consider the following scenario:

> Students 101 and 103 jointly worked on homework assignment 1. However, they have been awarded different points. For fairness reasons, both shall be given the same points, computable as the average points of both them.

We try to solve this by the SQL UPDATE statement shown below:

“Self-referential update”

```
UPDATE Results
  SET POINTS = (SELECT AVG(POINTS) FROM Results
                WHERE  CAT='H' AND ENO=1 AND
                       (SID=101 OR SID=103))
  WHERE SID=101 OR SID=103
```

✎ Can you imagine where the problem is?

… think of a *very* simple, iterative implementation.

Possible (naive) iterative evaluation strategy

Naive, iterative evaluation (buggy!)

```
foreach r ∈ RESULTS do
    if r.SID = 101 ∨ r.SID = 103 then
        avg ← evaluate(SubQuery);          /* compute avg from DB */
        r.POINTS ← avg;                /* incrementally change rows */
    fi
od
```

Execute this algorithm! (Initially, student 101 got 10, student 103 got 5 points for homework 1.)

Iterative interpretation of set-oriented updates does not (always) work!

What (else) can we do?

- (Identify and) exclude the "problem cases" (*e.g.*, self-referential updates).
  - There are more, and interesting cases, though.
- Try a different interpretation: "Parallel" or "snapshot" semantics.

1. First, all rows are identified that need to be updated.
2. Then, all ⟨NewValue⟩s and ⟨SubQuery⟩s, *i.e.*, all right-hand sides of assignments ("source expressions") are evaluated (in the *old* database state).
3. Finally, all assignments are "executed" *in parallel*.

Notice how this resembles the idea of "atomic state transitions".

Set-orientation and updates: The SQL solution

The problem with defining clear semantics for updates in the context of set-orientation is not specific for SQL, it is a very general phenomenon.

SQL: Self-referential updates

In SQL, **right-hand sides** of assignments, as well as the set of **updated rows** are computed (conceptually)[47] **in the old database state**, before the effects of the UPDATE statement take place.

N.B. this way, a well-defined, “snapshot” semantics for set-oriented updates is guaranteed.

[47] Remember: we’re talking *semantics* here. An RDBMS is free to chose a different implementation, provided it realizes the same semantics.

11.3 Updates and Integrity Constraints

Updates to the database state are the (only) source of **potential violations of integrity constraints.** For instance,

- Insertion of new tuples can possibly
  - introduce illegal values for attributes,
  - lead to duplicate key values,
  - store “dangling” foreign key values.
- Deletions may
  - leave “widowed” foreign key references.
- Updates may introduce all these kinds of inconsistencies.

In many cases, *more than one update statement* is required to transform the current database state into a new, valid database state.
($\rightarrow$ This is part of the motivation for database *transactions*.)

Integrity constraints and database transactions

Database (ACID-) transactions are the unit of integrity preservation.[48] Hence, the DBMS is obliged to check (all relevant) integrity constraints **by the end** of each transaction.

Conceptually, each database transaction takes the form

DB-transaction

```
⟨BOT⟩                                          /* begin of transcation */
...
(sequence of SQL statements)
...
⟨EOT⟩                                          /* end of transcation */
```

N.B. SQL's ⟨EOT⟩ reads "`COMMIT WORK`", there is no explicit ⟨BOT⟩.

[48]This is what the "C"in ACID stands for: consistency.

Constraints in SQL

A database schema in SQL can contain various kinds of integrity constraints in several places.

Kinds of constraints

- Keys and candidate keys
- Foreign keys
- `NOT NULL`
- `CHECK` (⟨SearchCondition⟩)

Places

- within an attribute declaration
- within a table (or view) declaration
- within a DB schema:
  `CREATE ASSERTION` ⟨AssertionName⟩ (⟨SearchCondition⟩)

Deferred vs. immediate constraint checking

- The semantics of the transaction construct requires the DBMS to check constraints **at the end of the transaction** (“deferred constraint checking”).
- *Within* a transaction, the consistency of the database may be (temporarily) violated.
- At $\langle$EOT$\rangle$, all constraints need to be satisfied again.
- It may be quite costly, though, to defer constraint checking, since a lot of bookkeeping is required to avoid having to check **all** defined constraints.
- Often, it is much cheaper, to check constraints *during* the update or at least at the end of each update statement (“immediate constraint checking”).

Constraint checking in SQL

The SQL default is to check all constraints *immediately*, after each individual update statement. Deferred checking can be switched on or off again with an explicit statement.

Syntax

$$\texttt{SET CONSTRAINTS} \begin{Bmatrix} \texttt{ALL} \\ \langle \text{ConstraintName} \rangle, \ldots \end{Bmatrix} \begin{Bmatrix} \texttt{DEFERRED} \\ \texttt{IMMEDIATE} \end{Bmatrix}$$

Example

A debit/credit transaction transferring money from one account to another would switch off immediate checking to avoid violating the “balance” integrity constraint.

N.B. in any case, all remaining checks will be performed at ⟨EOT⟩.

What if constraints are violated?

Whenever the DBMS detects the violation of some integrity constraint by an update transaction, there are basically two options:

- either **reject** the violating transaction (“passive integrity checking”),
- or apply a **compensating update** to salvage the situation (“active integrity preservation”).

While transaction rejection (“UNDO”, rollback) can always be applied, automatic follow-up updates require additional semantic knowledge and/or predefined corrective actions.

SQL provides two ways to specify corrective actions:

- CASCADE options in the special case of foreign key constraints,
- **triggers** for the general case.

Automatic maintenance of foreign key constraints

Together with the declaration of foreign keys, SQL allows for the specification of corrective actions in case of integrity violations.

SQL syntax for foreign key maintenance

… REFERENCES ⟨TableName⟩ [(⟨AttrName⟩, . . .)]

$$\left[\ \texttt{ON}\ \left\{\begin{array}{l}\texttt{UPDATE}\\ \texttt{DELETE}\end{array}\right\}\ \left\{\begin{array}{l}\texttt{CASCADE}\\ \texttt{SET}\ \left\{\begin{array}{l}\texttt{NULL}\\ \texttt{DEFAULT}\end{array}\right\}\\ \texttt{NO ACTION}\\ \texttt{RESTRICT}\end{array}\right\}\ \right]$$

- ON UPDATE CASCADE propagates changes on primary key values to referencing foreign keys.
- ON DELETE CASCADE deletes referencing tuples, if the referenced tuple is deleted.
- SET NULL changes the referencing foreign key value to NULL (if permitted).
- NO ACTION is the default (passive checking) mode.
- RESTRICT passively checks *before* trying the update (consistency cannot be reestablished by triggers).

Example

In the homework database, we may want to specify that RESULTS be removed, once we delete STUDENTS.

```
CREATE TABLE RESULTS ( SID DECIMAL(3) REFERENCES STUDENTS
                                      ON DELETE CASCADE
                                      ON UPDATE CASCADE,
                       ... ) ...
```

Now,

```
DELETE FROM Students WHERE SID=104
```

will not only remove the STUDENTS row, but also all referencing RESULTS.

By specifying ON UPDATE CASCADE as well, modification of SID values in the STUDENTS table will propagate to RESULTS.

Triggers

Many database systems offer a **trigger** mechanism, that extends the DBMS by some kind of **active rules**.

Event-Condition-Action (ECA) rules

... take the general form

$$\begin{array}{l}\text{ON } \langle\text{event}\rangle \\ \text{IF } \langle\text{condition}\rangle \\ \text{DO } \langle\text{action}\rangle\end{array}$$

Depending on the system capabilities and the ECA-language provided, this is an extremely powerful (often even *too* powerful) feature.

ECA-rules

In general,

- a triggering ⟨event⟩ can be (almost) everything, *e.g.*, specific updates occuring on some table(s)/row(s), system events (clock ticks, system startup/shutdown), …
- the ⟨condition⟩ can check a complex search condition, possibly refering to values of the triggering event/row(s),
- the ⟨action⟩ is given, *e.g.*, as a (sequence of) update statement(s) and/or other action items.
- In particular, an ⟨action⟩ can trigger one or more other ECA rules.

The interaction, timing, transactional coordination, confluence, or just the termination of several ECA rules is a very challenging research question in itself.

Triggers in SQL

In SQL, triggering events can be insertions, deletions and updates.

Syntax SQL trigger declaration

$$\texttt{CREATE TRIGGER } \langle\text{TriggerName}\rangle$$
$$\quad\left\{\begin{array}{l}\texttt{BEFORE}\\ \texttt{AFTER}\end{array}\right\}\left\{\begin{array}{l}\texttt{INSERT}\\ \texttt{DELETE}\\ \texttt{UPDATE [OF } \langle\text{AttrName}\rangle \ldots\texttt{]}\end{array}\right\}\texttt{ ON } \langle\text{TableName}\rangle$$
$$\quad\texttt{[REFERENCING}\left\{\begin{array}{l}\texttt{OLD}\\ \texttt{NEW}\end{array}\right\}\left\{\begin{array}{l}\texttt{ROW}\\ \texttt{TABLE}\end{array}\right\}\texttt{ AS } \langle\text{AliasName}\rangle \ldots\texttt{]}$$
$$\quad\langle\text{TriggeredAction}\rangle$$

- The trigger can be “fired” BEFORE or AFTER the triggering update.
- REFERENCING allows for the introduction of tuple and table variables for old and new values.

Triggers in SQL

⟨TriggeredAction⟩ syntax:

```
[ FOR EACH { ROW | STATEMENT } ]
[ WHEN (⟨SearchCondition⟩) ]
⟨TriggeredSQLStatement⟩
```

Where:

- ⟨TriggeredSQLStatement⟩ is a “SQL procedure statement”, *i.e.*, a single update statement or a complex program, written in SQL’s procedural programming language (see later).
- The WHEN clause is the optional condition (the “C” in ECA).
- The FOR EACH construct allows the specification of an action either once for the triggered event (STATEMENT) or once for each affected tuple (ROW).

Example

Make sure that a new employee’s salary (plus 20% overhead) is covered by the department’s budget.

```
CREATE TRIGGER AddNewEmpsSalaryToDeptBudget
   AFTER INSERT ON Employees
   REFERENCING NEW ROW AS E
   FOR EACH ROW
   UPDATE Departments D WHERE D.dno=E.dno
      SET D.budget = D.budget + E.salary * 1.2
```

Here, `FOR EACH ROW` makes sure that the departments’ budgets get updated correctly, if multiple rows are inserted with a single `INSERT` statement.

Triggers in PostgreSQL

Many database systems introduce their own syntax to declare triggers. In PostgreSQL, triggers are a special case of user-defined functions.

CHECK constraint from Slide 670

```
CREATE TABLE RESULTS (
   ...
   CHECK   (POINTS <= (SELECT MAXPT FROM EXERCISES E
                       WHERE  E.ENO=RESULTS.ENO AND E.CAT=RESULTS.CAT))
)
```

In the following, we show the PostgreSQL syntax that checks whether the value given for column POINTS is valid.

Triggers in PostgreSQL

Example

```
CREATE OR REPLACE FUNCTION check_result_points()
RETURNS TRIGGER AS
$$
BEGIN
   IF NEW.POINTS > (SELECT MAXPT FROM EXERCISES E
                    WHERE E.ENO=RESULTS.ENO AND E.CAT=RESULTS.CAT)
   THEN
      RAISE EXCEPTION ’% is larger than maximum points’, NEW.POINTS;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE PLpgSQL;

CREATE TRIGGER check_result_points BEFORE INSERT ON RESULTS
FOR EACH ROW EXECUTE PROCEDURE check_result_points();
```

11.4 Views and View Updates

We have mentioned **views** before, in passing …

What is a view, after all?

- A view is a *derived/computed* relation/table.
- Defining a view means to register a query under a given name in the schema.
- To a *query*, a view looks exactly like a stored (“base”) table.
- The contents of a view is computed anew upon each (read) access (“macro expansion”).
- Updates to base tables automatically propagate to the view.
- *Updates to the view* automatically propagate to the base tables, **if possible**. There are several **restrictions**, though!

The role of views

Views can serve several purposes, *e.g.*

- Giving a name to a (complex sub-) query can be utilized to *simplify* the formulation of complex *retrieval tasks*.
- As a means to *realize the external level* (subschemas) of the ANSI-3-Schema-Architecture, they can be used to tailor, to restrict, to restructure the logical schema for a particular class of applications or users.
- Views can serve a stable point of reference in the presence of *schema evolution*.
- Views can be used to *hide* unnecessary *data* (schema simplification) or sensitive data (privacy) from certain applications.

View definition and use in SQL

Syntax

```
CREATE VIEW ⟨ViewName⟩ [ (⟨ColumnList⟩) ]
  AS ⟨Query⟩ [ WITH CHECK OPTION ]
```

Define a view of those students, who have not submitted homeworks yet.

```
CREATE VIEW LazyStudents AS
   SELECT *
   FROM   Students
   WHERE  sid NOT IN (SELECT sid FROM Results WHERE cat='H')
```

Once a view is defined, its name can be used anywhere in a SQL query, where a relation name is required (*e.g.*, in the FROM clause).

Example

```
SELECT DISTINCT first, last FROM LazyStudents
```

View substitution

Processing a query on a view is—in principle—quite easy: simply substitute the view definition for the view name in the query using the view.

Example

```
SELECT DISTINCT first, last FROM LazyStudents
       ↓
SELECT DISTINCT first, last FROM
   (SELECT *
    FROM   Students
    WHERE  sid NOT IN (SELECT sid FROM Results WHERE cat='H')
   )
```

N.B. If SQL were *really, fully* orthogonal, it could be as simple as that. Since there are certain limitations, however, (also because nesting in the FROM clause was not allowed before SQL:1992) the actual algorithm is somewhat more involved.

View updates

Applying database updates *through views* is far from trivial!

Ambiguity

Consider a view definition with a UNION operator:

```
    CREATE VIEW U AS SELECT * FROM R UNION SELECT * FROM S.
```

If you INSERT new tuples into this view U, what should the DBMS do?

- Insert the new row into table R?
- Insert the new row into table S?
- Insert the new row into both tables?

There is no obvious answer!

Some (few) view updates can, others (many) can not be translated uniquely into updates to base tables.

Criteria for view updates

When translating view updates to base tables, you will want

- Conformity: the effect of an update to a view shall be the same as if the view were a stored base table.
- Minimality: a minimal set of updates to the base tables shall be generated that guarantees the effect (*cf.* the `INSERT` into the `UNION` view: do not insert into *both* base tables).
- Consistency preservation: updates to a view must not violate integrity constraints on base tables.
- Privacy: if the view was introduced to hide sensitive data, this data must not be affected by the update.
- Uniqueness: the translation must not be ambiguous or non-deterministic.

Some view update problems (1)

Projection view

View definition:

```
CREATE VIEW Mailinglist AS
  SELECT DISTINCT first, last, email FROM Students.
```

Update:

```
INSERT INTO Mailinglist VALUES ('John','Doe','jd@foo.edu').
```

Problem: No value can possibly be given for non-projected attributes.

- If these attributes are NULLable, or if a DEFAULT value is declared, can use these values.
- If (parts of) primary key is not projected: possibly duplicates have been removed, so 1 view row might "represent" multiple base table rows!

Consequence: Keep keys, no DISTINCT.

Some view update problems (2)

Selection view

View definition:

```
CREATE VIEW Mailinglist AS
  SELECT * FROM Students WHERE email IS NOT NULL.
```

Update:

```
UPDATE Mailinglist WHERE last='Smith' SET email=NULL.
```

Problems:

1. Effect of this UPDATE looks like a DELETE to the view user!
   The modified row was "migrated" as a result of the update.
2. If the translation is

   ```
   UPDATE Students WHERE last='Smith' SET email=NULL,
   ```

   other rows, not part of the view, could also be updated!
3. What about

   ```
   INSERT INTO Mailinglist VALUES (110,'Jim','Jones',NULL)?
   ```

Updates to selection views

1. The first problem (tuple migration) can be avoided by using an option in the SQL view definition.

   View definition: `WITH CHECK OPTION`

   ```
   CREATE VIEW Mailinglist AS
       SELECT * FROM Students WHERE email IS NOT NULL
       WITH CHECK OPTION
   ```

   Now, any update to a view-tuple that would let the tuple "disappear" is not allowed. Similarly, no INSERTs into the view are possible for tuples violating the view's selection condition.

2. The second problem is avoided by adding the view's selection condition to the translation of the update.

   Translation of `UPDATE ... WHERE`

   ```
   UPDATE Students WHERE last='Smith' AND email IS NOT NULL
       SET email=NULL
   ```

Some view update problems (3)

Join view

View definition:

```
CREATE VIEW ExRes AS
  SELECT sid, eno, cat, topic, points
  FROM Exercises NATURAL JOIN Results.
```

Updates

1. `INSERT INTO ExRes VALUES (104,1,’H’,’Calculus’,9)`
2. `DELETE FROM ExRes WHERE cat=’H’ AND sid=102 AND eno=1`

What is the correct translation?

1. Shall the existing EXERCISES row be modified, replaced, the update rejected?
2. How to delete the row from the join result? Delete a row from RESULTS, delete a row from STUDENTS, change a foreign key?

Some view update problems (4)

- Set operations pose problems similar to join views: unambiguous, minimal translation is hardly ever possible.
- Aggregation functions in view definitions can not be updated as well.

Aggregation view

```
CREATE VIEW TotalPointsPerStudents AS
    SELECT sid, cat, SUM(points) FROM Results
    GROUP BY sid, cat
```

There is no way to translate an UPDATE or INSERT into this view.

View update problems

As we have seen from various examples, there are at least the following problem areas with view updates:

- Effects on data not part of the view must be excluded.
- Minimality and uniqueness of the update translation should be guaranteed.
- A one-to-one correspondence between view tuples and base table tuples is needed (do not project out key attributes, no aggregation, no grouping, no duplicate elimination).
- Hidden information (privacy) shall not be revealed.

A lot of research has been devoted to the question which views can be updated with what update statements.

SQL takes a rather pragmatic, and rather restrictive, approach.

Updatable views in SQL

In SQL, a view is updatable, iff all of the following holds:

1. No join and no set operations are contained in the view definition.
2. No `DISTINCT` clause.
3. No arithmetics and no aggregation in the `SELECT` clause.
4. Exactly one table reference in the `FROM` clause.
5. No nested subqueries (in `WHERE`) with self-references (the table in the `FROM` clause must not appear in nested subqueries).
6. No `GROUP BY`.

Materialized views

Views, as described above, are a tool for the *external database schema*.

- They are redundant, derived tables. Conceptually, at least, their content is computed everytime their name is accessed in a query.

ℹ Materialized views

Many DBMSs nowadays offer the option to **materialize** a view.

- The redundant *content* of the view is stored, not only the base tables.
- This can improve the performance of complex, frequently accessed views.
- The view is computed once. Without intervening updates, the next read access can readily read the view result without a need for recomputation.
- Upon updates to the base tables, though, additional effort is necessary.
  - Drop the materialized view and recompute upon the next access.
  - Try to *maintain* the materialization.

This is particularly popular in (→) OLAP applications.

11.5 Access Control

Since a database keeps data of *all* parts of an application context, those data are typically subject to some **access restrictions**, not everyone (or each part of the application) is supposed to/allowed to work with all the data.

This might be

- for legal reasons (privacy of personal data)
  See, for instance, the German “Bundesdatenschutzgesetz”.
- the result of some enterprise policy
  Such as, keeping confidential data away from other divisions.
- a matter of shielding applications form the complexity of the schema
  To ease application development and to limit the scope of malfunctions.

Component questions

In order to exercise some kind of access control, a system needs to be able to

- Identify subjects (persons, groups, programs) that interact with it.

  Who is it?

- Reliably authenticate those subjects.

  Proof who you are!

- Identify objects to be worked on.

  Where do you want to go today?

- Distinguish operations to be carried out on those objects.

  What do you want to do?

Identification and authentication

… is an important issue that is beyond the scope of this course, though.

We (and SQL) assume that a reliable and secure identification and authentication of **database users** (possibly very distinct from users of the underlying OS) is supported. The bare minimum is a UserID and Password scheme.

SQL CONNECT with userID and password

```
CONNECT TO ⟨database⟩ AS ⟨user⟩ IDENTIFIED BY ⟨password⟩
```

[As an aside: User authentication]

In general, authentication can be based on

- **Knowledge.**
  - Identificators: passwords, PINs, signatures
  - Credentials: knowledge
  - Disadvantages: can be passed to somebody else (knowingly or not).
- **Possession.**
  - Identificators: passports
  - Credentials: certificates, keys, cards
  - Disadvantages: possible theft, can be passed to somebody else
- **Biometrics.**
  - Identificators: finger prints, retina scan, speech, typing rhythm
  - Disadvantages: can not be “retracted”

Or combinations thereof.

Access control matrix

A widely used mechanism for access control is to maintain, in some form, an **access control matrix** that implements a function

$$\text{subject} \times \text{object} \rightarrow \text{permission}$$

where

- **subject** is a person, group, or role, or a process initiated by one of those
- **object** depends on context
  in SQL, *e.g.*, maybe a table, row, attribute, ...
- **permission** depends on context
  in SQL, *e.g.*, SELECT, INSERT, UPDATE, DELETE, ...

In general, such a matrix can get quite huge. Typically, it is stored in some partitioning scheme.

SQL privileges

Since SQL offers quite subtle data access operations, its access control is based on a diverse set of permissions ("privileges" in SQL speak), *e.g.*

- `SELECT` [(⟨column-or-method-list⟩)]
- `DELETE`
- `INSERT` [(⟨column-list⟩)]
- `UPDATE` [(⟨column-list⟩)]
- `REFERENCES` [(⟨column-list⟩)]
- `USAGE`
- `TRIGGER`
- `UNDER`
- `EXECUTE`
- `ALL PRIVILEGES`

Each of these can be granted on a table, a view, a domain, ...

SQL's GRANT and REVOKE statements

Syntax

$$\left\{\begin{array}{l}\texttt{GRANT}\\ \texttt{REVOKE}\end{array}\right\}\left\{\begin{array}{l}\texttt{ALL PRIVILEGES}\\ \langle\text{privilege}\rangle[,\ldots]\end{array}\right\}\texttt{ON}\left\{\begin{array}{l}{[}\,\texttt{TABLE}\,{]}\ \langle\text{relview}\rangle\\ \texttt{DOMAIN}\ \langle\text{domain}\rangle\\ \ldots\end{array}\right\}$$

$$\left\{\begin{array}{l}\texttt{TO}\\ \texttt{FROM}\end{array}\right\}\left\{\begin{array}{l}\texttt{PUBLIC}\\ \langle\text{user}\rangle[,\ldots]\end{array}\right\}\left\{\begin{array}{l}{[}\ \texttt{WITH GRANT OPTION}\ {]}\\ {[}\ \texttt{CASCADE}\ {]}\end{array}\right\}$$

Remarks

- Initially, only the owner of an object has permissions.
- The owner can `grant` permissions to others.
- Value-dependent permissions can be specified by granting access to (selection) views.
- The `WITH GRANT OPTION` (only for `GRANT`, not for `REVOKE`) allows the grantor to specify that the grantee can delegate the granted permission to a third party.

Delegation of permissions and cascading `revoke`

If the grantee *B* received certain privileges `WITH GRANT OPTION` from a grantor *A*, those privileges may be delegated to somebody else, *e.g.*, *C*.

- When the original grantor *A* eventually `REVOKE`s the privilege from *B*, there is a choice for *A*:
  1. Use a "simple" `REVOKE`, or
  2. a cascading `REVOKE … CASCADE`.
- In the first case, the grantee *B* loses the permission, but the third party *C* retains it.
- In the second case, the grantee *B* and the third party *C* lose the permission.
- **Exception:** the third party *C* has also obtained the privilege from somebody else (*D*, say). In that case, *C* retains the privilege.

Challenge: cascading `revoke`

✎ **Outsmart SQL's access control!**

Suppose you're grantee *B* and you want to make sure you will never lose that particularly interesting SELECT privilege you just received from your boss *A*. There is this pal of your's, *C*. Can you protect yourself from losing that privilege, even if your boss *A* revokes it with the CASCADE option ...?

Unfortunately, you're not the first one who came up with this idea ...!

Keeping track of privilege delegations

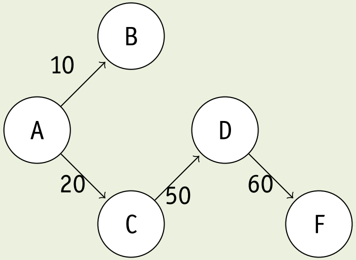
One way of keeping track is by using a directed graph, whose

nodes	represent subjects who obtained a certain privilege,
edges	represent granting of privileges,
labels	on the edges record a time stamp for the GRANT command.

In case of a privilege revocation with the CASCADE option, the digraph and its edge labels are examined to determine, if a privilege has potentially been obtained "along a particular path".

Example: Cascading revoke (1)

Resulting situation



User *B* revokes privilege from *D* with CASCADE.

Resulting situation should look as if *D* never received the privilege from *B*.

- *E*, *G* could only have received the privilege “via *B*”,
- whereas *F* could also have received it “via *C*”.

Therefore, the resulting situation should be … (see above).

Example: Cascading revoke (2)

With different time stamps in the initial situation, we obtain...

B: REVOKE...FROM *D* CASCADE

$$\begin{tikzcd}
 & B \arrow[dr, "30"] & & E \arrow[r, "60"] & G & & & B & \\
A \arrow[ur, "10"] \arrow[dr, "20"'] & & D \arrow[ur, "40"] \arrow[dr, "50"] & & & \longrightarrow & A \arrow[ur, "10"] \arrow[dr, "20"'] & & D \\
 & C \arrow[ur, "70"'] & & F & & & & C \arrow[ur, "70"'] &
\end{tikzcd}$$

Example: Non-cascading revoke

Revoking without CASCADE requires adjustments

$$\begin{tikzcd}
 & B \arrow[dr, "30"] & & E \arrow[r, "60"] & G \\
A \arrow[ur, "10"] \arrow[dr, "20"'] & & D \arrow[ur, "40"] \arrow[dr, "70"] & & \\
 & C \arrow[ur, "50"'] & & F &
\end{tikzcd}$$

B: REVOKE...FROM *D* $\longrightarrow$

$$\begin{tikzcd}
 & B \arrow[rr, "40"] \arrow[drr, "70"] & & E \arrow[r, "60"] & G \\
A \arrow[ur, "10"] \arrow[dr, "20"'] & & D \arrow[r, "70"'] & F & \\
 & C \arrow[ur, "50"'] & & &
\end{tikzcd}$$

Outsmarting will no longer work!

You (B) and your pal (C) ...

$$\begin{tikzcd}
A \arrow[r, "t_1"] & B \arrow[r, bend left, "t_2 > t_1"] & C \arrow[l, bend left, "t_3 > t_2"]
\end{tikzcd}$$

Principal problems

- Depends heavily on reliable and secure user identification and authentication.
- Privileges (objects implementing them) are extremely sensitive
  $\Rightarrow$ may need even stronger protection.
- Privileges protect *data objects*, not the *information contained* within.
  $\Rightarrow$ implicit transfer (and even extension!) of permissions is possible!

DAC dilemma

— User A has SELECT permissions on object O,

— user B has no permissions on O.

— User A has no permission to grant access on O to B.

— Yet, A can give read access to O's content to B:

1. A creates a new object O', copies O's content into O'.
2. A — as the owner of object O' — grants read access on O' to B.

Problem source: *discretionary access control*. It's all up to the user...