Part 6: Relational Algebra

References:

- Elmasri/Navathe:Fundamentals of Database Systems, 3rd Edition, 1999.
  Section 7.4 "Basic Relational Algebra Operations",
  Section 7.5 "Additional Relational Algebra Operations",
  Section 7.6 "Examples of Queries in Relational Algebra"
- Kemper/Eickler: Datenbanksysteme (in German), 4th Edition, 2001.
  Section 3.4, "Die relationale Algebra" ("The Relational Algebra")
- Silberschatz/Korth/Sudarshan: Database System Concepts, Third Edition, 1999.
  Section 3.2: "The Relational Algebra"
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Codd: A relational model of data for large shared data banks. Communications of the ACM, 13(6), 377–387, 1970. Reprinted in CACM 26(1), 64–69, 1983.
  See also: [http://www1.acm.org:81/classics/nov95/toc.html] (incomplete)

Objectives

After completing this chapter, you should be able to:

- enumerate and explain the operations of relational algebra.

  Especially, you should know the five basic operations.

- write relational algebra queries of the type "join-select-project".

  Plus simple queries involving set difference and union.

- discuss correctness and equivalence of given relational algebra queries.

Overview

1. Introduction, Selection, Projection
2. Cartesian Product, Join
3. Set Operations
4. Outer Join
5. Formal Definitions, A Bit of Theory

Example Database (1)

STUDENTS

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

EXERCISES

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

RESULTS

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

Example Database (2)

- `STUDENTS`: one row for each student in the course.
  - ◇ `SID`: “Student ID” (primary key).
  - ◇ `FIRST`, `LAST`: First and last name.
  - ◇ `EMAIL`: Email address (can be null).
- `EXERCISES`: one row for each graded exercise.
  - ◇ `CAT`: Exercise category (key together with `ENO`).

    E.g. `'H'`: homework, `'M'`: midterm exam, `'F'`: final exam.
  - ◇ `ENO`: Exercise number (within category).
  - ◇ `TOPIC`: Topic of the exercise.
  - ◇ `MAXPT`: Max. no. of points (How many points is it worth?).

Example Database (3)

- `RESULTS`: one row for each submitted solution to an exercise.
  - ◇ `SID`: Student who wrote the solution.

    This is a foreign key referencing `STUDENTS`.
  - ◇ `CAT`, `ENO`: Identification of the exercise.

    This is a foreign key referencing `EXERCISES`.
    Together with `SID` it forms the primary key of the table.
  - ◇ `POINTS`: Number of points the student got for the solution.
  - ◇ A missing row means that the student did not yet hand in a solution to the exercise.

Relational Algebra (1)

- Relational algebra (RA) is a theoretical query language for the relational model.
- Relational algebra is not used in any commerical system on the user interface level.
- However, variants of it are used to represent queries internally (for query optimization and execution).
- Knowledge of relational algebra will help in understanding SQL and relational database systems.

  E.g. one talks about "joins" (a relational algebra operation) even when discussing SQL queries. Explicit joins were added in SQL-92.

Relational Algebra (2)

- An algebra is a set together with operations on this set.
- For instance, the set of integers together with the operations $+$ and $*$ forms an algebra.
- In the case of relational algebra, the set is the set of all finite relations.
- One operation of relational algebra is $\cup$ (union). This is natural since relations are sets.

Relational Algebra (3)

- Another operation of relational algebra is selection.

  In contrast to operations like + for integers, the selection σ is parameterized by a simple condition.

- E.g. $\sigma_{\mathtt{SID=101}}$ selects all tuples in the input relation that have the value "101" in column "SID":

$\sigma_{\mathtt{SID=101}}$ (

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

) =

SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12

Relational Algebra (4)

- Since the output of a relational algebra operation is again a relation, it can be input for another relational algebra operation.

  And so on, until the query result is computed (again a relation). The relational algebra is so simple because the relational model has only a single construct: The relation.

- A query is then a term/expression in this algebra.
- Arithmetic expressions like $(x + 2) * y$ are familiar.
- In relational algebra, relations are connected:

$$\pi_{\texttt{FIRST},\,\texttt{LAST}}(\texttt{STUDENTS} \bowtie \sigma_{\texttt{CAT}=\texttt{'M'}}(\texttt{RESULTS})).$$

Relational Algebra (5)

Minor Data Model Differences to SQL:

- Null values are usually excluded in the definition of relational algebra, except when operations like the outer join are defined (last section of this chapter).

  Even for the outer join, the null value is treated simply like an additional value added to every data type. Using a three-valued logic as in SQL would make the definitions significantly more complicated.

- Relational algebra treats relations as sets, i.e. any duplicate tuples are automatically eliminated.

  In SQL, relations are multisets and can contain duplicates. If necessary, one has to request duplicate elimination explicitly ("`DISTINCT`"). In relational algebra, one does not have to think about duplicates.

Relational Algebra (6)

Importance of Relational Algebra for DB Theory:

- Relational algebra is much simpler than SQL, it has only five basic operations and can be completely defined on one page.
- Relational algebra is also a yardstick for measuring the expressiveness of query languages.
- E.g., every query that can be formulated in relational algebra can also be formulated in SQL.

  I.e. SQL is at least as powerful as relational algebra. Vice versa, every SQL query (without null values, aggregations, and duplicates) can also be written in relational algebra. See also Slide 6-115.

Selection (1)

- The operation σ_φ selects a subset of the tuples of a relation, namely those which satisfy the condition φ. Selection acts like a filter on the input set.

  σ is the greek letter sigma, φ is the greek letter phi.
  All textbooks use σ for selection, but φ is not standard.
  In ASCII, write e.g. `SELECT[condition](Relation)`.

- Example:

$$\sigma_{A=1}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 1 & 4 \\ 2 & 5 \\ \hline\end{array}\right) = \begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 1 & 4 \\ \hline\end{array}$$

Selection (2)

- The selection condition has the following form:

  ⟨Term⟩ ⟨Comparison-Operator⟩ ⟨Term⟩

- The selection condition returns a boolean value (true or false) for a given input tuple.

- ⟨Term⟩ (or "expression") is something that can be evaluated to a data type element for a given tuple:
  - ◇ an attribute name,
  - ◇ a data type constant, or
  - ◇ an expression composed from attributes and constants by data type operations like $+$, $-$, $*$, $/$.

Selection (3)

- ⟨Comparison-Operator⟩ is
  - $=$ (equals), $\neq$ (not equals),

    One can also write "`<>`" instead of $\neq$.
  - $<$ (less than), $>$ (greater than), $\leq$, $\geq$,

    One can also write `<=` instead of $\leq$ and `>=` instead of $\geq$.
  - or other data type predicates (e.g. `LIKE`).
- Examples for Conditions:
  - `LAST = 'Smith'`
  - `POINTS >= 8`
  - `POINTS = MAXPT` (the input relation must have both attributes).

Selection (4)

- $\sigma_\varphi(R)$ can be implemented as:

```
Create new temporary relation $T$;
**foreach** tuple $t$ **from** input relation $R$ **do**
    Evaluate condition $\varphi$ for tuple $t$;
    **if** true **then**
        **insert** $t$ **into** $T$;
    **fi**
**od**;
**return** $T$;
```

- With other data structures (e.g. a B-tree index), it might be possible to compute $\sigma_\varphi(R)$ without reading each tuple of the input relation.

Selection (5)

- Of course, the attributes used in the selection condition must appear in the input table:

$$\sigma_{C=1}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 2 & 4 \\ \hline\end{array}\right) = \textbf{Error}$$

- The following is legal, but the selection is superfluous, because the condition is always true:

$$\sigma_{A=A}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 2 & 4 \\ \hline\end{array}\right) = \begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 2 & 4 \\ \hline\end{array}$$

Selection (6)

- It is no error if the result of a relational algebra expression happens to be empty in a specific state:

$$\sigma_{A=3}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 2 & 4 \\ \hline\end{array}\right) = \emptyset$$

- It is legal, but most probably an error, to use a condition that is always false (inconsistent):

$$\sigma_{1=2}\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 2 & 4 \\ \hline\end{array}\right) = \emptyset$$

Selection (7)

- $\sigma_\varphi(R)$ corresponds to the following SQL query:

```
SELECT *
FROM   R
WHERE  φ
```

- I.e. selection corresponds to the `WHERE`-clause.
- A different relational algebra operation called “projection” corresponds to the `SELECT`-clause in SQL. This can be slightly confusing.

Extended Selection (1)

- In the basic selection operation, only simple conditions consisting of a single comparison ("atomic formula") are possible.
- However, one can extend the possible conditions by permitting to combine the single conditions by the logical operators $\wedge$ (and), $\vee$ (or), and $\neg$ (not):

φ_1	φ_2	$\varphi_1 \wedge \varphi_2$	$\varphi_1 \vee \varphi_2$	$\neg\varphi_1$
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Extended Selection (2)

- $\varphi_1 \wedge \varphi_2$ is called the "conjunction of φ_1 and φ_2"
- $\varphi_1 \vee \varphi_2$ is called the "disjunction of φ_1 and φ_2"
- $\neg\varphi_1$ is called the "negation of φ_1".
- One can write "`and`", "`or`" and "`not`" instead of the symbols "$\wedge$", "$\vee$", "$\neg$" used in mathematical logic.

  "$\wedge$" is similar to the intersection symbol "$\cap$", and indeed the tuples satisfying the conjunction "$\wedge$" are the intersection of the tuples that satisfy the two subconditions. In the same way is "$\vee$" similar to the "$\cup$" (set union) symbol.

Extended Selection (3)

- The selection condition must permit evaluation for each input tuple in isolation.

  Thus, "exists" ($\exists$) and "for all" ($\forall$) and nested relational algebra queries are not permitted in selection conditions.

- This extended form of selection is not necessary, since it can always be expressed with the basic operations of relational algebra. But it is convenient.

- E.g. $\sigma_{\varphi_1 \wedge \varphi_2}(R)$ is equivalent to $\sigma_{\varphi_1}(\sigma_{\varphi_2}(R))$.

  $\vee$ and $\neg$ need $\cup$ (union) and $-$ (set difference), which are also basic operations of relational algebra (see below): $\sigma_{\varphi_1 \vee \varphi_2}(R)$ is equivalent to $\sigma_{\varphi_1}(R) \cup \sigma_{\varphi_2}(R)$ and $\sigma_{\neg\varphi}(R)$ is equivalent to $R - \sigma_{\varphi}(R)$.

Exercise

Write the following queries in relational algebra:

- Which exercises are about "SQL"?

  Print the entire row of the table. Eiminating columns is treated below.

- List all entries for Homework 1 (requires `CAT='H'`) in the table `RESULTS` that have less than 10 points.

This refers to the schema on Slide 6-4:

- `STUDENTS(`<u>`SID`</u>`, FIRST, LAST, EMAIL`^{o}`)`
- `EXERCISES(`<u>`CAT`</u>`, `<u>`ENO`</u>`, TOPIC, MAXPT)`
- `RESULTS(`<u>`SID`</u>`→STUDENTS, (`<u>`CAT`</u>`, `<u>`ENO`</u>`)→EXERCISES, POINTS)`

Projection (1)

- The projection π eliminates attributes (columns) from the input relation.

  π is the greek letter "pi".
  In databases it always means "projection", and not 3.14. . . .
  Some authors use a capital Π for distinction.
  To use ASCII characters, write `PROJECT[Columns](Relation)`.

- Example:

$$\pi_{A,C}\left(\begin{array}{|c|c|c|}\hline A & B & C \\ \hline 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \\ \hline\end{array}\right) = \begin{array}{|c|c|}\hline A & C \\ \hline 1 & 7 \\ 2 & 8 \\ 3 & 9 \\ \hline\end{array}$$

Projection (2)

- In general, the projection $\pi_{A_{i_1},\ldots,A_{i_k}}(R)$ produces for each input tuple $(A_1\colon d_1, \ldots, A_n\colon d_n)$ an output tuple $(A_{i_1}\colon d_{i_1}, \ldots, A_{i_k}\colon d_{i_k})$.

  While σ selects certain rows from the input relation, and discards the others, π selects certain columns, and discards the others.

- I.e. the attribute values are not changed, but only the explicitly mentioned attributes are retained. All other attributes are “projected away”.

  Note: “to project a column away” is database slang. Normally, things are projected onto or into something else.

Projection (3)

- Normally, there is one output tuple for every input tuple. However, if two input tuples lead to the same output tuple, the duplicate will be eliminated.

  DBMS use an explicit duplicate elimination when needed. But in theory, relations are sets.

- Example:

$$\pi_B\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 4 \\ 2 & 5 \\ 3 & 4 \\ \hline\end{array}\right) = \begin{array}{|c|}\hline B \\ \hline 4 \\ 5 \\ \hline\end{array}$$

Projection (4)

- $\pi_{A_{i_1},\ldots,A_{i_k}}(R)$ can be implemented as follows:

```
Create new temporary relation $T$;
foreach $t = (A_1\colon d_1, \ldots, A_n\colon d_n)$ in $R$ do
    Compute $u = (A_{i_1}\colon d_{i_1}, \ldots, A_{i_k}\colon d_{i_k})$;
    insert $u$ into $T$;
od;
return $T$;
```

- This program fragment assumes that “**insert**” does the duplicate elimination which might be necessary.

Projection (5)

- The projection can be more general:
  - ◇ Attributes can be renamed: $\pi_{B_1 \leftarrow A_{i_1}, \ldots, B_k \leftarrow A_{i_k}}(R)$ transforms the input tuple $(A_1\colon d_1, \ldots, A_n\colon d_n)$ into the output tuple $(B_1\colon d_{i_1}, \ldots, B_k\colon d_{i_k})$.
  - ◇ Return values can be computed by datatype operations such as $+$ or `||` (string concatenation):

$$\pi_{\texttt{SID},\ \texttt{NAME} \leftarrow \texttt{FIRST || ' ' || LAST}}(\texttt{STUDENTS}).$$

  - ◇ Columns can be created with constant values:

$$\pi_{\texttt{SID}, \texttt{FIRST}, \texttt{LAST}, \texttt{GRADE} \leftarrow \texttt{'A'}}(\texttt{STUDENTS}).$$

Projection (6)

- The projection is a mapping, which is applied to every input tuple.
- Each input tuple is mapped locally to an output tuple. Only functions which are defined based on single input tuples are allowed.

  Values from different input tuples cannot be combined into one output tuple (but see the cartesian product below). Otherwise, quite general tuple-to-tuple mappings are possible.

Projection (7)

- $\pi_{A_1,\ldots,A_n}(R)$ corresponds to the SQL query:

```
SELECT DISTINCT A1, ..., An
FROM   R
```

- The keyword `DISTINCT` is not always necessary.

  The query will run faster without it. `DISTINCT` is unnecessary when $A_1, \ldots, A_n$ contain a key. Sometimes one also wants duplicates.

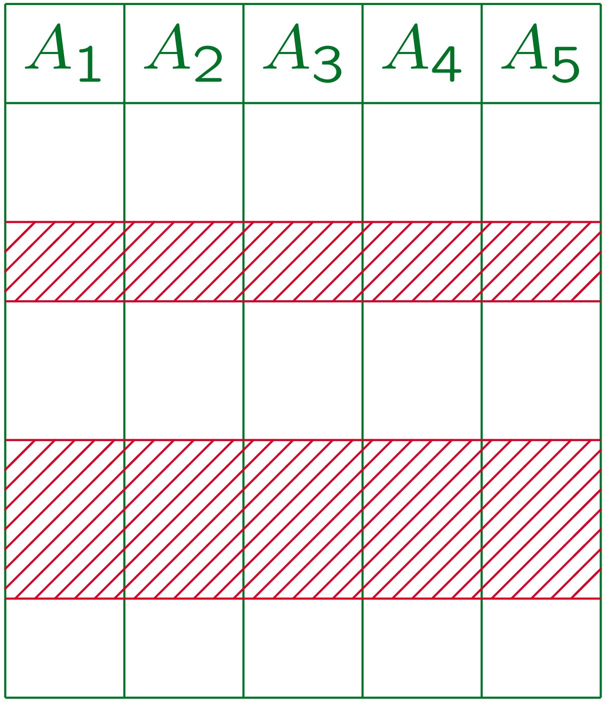
- $\pi_{B_1 \leftarrow A_1,\ldots,B_n \leftarrow A_n}(R)$ is written in SQL as follows:

```
SELECT DISTINCT A1 AS B1, ..., An AS Bn
FROM   R
```

- The keyword `AS` can be left out ("syntactic sugar").

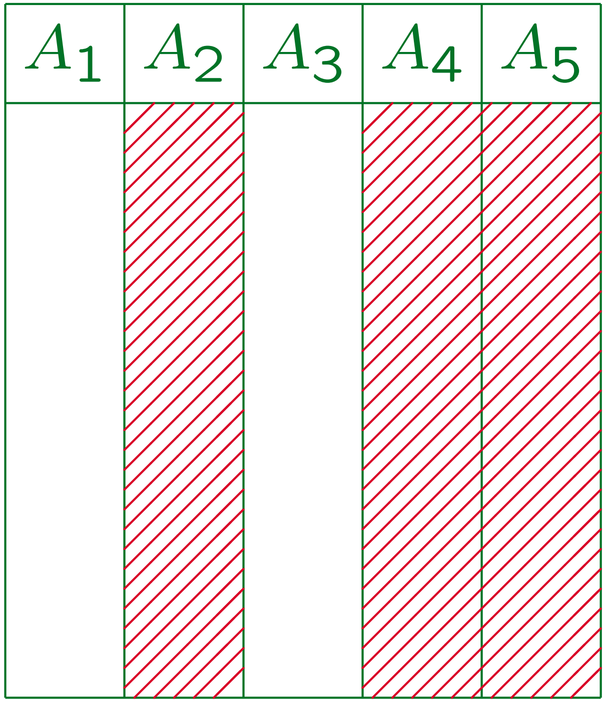
Summary

Selection σ



(Filters some rows)

Projection π



(Maps each row)

Combining Operations (1)

- Since the result of a relational algebra operation is also a relation, it can act as input to another algebra operation.
- For instance, to compute the exercises solved by student 102:

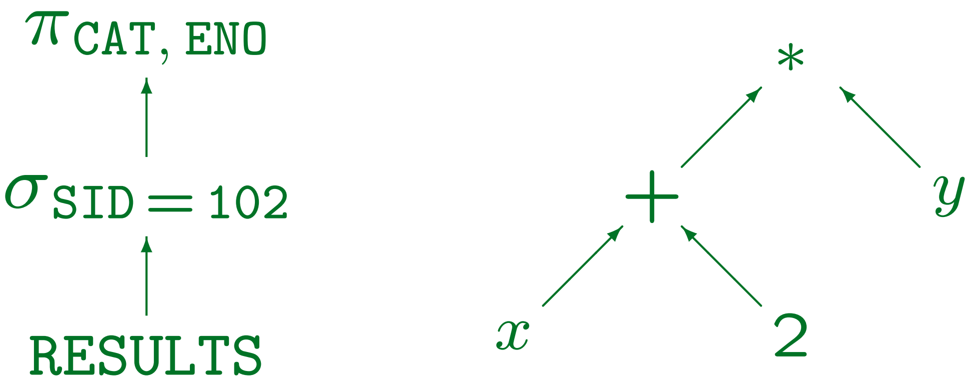
$$\pi_{\texttt{CAT},\,\texttt{ENO}}(\sigma_{\texttt{SID}\,=\,102}(\texttt{RESULTS}))$$

- An intermediate result can be stored in a temporary relation (can also be seen as macro definition):

$$\texttt{S102} \;:=\; \sigma_{\texttt{SID}\,=\,102}(\texttt{RESULTS});$$
$$\pi_{\texttt{CAT},\,\texttt{ENO}}(\texttt{S102})$$

Combining Operations (2)

- Expressions of relational algebra may become clearer if depicted as operator tree:



- For comparison, an operator tree for the arithmetic expression $(x + 2) * y$ is shown on the right.

Intermediate results flow along the lines from bottom to top.

Combining Operations (3)

- In SQL-92, one can also use the result of an SQL query as input to another SQL query:

```
SELECT CAT, ENO
FROM   (SELECT *
        FROM   RESULTS
        WHERE  SID = 102) AS S102
```

- However, this is untypical in SQL, and was not contained in the first SQL standard (SQL-86).
- It is not good programming style to simulate relational algebra in SQL 1:1.

Combining Operations (4)

- In SQL, σ and π (and $\times$, see below) can be combined in a single `SELECT`-expression:

```
SELECT CAT, ENO
FROM   RESULTS
WHERE  SID = 102
```

- Complex queries can be constructed step by step:

```
CREATE VIEW S102
AS SELECT *
   FROM   RESULTS
   WHERE  SID = 102
```

- Then `S102` can be used like a stored table.

Basic Operands

- The leaves of the operator tree are
  - ◇ the names of database relations
  - ◇ constant relations (explicitly enumerated tuples).
- A relation name R is a legal expression of relational algebra. Its value is the entire relation stored under that name. It corresponds to the SQL query:

```
SELECT *
FROM   R
```

- It is not necessary to write a projection on all attributes.

Exercises (1)

Write the following query in relational algebra:

- Print the email address of Ann Smith.

  Write the query as a tree and nested with parentheses.

This refers to the schema on Slide 6-4:

- STUDENTS(<u>SID</u>, FIRST, LAST, EMAIL^{o})
- EXERCISES(<u>CAT</u>, <u>ENO</u>, TOPIC, MAXPT)
- RESULTS(<u>SID</u>→STUDENTS, (<u>CAT</u>, <u>ENO</u>)→EXERCISES, POINTS)

Exercises (2)

- Which of the following relational algebra expressions are syntactically correct? What do they mean?

☐ $\mathtt{STUDENTS}$.

☐ $\sigma_{\mathtt{MAXPT} \neq 10}(\mathtt{EXERCISES})$.

☐ $\pi_{\mathtt{FIRST}}(\pi_{\mathtt{LAST}}(\mathtt{STUDENTS}))$.

☐ $\sigma_{\mathtt{POINTS} \leq 5}(\sigma_{\mathtt{POINTS} \geq 1}(\mathtt{RESULTS}))$.

☐ $\sigma_{\mathtt{POINTS}}(\pi_{\mathtt{POINTS} = 10}(\mathtt{RESULTS}))$.

Overview

1. Introduction, Selection, Projection
2. Cartesian Product, Join
3. Set Operations
4. Outer Join
5. Formal Definitions, A Bit of Theory

Cartesian Product (1)

- Often, answer tuples must be computed that are derived from two (or more) input tuples.

  For σ and π, it holds that each output tuple is derived from a single input tuple. Since π eliminates duplicates, it is possible that the same output tuple is derived from two different input tuples, but then one of the two would already be sufficient.

- This is done by the "cartesian product" $\times$.

  Remember "cartesian coordinates". It is also called "cross product".

- $R \times S$ concatenates ("glues together") each tuple from R with each tuple from S.

  In ASCII, write "`R PRODUCT S`" or "`R x S`" for $R \times S$. If necessary, use $(\ldots)$ around the input relations.

Cartesian Product (2)

- Example:

A	B
1	2
3	4

$\times$

C	D
6	7
8	9

$=$

A	B	C	D
1	2	6	7
1	2	8	9
3	4	6	7
3	4	8	9

- Since attribute names must be unique within a tuple, the cartesian product may only be applied when R and S have no attribute in common.
- This is no real restriction, since we may rename the attributes first (with π) and then apply $\times$.

Cartesian Product (3)

- Some authors define × such that it automatically renames double attributes:
  - ◇ E.g. for relations $R(A, B)$ and $S(B, C)$ the product $R \times S$ has attributes $(R.A, R.B, S.B, S.C)$.
  - ◇ As in SQL, one can also use the names A and C, because they uniquely identify the attributes.
- **In this course, this is not permitted!**

  The formal definition is simpler: What happens e.g. with $(R \cup S) \times T)$ and with $R \times R$? Usually, authors that permit such names do not give a formal definition. In this course, one can use the renaming operator (see below) to introduce attribute names of the form $R.A$, but then A alone cannot be used: Every attribute has exactly one name.

Cartesian Product (4)

- If $t = (A_1{:}\,a_1, \ldots, A_n{:}\,a_n)$, $u = (B_1{:}\,b_1, \ldots, B_m{:}\,b_m)$, let $t \circ u = (A_1{:}\,a_1, \ldots, A_n{:}\,a_n, B_1{:}\,b_1, \ldots, B_m{:}\,b_m)$.
- The cartesian product $R \times S$ can be computed by two nested loops:

```
Create new temporary relation $T$;
foreach tuple $t$ in $R$ do
    foreach tuple $u$ in $S$ do
        insert $t \circ u$ into $T$;
    od;
od;
return $T$;
```

Cartesian Product (5)

- If the relation R contains n tuples, and the relation S contains m tuples, then $R \times S$ contains $n * m$ tuples.
- The cartesian product is in itself seldom useful, because it leads to a “blowup” in relation size.
- The problem is that $R \times S$ combines each tuple from R with each tuple from S. Usually, the goal is to combine only selected pairs of tuples.
- Thus, the cartesian product is useful only as input for a following selection.

Cartesian Product (6)

- $R \times S$ is written in SQL as

```
SELECT *
FROM   R, S
```

- In SQL it is no error if the two relations have common attribute names, since one can reference attributes also in the form "$R.A$" or "$S.A$".

  If the query is executed as above, and R and S both have an attribute called "A", then the result relation will have two different columns with the same name "A". This is forbidden for stored relations, but it can happen for query results (as in this example). One can use nested queries as input relations under `FROM`, but then any try to access the double attribute A in the query gives an error ("column ambiguously defined").

Renaming

- An operator $\rho_R(S)$ that prepends "$R.$" to all attribute names is sometimes useful:

$$\rho_R\left(\begin{array}{|c|c|}\hline A & B \\ \hline 1 & 2 \\ 3 & 4 \\ \hline\end{array}\right) = \begin{array}{|r|r|}\hline R.A & R.B \\ \hline 1 & 2 \\ 3 & 4 \\ \hline\end{array}$$

- This is only an abbreviation for an application of the projection: $\pi_{R.A \leftarrow A,\, R.B \leftarrow B}(S)$.

- Otherwise, attribute names in relational algebra do not automatically contain the relation name.

  Some authors define it that way, but the formal definition is not easy.

Join (1)

- Since this combination of cartesian product and selection is so common, a special symbol has been introduced for it:

  $R \underset{A=B}{\bowtie} S$ is an abbreviation for $\sigma_{A=B}(R \times S)$.

- This operation is called "join": It is used to join two tables (i.e. combine their tuples).

  In ASCII write "`R JOIN[A=B] S`".

- The join is one of the most important and useful operations of the relational algebra.

  Immediately after the selection.

Join (2)

STUDENTS $\bowtie$ RESULTS						
SID	FIRST	LAST	EMAIL	CAT	ENO	POINTS
101	Ann	Smith	···	H	1	10
101	Ann	Smith	···	H	2	8
101	Ann	Smith	···	M	1	12
102	Michael	Jones	(null)	H	1	9
102	Michael	Jones	(null)	H	2	9
102	Michael	Jones	(null)	M	1	10
103	Richard	Turner	···	H	1	5
103	Richard	Turner	···	M	1	7

- Student Maria Brown does not appear, because she has not submitted any homework and did not participate in the exam.
- What is shown above, is the natural join of the two tables. However, in the following first the standard join is explained.

Join (3)

- $R \underset{A=B}{\bowtie} S$ can be evaluated similarly to $\sigma_{A=B}(R \times S)$:

```
Create new temporary relation $T$;
foreach tuple $t$ in $R$ do
    foreach tuple $u$ in $S$ do
        if $t.A = u.B$ then
            insert $t \circ u$ into $T$;
        fi;
    od;
od;
return $T$;
```

Join (4)

- The above procedure is called "nested loop join".
- Note that the intermediate result of $R \times S$ is not materialized (explicitly stored).

  Of course, a real DBMS anyway does not materialize intermediate results unless necessary. Every algebra operator computes tuples only on demand ("pipelined evaluation"). Then the nested loop join is actually the same as a cartesian product followed by a selection.
- Quite a lot of different algorithms have been developed for computing the join.

  E.g. "merge join", "index join", "hash join". The nested loop join is efficient only if one of the two relations is small. Thus, the combined operation can often be executed more efficiently than × followed by σ.

Join (5)

- The join condition does not have to take the form $A = B$ (although this is most common). It can be an arbitrary condition, for instance also $A < B$.

  A join with condition of the form $A = B$ (or $A_1 = B_1 \wedge \cdots \wedge A_n = B_n$) is called an "equijoin".

- A typical application of a join is to combine tuples based on a foreign key, e.g.

$$\mathtt{RESULTS} \underset{\mathtt{SID}=\mathtt{SID}'}{\bowtie} \pi_{\mathtt{SID}' \leftarrow \mathtt{SID}, \mathtt{FIRST}, \mathtt{LAST}, \mathtt{EMAIL}}(\mathtt{STUDENTS})$$

  The renaming of "`SID`" is necessary, because the cartesian product requires disjoint attribute names. But see the natural join below.

Join (6)

- The join not only combines tuples, but also acts as a filter: It eliminates tuples without join partner. (Note: Foreign key ensures that join partner exists.)

A	B
1	2
3	4

$\underset{B=C}{\bowtie}$

C	D
4	5
6	7

$=$

A	B	C	D
3	4	4	5

- A "semijoin" ($\ltimes$, $\rtimes$) works only as a filter.

  It first does the join, but then projects the result tuples on the attributes of the left relation (left semijoin) or right relation (right semijoin).

- An "outer join" (see end of this part) does not work as a filter: It preserves all input tuples.

Natural Join (1)

- Another useful abbreviation is the "natural join" $\bowtie$.

  If you don't have the fancy symbol, you can use "$*$".

- It combines tuples which have equal values in attributes with the same name.

  Whereas the cartesian product as well as the general join require that the attributes of both relations have distinct names, the natural join uses the equal names to derive a join condition.

A	B
1	2
3	4

$\bowtie$

B	C
4	5
4	8
6	7

$=$

A	B	C
3	4	5
3	4	8

Natural Join (2)

- The natural join of two relations
  - ◇ $R(A_1, \ldots, A_n, B_1, \ldots, B_k)$ and
  - ◇ $S(B_1, \ldots, B_k, C_1, \ldots, C_m)$

  produces in database state $\mathcal{I}$ all tuples of the form

$$(a_1, \ldots, a_n, b_1, \ldots, b_k, c_1, \ldots, c_m)$$

  such that
  - ◇ $(a_1, \ldots, a_n, b_1, \ldots, b_k) \in \mathcal{I}(R)$ and
  - ◇ $(b_1, \ldots, b_k, c_1, \ldots, c_m) \in \mathcal{I}(S)$.

Natural Join (3)

- The natural join not only corresponds to a cartesian product followed by a selection, but also
  - automatically renames one copy of each common attribute before the cartesian product, and
  - uses a projection to eliminate these double attributes at the end.
- E.g., given $R(A, B)$, and $S(B, C)$, then $R \bowtie S$ is an abbreviation for

$$\pi_{A,B,C}(\sigma_{B=B'}(R \times \pi_{B' \leftarrow B,\, C}(S))).$$

Natural Join (4)

A Note on Relational Database Design:

- In order to support the natural join, it is beneficial to give attributes from different relations, which are typically joined together, the same name.
- Even if the utilized query language does not have a natural join, this provides good documentation.
- If domain names are used as attribute names, this will happen automatically.
- Try to avoid giving the same name to attributes which will probably not be joined.

Joins in SQL (1)

- $R \underset{A=B}{\bowtie} S$ is normally written in SQL like $\sigma_{A=B}(R \times S)$:

```
SELECT *
FROM   R, S
WHERE  A = B
```

- Attributes can also referenced with explicit relation name (required if the attribute name appears in both relations):

```
SELECT *
FROM   R, S
WHERE  R.A = S.B
```

Joins in SQL (2)

- In SQL-92, one can also write:

```
SELECT *
FROM   R JOIN S ON R.A = S.B
```

- This shows the influence of relational algebra, but it is not really in the spirit of SQL.

  The classical form of the join in SQL (see previous slide) is used for years, and many people think that it is easier to read. Although SQL always had the operation `UNION` from relational algebra, it is otherwise more based on a logical formalism called "tuple relational calculus". The new syntax was probably introduced only because the "outer join" (see below) is more difficult to formulate in the classical way.

- E.g. in Oracle 8i, the new alternative syntax for the join is not supported.

Algebraic Laws (1)

- The join satisfies the associativity condition:

$$(R \bowtie S) \bowtie T \;=\; R \bowtie (S \bowtie T).$$

- Therefore, the parentheses are not needed:

$$R \bowtie S \bowtie T.$$

- The join is not quite commutative: The sequence of columns (from left to right) will be different.
- However, if a projection follows later, this does not matter (one can also introduce π for this purpose):

$$\pi_{\ldots}(R \bowtie S) \;=\; \pi_{\ldots}(S \bowtie R).$$

Algebraic Laws (2)

- Further algebraic laws hold, which are utilized in the query optimizer of a relational DBMS.
- E.g., if the condition φ refers only to S, then

  $$\sigma_\varphi(R \bowtie S) = R \bowtie \sigma_\varphi(S).$$

  The right hand side can often be evaluated more efficiently (depending on relation sizes, indexes).
- But for this course, efficiency is not important.

  The query optimizer transforms a given query automatically into a more efficient variant, so the user does not have to think about this. Full points will be given for any correct solution, except that points will be taken off for unnecessary complications (e.g. π on all columns).

A Common Query Pattern (1)

- The following query structure is very common:

$$\pi_{A_1,\ldots,A_k}(\sigma_\varphi(R_1 \bowtie \cdots \bowtie R_n)).$$

  - ◇ First join all tables which are needed to answer the query.
  - ◇ Second, select the relevant tuples.

    Since the first step is to join the tables, the selection condition may refer to attributes from all of these relations.
  - ◇ Third, project on the attributes which should appear in the output.

A Common Query Pattern (2)

- Patterns are often useful conventions of thought.
- But relational algebra operations can be combined in any way. It is not necessary to adhere to this pattern.

  E.g. if no projection or selection is needed, it is wrong to complicate the query by using a projection on all attributes or a selection with an always true condition (which are simply the identity mapping).

- In contrast, in SQL the keywords `SELECT` and `FROM` are required, and the sequence must always be

  `SELECT ... FROM ... WHERE ...`

A Common Query Pattern (3)

- $\pi_{A_1,\ldots,A_k}(\sigma_\varphi(R_1 \bowtie \cdots \bowtie R_n))$ is written in SQL as:

  SELECT DISTINCT A_1, ..., A_k
  FROM R_1, ..., R_n
  WHERE φ **AND** ⟨Join Conditions⟩

- It is a common mistake to forget a join condition.

  Then one gets a cartesian product, which will give wrong answers and often the query result will be very large.

- Usually, every two relations are linked (directly or indirectly) by equations, e.g. $R_1.B_1 = R_2.B_2$.

- "`DISTINCT`" is not always necessary (see above).

A Common Query Pattern (4)

- To formulate a query, think first about the relations needed:
  - ◇ Usually, the natural language version of the query names certain attributes.

    It is also possible that data values are mentioned that correspond to certain attributes (e.g. student names).
  - ◇ Each such attribute requires at least one relation which contains this attribute.

    So that the attribute can be used in the selection condition or the projection list.

A Common Query Pattern (5)

- Query Formulation, continued:
  - ◇ Finally, sometimes intermediate relations are required in order to make the join meaningful.
  - ◇ E.g., suppose that relations $R(A, B)$, $S(B, C)$, $T(C, D)$ are given and attributes A and D are needed. Then $R \bowtie T$ would not be correct. Why?
  - ◇ Instead, the join must be $R \bowtie S \bowtie T$.
  - ◇ It often helps to have a graphical representation of the foreign key links between the tables (which correspond to typical joins).

Exercise

Write the following queries in relational algebra:

- Print all homework results for Ann Smith (exercise number and points).
- Who has got the full points for a homework? Print first name, last name, and homework number.

This refers to the schema on Slide 6-4:

- $\texttt{STUDENTS}(\underline{\texttt{SID}}, \texttt{FIRST}, \texttt{LAST}, \texttt{EMAIL}^{o})$
- $\texttt{EXERCISES}(\underline{\texttt{CAT}}, \underline{\texttt{ENO}}, \texttt{TOPIC}, \texttt{MAXPT})$
- $\texttt{RESULTS}(\underline{\texttt{SID}}\rightarrow\texttt{STUDENTS}, (\underline{\texttt{CAT}}, \underline{\texttt{ENO}})\rightarrow\texttt{EXERCISES}, \texttt{POINTS})$

Self Joins (1)

- Sometimes, it is necessary to refer to more than one tuple from one relation at the same time.
- E.g. who got more points than student 101 for any exercise?
- In this case, two tuples of the relation `RESULTS` are needed in order to compute one result tuple:
  - ◇ One tuple for the student 101.
  - ◇ One tuple for the same exercise, in which `POINTS` is greater than in the first tuple.

Self Joins (2)

- This requires a generalization of the above query pattern, where two copies of a relation are joined (at least one must be renamed first).

$$S := \rho_{\mathtt{X}}(\mathtt{RESULTS}) \underset{\substack{\mathtt{X.CAT} = \mathtt{Y.CAT}\\ \wedge\, \mathtt{X.ENO} = \mathtt{Y.ENO}}}{\bowtie} \rho_{\mathtt{Y}}(\mathtt{RESULTS});$$

$$\pi_{\mathtt{X.SID}}(\sigma_{\mathtt{X.POINTS}>\mathtt{Y.POINTS} \wedge \mathtt{Y.SID}=\mathtt{101}}(S))$$

- Such joins of a table with itself are sometimes called “self joins”.

Overview

1. Introduction, Selection, Projection
2. Cartesian Product, Join
3. Set Operations
4. Outer Join
5. Formal Definitions, A Bit of Theory

Set Operations (1)

- Since relations are sets (of tuples), the usual set operations $\cup$, $\cap$, $-$ can also be applied to relations.
- However, both input relations must have the same schema.

  For instance, it is not possible to take the union of two relations $R(A)$ and $S(B, C)$, because there is no common schema for the output relation.
- $R \cup S$ contains all tuples which are contained in R, in S, or in both relations (Union).

Set Operations (2)

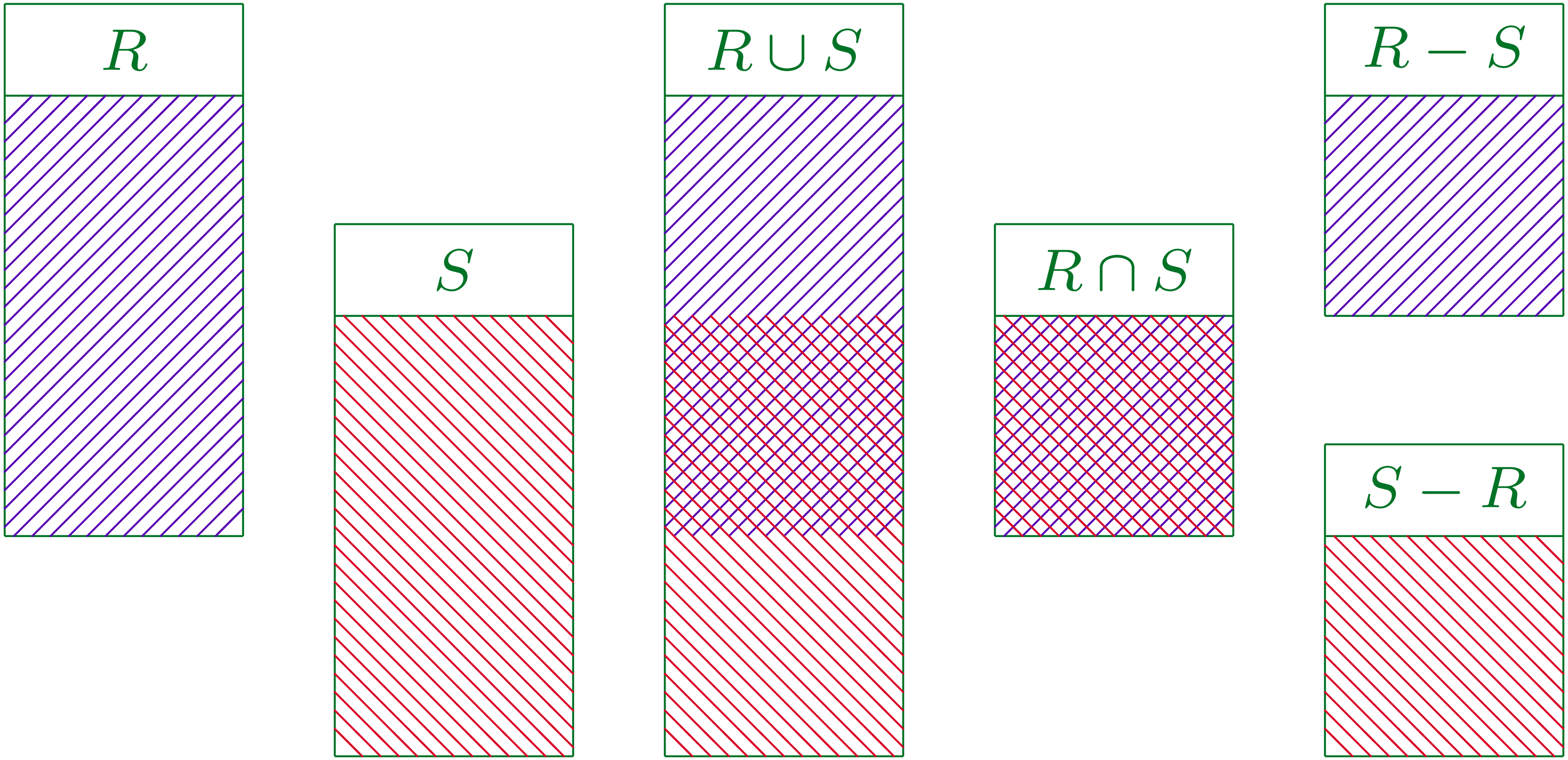
- $R - S$ contains all tuples which are contained in R, but not in S (Set Difference).
- $R \cap S$ contains all tuples which are contained in both, R and S (Intersection).
- Intersection is (like the join) a derived operation: It can be expressed in terms of $-$:

$$R \cap S = R - (R - S).$$

- Exercise: Prove this equation.

  E.g. draw a Venn diagram.

Set Operations (3)



Set Operations (4)

- $R \cup S$ can be implemented as:

```
Create new temporary relation $T$;
foreach tuple $t$ in $R$ do
    insert $t$ into $T$;
od;
foreach tuple $t$ in $S$ do
    insert $t$ into $T$;
od;
return $T$;
```

- **insert** might have to do duplicate elimination.

  In SQL, there are `UNION` (with duplicate elimination) and `UNION ALL` (without duplicate elimination, runs faster).

Set Operations (5)

- $R - S$ can be implemented as:

```
Create new temporary relation $T$;
foreach tuple $t$ in $R$ do
    Remove := false;
    foreach tuple $u$ in $S$ do
        if $u = t$ then
            Remove := true;
    od;
    if not Remove then
        insert $t$ into $T$;
od;
return $T$;
```

Union (1)

- Without $\cup$, every result column can contain only values from a single column of the stored tables.

  Or a single constant. If datatype operations are allowed in the projection, the result value can be computed with a single formula from several input columns, but still this does not give a "Union" behaviour.

- E.g. suppose that besides the registered students, who submit homeworks and write exams, there are also guests that attend the course:

  $\texttt{GUESTS}(\underline{\texttt{FIRST}}, \underline{\texttt{LAST}}, \texttt{EMAIL}^{o})$.

- The task is to produce a list of email addresses of registered students and guests in one query.

Union (2)

- With $\cup$, this is simple:

$$\pi_{\mathtt{EMAIL}}(\mathtt{STUDENTS}) \cup \pi_{\mathtt{EMAIL}}(\mathtt{GUESTS}).$$

- This query cannot be formulated without $\cup$.

- Another typical application of $\cup$ is a case analysis:

$$\mathtt{MPOINTS} := \pi_{\mathtt{SID,POINTS}}(\sigma_{\mathtt{CAT='M'} \wedge \mathtt{ENO=1}}(\mathtt{RESULTS}));$$

$$\begin{aligned}
&\pi_{\mathtt{SID,\ GRADE\leftarrow 'A'}}(\sigma_{\mathtt{POINTS} \geq \mathtt{12}}(\mathtt{MPOINTS}))\\
\cup\ &\pi_{\mathtt{SID,\ GRADE\leftarrow 'B'}}(\sigma_{\mathtt{POINTS} \geq \mathtt{10} \wedge \mathtt{POINTS} < \mathtt{12}}(\mathtt{MPOINTS}))\\
\cup\ &\pi_{\mathtt{SID,\ GRADE\leftarrow 'C'}}(\sigma_{\mathtt{POINTS} \geq \mathtt{7} \wedge \mathtt{POINTS} < \mathtt{10}}(\mathtt{MPOINTS}))\\
\cup\ &\pi_{\mathtt{SID,\ GRADE\leftarrow 'F'}}(\sigma_{\mathtt{POINTS} < \mathtt{7}}(\mathtt{MPOINTS}))
\end{aligned}$$

Union (3)

- In SQL, `UNION` can be written between two `SELECT`-expressions:

```
SELECT SID, 'A' AS GRADE
FROM   RESULTS
WHERE  CAT = 'M' AND ENO = 1 AND POINTS >= 12
UNION
SELECT SID, 'B' AS GRADE
FROM   RESULTS
WHERE  CAT = 'M' AND ENO = 1
AND    POINTS >= 10 AND POINTS < 12
UNION
...
```

Union (4)

- `UNION` was already contained in the first SQL standard (SQL-86) and is supported in all DBMS.
- There is no other way to formulate a union in SQL.

  In contrast, the SQL-92 join operators are not required.
- `UNION`, an algebra operator, is a bit strange in SQL.

  In the theoretical "Tuple Relational Calculus" on which SQL is based, it is possible to declare "tuple variables" that are not bound to a specific relation. Then one can e.g. use a disjunction to talk about tuples that are contained in one of two or more relations. But this also permits "unsafe" queries that are a bit difficult to exclude. Therefore, this possibility was removed in SQL. The price that had to be paid was that the somewhat "foreign" `UNION` operator had to be added.

Set Difference (1)

- The operators σ, π, $\times$, $\bowtie$, $\cup$ have a monotonic behaviour, e.g.

$$R \subseteq S \implies \sigma_\varphi(R) \subseteq \sigma_\varphi(S)$$

- Then it follows that also every query Q that uses only the above operators behaves monotonically:
  - ◇ Let $\mathcal{I}_1$ be a database state, and let $\mathcal{I}_2$ result from $\mathcal{I}_1$ by the insertion of one or more tuples.
  - ◇ Then every tuple t contained in the answer to Q in $\mathcal{I}_1$ is also contained in the answer to Q in $\mathcal{I}_2$.

    I.e. correct answers are never invalidated by an insertion.

Set Difference (2)

- If the query must behave nonmonotonically, it is clear that the previous operations are not sufficient, and one must use set difference “−”. E.g.
  - ◇ Which student has not solved any exercise?
  - ◇ Who got the most points in Homework 1?
  - ◇ Who has solved all exercises in the database?
- Exercise: Give for each of these questions an answer tuple in the example state (repeated on next slide) and give for each such answer a tuple that can be inserted into a table to invalidate that answer.

Set Difference (3)

STUDENTS

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

EXERCISES

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

RESULTS

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

Set Difference (4)

- E.g. which student has not solved any exercise?

$$\mathtt{NO\_SOL} := \pi_{\mathtt{SID}}(\mathtt{STUDENTS}) - \pi_{\mathtt{SID}}(\mathtt{RESULTS});$$

$$\pi_{\mathtt{FIRST},\,\mathtt{LAST}}(\mathtt{STUDENTS} \bowtie \mathtt{NO\_SOL})$$

- Exercise: What is the error in this query?

$$\pi_{\mathtt{SID},\,\mathtt{FIRST},\,\mathtt{LAST}}(\mathtt{STUDENTS}) - \pi_{\mathtt{SID}}(\mathtt{RESULTS})$$

- Is this a correct solution?

$$\pi_{\mathtt{FIRST},\,\mathtt{LAST}}(\mathtt{STUDENTS} \underset{\mathtt{SID} \neq \mathtt{SID2}}{\bowtie} \pi_{\mathtt{SID2} \leftarrow \mathtt{SID}}(\mathtt{RESULTS}))$$

Set Difference (5)

- When using $-$, a typical pattern is the anti-join.
- E.g. given $R(A, B)$ and $S(B, C)$, the tuples from R that do not have a join partner in S can be computed as follows:

$$R \bowtie (\pi_B(R) - \pi_B(S)).$$

- The following is equivalent: $R - \pi_{A,B}(R \bowtie S)$.

  In both cases, the problem is that the set difference requires the same schema on both sides. Therefore, one needs also projection and join.

- A symbol for the anti-join is not common, but one could use $R \mathbin{\overline{\ltimes}} S$ (a complemented semi-join).

Set Difference (6)

- Note that in order for the set difference $R - S$ to be applicable, it is not (!) required that $S \subseteq R$.

  In one exam, quite a lot of the solutions contained unnecessary complications that could be attributed to this misunderstanding.

- E.g. this query computes the SIDs of students that have solved Homework 2, but not Homework 1:

$$\begin{array}{rl} & \pi_{\mathtt{SID}}(\sigma_{\mathtt{CAT}='\mathtt{H}' \wedge \mathtt{ENO}=2}(\mathtt{RESULTS})) \\ - & \pi_{\mathtt{SID}}(\sigma_{\mathtt{CAT}='\mathtt{H}' \wedge \mathtt{ENO}=1}(\mathtt{RESULTS})) \end{array}$$

- It is no problem that there might also be students that have solved Homework 1, but not Homework 2.

Set Difference (7)

- Suppose that R and S are represented in SQL as
  - ◇ `SELECT` A_1, ..., A_n `FROM` R_1, ..., R_m `WHERE` φ_1
  - ◇ `SELECT` B_1, ..., B_n `FROM` S_1, ..., S_k `WHERE` φ_2
- Then $R - S$ can be represented as

```
SELECT A1, ..., An
FROM   R1, ..., Rm
WHERE  φ1 AND NOT EXISTS
         (SELECT * FROM S1, ..., Sk
          WHERE  φ2
          AND    B1 = A1 AND ... AND Bn = An)
```

Set Difference (8)

- The `NOT EXISTS`-condition is true if the subquery returns 0 answers.

  Subqueries are explained in much more detail in Chapter 8 (SQL II). The subquery is evaluated once for each tuple $A_1, \ldots, A_n$ computed in the main query. The subquery gives then a non-empty result only if the second query can compute the same tuple.

- If one uses "`NOT EXISTS`" in SQL, one has automatically the "anti-join": It is not necessary to project attributes away and restore them later with a join.
- SQL-92 also has "`EXCEPT`" that can be used like `UNION`. E.g. not in Oracle 8 (there "`MINUS`").

Exercises

Write the following queries in relational algebra:

- Who got the most points in Homework 1?

  Hint: Compute first students who did not get the most points, i.e. for which there is a student with more points. Then use set difference.

- Which students solved all exercises in the database?

This refers to the schema on Slide 6-4:

- `STUDENTS(SID, FIRST, LAST, EMAIL`^{O}`)`
- `EXERCISES(CAT, ENO, TOPIC, MAXPT)`
- `RESULTS(SID→STUDENTS, (CAT, ENO)→EXERCISES, POINTS)`

Union vs. Join

- Two alternative representations of the homework, midterm, and final totals of the students are:

Results_1

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

Results_2

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

- Give algebra expressions to translate between them.

Summary

The five basic operations of relational algebra are:

- σ_φ: Selection
- $\pi_{A_1,\ldots,A_k}$: Projection
- $\times$: Cartesian Product
- $\cup$: Union
- $-$: Set Difference

Derived operations: The general join $\underset{\varphi}{\bowtie}$, the natural join $\bowtie$, the renaming operator ρ, the intersection $\cap$.

Overview

1. Introduction, Selection, Projection
2. Cartesian Product, Join
3. Set Operations
4. Outer Join
5. Formal Definitions, A Bit of Theory

Outer Join (1)

- The usual join eliminates tuples without partner:

$$\begin{array}{|c|c|}\hline A & B \\ \hline a_1 & b_1 \\ a_2 & b_2 \\ \hline\end{array} \bowtie \begin{array}{|c|c|}\hline B & C \\ \hline b_2 & c_2 \\ b_3 & c_3 \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline A & B & C \\ \hline a_2 & b_2 & c_2 \\ \hline\end{array}$$

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

$$\begin{array}{|c|c|}\hline A & B \\ \hline a_1 & b_1 \\ a_2 & b_2 \\ \hline\end{array} ⟕ \begin{array}{|c|c|}\hline B & C \\ \hline b_2 & c_2 \\ b_3 & c_3 \\ \hline\end{array} = \begin{array}{|c|c|c|}\hline A & B & C \\ \hline a_1 & b_1 & \\ a_2 & b_2 & c_2 \\ \hline\end{array}$$

Rows from the left table are filled with “null” if necessary.

Outer Join (2)

- The right outer join preserves tuples from the right table:

A	B
a_1	b_1
a_2	b_2

⟖

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_2	b_2	c_2
	b_3	c_3

- The full outer join does not eliminate any tuples:

A	B
a_1	b_1
a_2	b_2

⟗

B	C
b_2	c_2
b_3	c_3

$=$

A	B	C
a_1	b_1	
a_2	b_2	c_2
	b_3	c_3

Outer Join (3)

$R \mathbin{⟕}_{A=B} S$:

```
Create new temporary relation $T$;
**foreach** tuple $t$ **in** $R$ **do**
    HasJoinPartner := **false**;
    **foreach** tuple $u$ **in** $S$ **do**
        **if** $t.A = u.B$ **then**
            **insert** $t \circ u$ **into** $T$;
            HasJoinPartner := **true**;
        **fi**;
    **od**;
    **if not** HasJoinPartner **then**
        **insert** $t \circ$ (null, . . . , null) **into** $T$;
**od**;
**return** $T$;
```

Outer Join (4)

- E.g. students with their homework results, students without homework result are listed with null values:

$$\texttt{STUDENTS} ⟕ \pi_{\texttt{SID,ENO,POINTS}}(\sigma_{\texttt{CAT='H'}}(\texttt{RESULTS}))$$

SID	FIRST	LAST	EMAIL	ENO	POINTS
101	Ann	Smith	$\cdots$	1	10
101	Ann	Smith	$\cdots$	2	8
102	Michael	Jones	(null)	1	9
102	Michael	Jones	(null)	2	9
103	Richard	Turner	$\cdots$	1	5
104	Maria	Brown	$\cdots$	(null)	(null)

Outer Join (5)

- Exercise: Is there any difference between
  - ◇ STUDENTS ⋈ RESULTS and
  - ◇ STUDENTS ⟖ RESULTS?
- The outer join is especially useful together with aggregation functions (e.g. `count`, `sum`), see below.

  Aggregation functions are only introduced for SQL in this course.
- With a selection on the outer join result, one can use it like a set difference: But questionable style.

  Necessary in MySQL (has no subqueries).

Outer Join (6)

- The outer join is a derived operation (like $\bowtie$, $\cap$), i.e. it can be simulated with the five basic relational algebra operations.
- E.g. consider relations $R(A, B)$ and $S(B, C)$.
- The left outer join $R ⟕ S$ is an abbreviation for

  $$R \bowtie S \quad \cup \quad (R - \pi_{A,B}(R \bowtie S)) \times \{(C\colon null)\}$$

  (where $\bowtie$ can be further replaced by $\times$, σ, π).

  I.e. the outer join adds to the normal join result those tuples from R that do not have a join partner (filled with $C\colon null$ to get the same schema, because otherwise the union would not be applicable).

Outer Join (7)

- The SQL-86 standard had no explicit joins. Since joins including the outer join can be simulated with other constructs, this is no real problem.
- However, it turned out that some queries become much shorter if the outer join can be used.
- Therefore, the outer join was added in SQL-92:

```
SELECT R.A, R.B, S.C
FROM   R LEFT OUTER JOIN S ON R.B = S.B
```

- But in this way, SQL became a quite complex mixture of relational algebra and tuple calculus.

Overview

1. Introduction, Selection, Projection
2. Cartesian Product, Join
3. Set Operations
4. Outer Join
5. Formal Definitions, A Bit of Theory

Definitions: Syntax (1)

Let the following be given:

- A set $\mathcal{S}_\mathcal{D}$ of data type names, and for each $D \in \mathcal{S}_\mathcal{D}$ a set $val(D)$ of values.

  As mentioned before, for simplicity we do not distinguish between constants/literals and the values they denote.

- A set $\mathcal{A}$ of possible attribute names (identifiers).
- A relational database schema $\mathcal{S}$ that consists of
  - ◇ a finite set of relation names $\mathcal{R}$, and
  - ◇ for every $R \in \mathcal{R}$, a relation schema $sch(R)$.

  (Constraints are not important here.)

Definitions: Syntax (2)

- One recursively defines the set of relational algebra (RA) expressions (queries) together with the relation schema of each RA expression. Base Cases:
  - ◇ $\boxed{R}$: For every $R \in \mathcal{R}$, the relation name R is an RA expression with schema $sch(R)$.
  - ◇ $\boxed{\{(A_1{:}\, d_1, \ldots, A_n{:}\, d_n)\}}$ ("relation constant") is an RA expression if $A_1, \ldots, A_n \in \mathcal{A}$, and $d_i \in val(D_i)$ for $1 \leq i \leq n$ with $D_1, \ldots, D_n \in \mathcal{S}_{\mathcal{D}}$. The schema of this RA expression is $(A_1{:}\, D_1, \ldots, A_n{:}\, D_n)$.

Definitions: Syntax (3)

- Recursive cases, Part 1: Let Q be an RA expression with schema $\rho = (A_1\colon D_1, \ldots, A_n\colon D_n)$. Then also the following are RA expressions:
  - ◇ $\boxed{\sigma_{A_i=A_j}(Q)}$ for $i, j \in \{1, \ldots, n\}$. It has schema ρ.
  - ◇ $\boxed{\sigma_{A_i=d}(Q)}$ for $i \in \{1, \ldots, n\}$ and $d \in val(D_i)$. It has schema ρ.
  - ◇ $\boxed{\pi_{B_1 \leftarrow A_{i_1}, \ldots, B_m \leftarrow A_{i_m}}(Q)}$ for $i_1, \ldots, i_k \in \{1, \ldots, n\}$ and $B_1, \ldots, B_m \in \mathcal{A}$ such that $B_j \neq B_k$ for $j \neq k$. It has schema $(B_1\colon D_{i_1}, \ldots, B_m\colon D_{i_m})$.

Definitions: Syntax (4)

- Recursive cases, continued: Let Q_1 and Q_2 be RA expressions with the same schema ρ. Then also the following are RA expressions with schema ρ:
  - $\boxed{(Q_1) \cup (Q_2)}$ and ◇ $\boxed{(Q_1) - (Q_2)}$
- Let Q_1 and Q_2 be RA expressions with the schemas $(A_1\colon D_1, \ldots, A_n\colon D_n)$ and $(B_1\colon E_1, \ldots, B_n\colon E_m)$, resp. If $\{A_1, \ldots, A_n\} \cap \{B_1, \ldots, B_m\} = \emptyset$, then also the following is an RA expression:
  - $\boxed{(Q_1) \times (Q_2)}$

    Schema: $(A_1{:}D_1, \ldots, A_n{:}D_n, B_1{:}E_1, \ldots, B_m{:}E_m)$.

Definitions: Syntax (5)

- Nothing else is a relational algebra expression.

  This is formally necessary to complete the definition. The definition consists otherwise only of conditions of the form "If R is an RA expression, then S is an RA expression." This would permit that everything is an RA expression (the conclusion of the rules is then always true, thus the rules are satisfied). This is of course not meant by the definition. Therefore, it is necessary to state that something is an RA expression only if can really be constructed by a finite number of applications of the above rules, because "nothing else is an RA expression".

- Exercise: Define a context free grammar for relational algebra expressions (without the restrictions about declared attribute names).

Abbreviations

- Parentheses can be left out if the structure is clear (or the possible structures are equivalent).

  The definition above requires a lot of parentheses in order to make sure with simple rules that the structure is always uniquely determined. With more complex rules, it is possible to reduce the number of parentheses. One can also define binding strengths (operator precedences), e.g. $\times$ binds stronger than $\cup$. However, for a theoretical investigation, this is not really important.

- As explained above, additional algebra operations (like the join) can be introduced as abbreviations.

  Again, for the practical usage of the query language, this is important, but not for theoretical results, since the abbreviations can always be expanded to their full form.

Definitions: Semantics (1)

- A database state $\mathcal{I}$ defines a finite relation $\mathcal{I}(R)$ for every relation name R in the database schema.

  If $sch(R) = (A_1: D_1, \ldots, A_n: D_n)$, then $\mathcal{I}(R) \subseteq val(D_1) \times \cdots \times val(D_n)$.

- The result of a query Q, i.e. an RA expression, in a database state $\mathcal{I}$ is a relation. The query result is written $\mathcal{I}[Q]$ and defined recursively corresponding to the structure of Q:
  - ◇ If Q is a relation name R, then $\mathcal{I}[Q] := \mathcal{I}(R)$.
  - ◇ If Q is a constant relation $\{(A_1: d_1, \ldots, A_n: d_n)\}$, then $\mathcal{I}[Q] := \{(d_1, \ldots, d_n)\}$.

Definitions: Semantics (2)

- Definition of the result $\mathcal{I}[Q]$ of an RA expression Q in state $\mathcal{I}$, continued:
  - ◇ If Q has the form $\sigma_{A_i=A_j}(Q_1)$, then

    $$\mathcal{I}[Q] := \{(d_1, \ldots, d_n) \in \mathcal{I}[Q_1] \mid d_i = d_j\}.$$
  - ◇ If Q has the form $\sigma_{A_i=d}(Q_1)$, then

    $$\mathcal{I}[Q] := \{(d_1, \ldots, d_n) \in \mathcal{I}[Q_1] \mid d_i = d\}.$$
  - ◇ If Q has the form $\pi_{B_1 \leftarrow A_{i_1}, \ldots, B_m \leftarrow A_{i_m}}(Q_1)$, then

    $$\mathcal{I}[Q] := \{(d_{i_1}, \ldots, d_{i_m}) \mid (d_1, \ldots, d_n) \in \mathcal{I}[Q_1]\}.$$

Definitions: Semantics (3)

- Definition of $\mathcal{I}[Q]$, continued:
  - ◇ If Q has the form $(Q_1) \cup (Q_2)$ then
  $$\mathcal{I}[Q] := \mathcal{I}[Q_1] \cup \mathcal{I}[Q_2].$$
  - ◇ If Q has the form $(Q_1) - (Q_2)$ then
  $$\mathcal{I}[Q] := \mathcal{I}[Q_1] - \mathcal{I}[Q_2].$$
  - ◇ If Q has the form $(Q_1) \times (Q_2)$, then
  $$\begin{aligned}\mathcal{I}[Q] := \{(d_1, \ldots, d_n, e_1, \ldots, e_m) \mid \\ (d_1, \ldots, d_n) \in \mathcal{I}[Q_1], \\ (e_1, \ldots, e_m) \in \mathcal{I}[Q_2]\}.\end{aligned}$$

Monotonicity

- Definition: A database state $\mathcal{I}_1$ is smaller than (or equal to) a database state $\mathcal{I}_2$, written $\mathcal{I}_1 \subseteq \mathcal{I}_2$, if and only if $\mathcal{I}_1(R) \subseteq \mathcal{I}_2(R)$ for all relation names R in the schema.
- Theorem: If an RA expression Q does not contain the $-$ (set difference) operator, then the following holds for all database states $\mathcal{I}_1, \mathcal{I}_2$:

$$\mathcal{I}_1 \subseteq \mathcal{I}_2 \implies \mathcal{I}_1[Q] \subseteq \mathcal{I}_2[Q].$$

- Exercise: Prove this by induction on the structure of Q (“structural induction”).

Equivalence (1)

- **Definition:** Two RA expressions Q_1 and Q_2 are equivalent if and only if they have the same schema and for all database states $\mathcal{I}$ the following holds:

$$\mathcal{I}[Q_1] = \mathcal{I}[Q_2].$$

- There are two notions of equivalence, depending on whether one considers all structurally possible states or only states that satisfy the constraints.

  The first alternative is a stricter requirement. The second alternative gives more pairs of equivalent queries. In the following, it is not important which version is chosen.

Equivalence (2)

- Examples for equivalences:
  - ◇ $\sigma_{\varphi_1}(\sigma_{\varphi_2}(Q))$ is equivalent to $\sigma_{\varphi_2}(\sigma_{\varphi_1}(Q))$.
  - ◇ $(Q_1 \times Q_2) \times Q_3$ is equivalent to $Q_1 \times (Q_2 \times Q_3)$.
  - ◇ If A is an attribute in the schema of Q_1:
    $\sigma_{A=d}(Q_1 \times Q_2)$ is equivalent to $(\sigma_{A=d}(Q_1)) \times Q_2$
- Theorem: The equivalence of relational algebra expressions is undecidable.

  I.e. one cannot write a program that reads two arbitrary RA expressions Q_1 and Q_2 and outputs "yes" or "no" depending on whether Q_1 and Q_2 are equivalent and is guaranteed to stop after a finite amount of computation time.

Limitations of RA (1)

- Let R be a relation name with schema $(A{:}\, D, B{:}\, D)$ and let $val(D)$ be infinite.
- The transitive closure of $\mathcal{I}(R)$ is the set of all $(d, e) \in val(D) \times val(D)$ such that there are $n \in \mathbb{N}$ $(n \geq 1)$ and $d_0, \ldots, d_n \in val(D)$ with $d = d_0$, $e = d_n$ and $(d_{i-1}, d_i) \in \mathcal{I}(R)$ for $i = 1, \ldots, n$.
- E.g. R could be the relation "`PARENT`", then the transitive closure are all ancestor-relationships (parents, grandparents, great-grandparents, . . .).

Limitations of RA (2)

- Theorem: There is no RA expression Q such that $\mathcal{I}[Q]$ is the transitive closure of $\mathcal{I}(R)$ for all database states $\mathcal{I}$.
- E.g. in the ancestor example, one would need an additional join for every additional generation.
- Therefore, if one does not know, how many generations the database contains, one cannot write a query that works for all possible database states.

  Of course, one can write a query that works up to e.g. the great-grandparents. But then it does not work correctly if the database should contain great-great-grandparents.

Limitations of RA (3)

- This of course implies that relational algebra is not computationally complete:
  - ◇ Not every function from database states to relations for which a C program exists can be formulated in relational algebra.
  - ◇ However, this can also not be expected, since one wants to be sure that query evaluation always terminates. This is guaranteed for RA.

    It is undecidable whether general programs terminate ("halting problem"). Therefore any computationally complete language necessarily permits to formulate queries/programs that sometimes do not terminate and there is no way to check this beforehand.

Limitations of RA (4)

- All RA queries can be computed in time that is polynomically in the size of the database.
- Thus, also very complex functions cannot be formulated in relational algebra.

  E.g. if you should find a way to formulate the Travelling Salesman Problem in relational algebra, you had solved the famous P=NP problem (with a solution that nobody expects). Since that is extremely unlikely, you should not try it, but instead write a C program.
- As the transitive closure shows, not all problems of polynomial complexity can be formulated in RA.

  With a fixpoint operator and a linear order on the domain, this is possible (→ Deductive DB).

Expressive Power (1)

- A query language $\mathcal{L}$ for the relational model is called **strong relationally complete** if for every database schema $\mathcal{S}$ and for every RA expression Q_1 with respect to $\mathcal{S}$ there is a query $Q_2 \in \mathcal{L}_{\mathcal{S}}$ such that for all database states $\mathcal{I}$ with respect to $\mathcal{S}$ the two queries produce the same result: $\mathcal{I}[Q_1] = \mathcal{I}[Q_2]$.
- I.e. the requirement is that every relational algebra expression can be translated into an equivalent query in that language.

Expressive Power (2)

- E.g. SQL is strong relationally complete.
- If the translation of queries is possible in both directions, the two query languages have the same expressive power.

  E.g. SQL contains aggregations that cannot be simulated in relational algebra. Thus, SQL is more powerful than relational algebra. But one can of course extend relational algebra with aggregation operations.

- "Relationally complete" (without "strong") permits to use a sequence of queries and to store intermediate results in temporary relations.

Expressive Power (3)

- The following languages have the same expressive power (queries can be translated between them):
  - ◇ Relational algebra
  - ◇ SQL without aggregations and with mandatory duplicate elimination.
  - ◇ Tuple relational calculus (first order logic with variables for tuples, see below), Domain RC
  - ◇ Datalog (a Prolog variant) without recursion
- Thus, the set of functions that can be expressed in RA is at least not arbitrary.