Relational Algebra: Review

- We will see that the operators of set theory are a good basis for manipulating relations (as they are sets indeed), but that they have deficiencies and thus have to be amended and extended.

- Already in his seminal paper introducing relational databases Codd introduced a choice of operators particularly tailored for dealing with relations. This was the basis of the formal language called the relational algebra today.

- Relational algebra is a mathematical language and thus not particularly user-friendly. But its operators have been incorporated into most of the query languages for relational databases in use today (e.g., in SQL). Thus, it is important to know about them.

- Moreover, relational algebra is used internally by a DBMS for evaluating queries written in SQL (or other languages). SQL queries are compiled into relational algebra expressions and then transformed into equivalent formulations which can be evaluated more efficiently ("query optimization").
> Algebra

- An algebra is a system of operators manipulating objects in a particular carrier set, i.e.
  - all input parameters are taken from this set, and
  - the result after applying the operators is contained in the carrier as well.

- consequence: Operators can be applied to results of previous operator applications, i.e., nesting of operators is possible.

- e.g.: arithmetic (numbers, + / * / −),
  propositional logic (truth values, or / and / not ),
  set algebra (power set of a set, ∪ / ∩ / −)

- The relational algebra (RA) is a special variant of set algebra, the carrier set of which consists in particular of relations rather than arbitrary objects. Arguments of RA-operators as well as their results are relations.
Set Operators for Relations

- Relations are (special) sets, and thus operators of set algebra are applicable to relations, too:

  union: \( R \cup S \)  
  difference: \( R - S \)  
  intersection: \( R \cap S \)  
  product: \( R \times S \)

- Intersection can be expressed via difference:

  \[ R \cap S = R - (R - S) \]

- **Attention!** Even if all input parameters of one of these operators are relations, it is by no means guaranteed that the results are relations, too. It may well be that applying a set operator to relations returns „just“ an ordinary set, but not a relation!

- Thus, **not every** application of set operators in RA is defined!
### Set Operators for Relations (2)

**R**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

**S**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>

---

**Union of two inhomogeneous relations . . .**

\[
R \cup S
\]

... results in a set, ...

\[
(1,2,3) \\
(a,b) \\
(2,4,5) \\
(c,d) \\
(3,6,9) \\
(x,y)
\]

... but **not** in a relation!
Union Compatibility

- Only „similar“ relations can be united, intersected or subtracted. For product, however, similarity is not required.

- Relations the union of which is a relation again, are called union compatible

- „Similarity“ of relations can be defined in various ways, a minimal requirement being
  - identical arity
  - identity of types of all columns.

- In addition, identity of names of all columns in both relations is often required.

- Identity of names can be reached by systematic renaming of columns. RA has an „auxiliary“ operator $\rho$ (Greek rho) for denoting renamings, e.g.:

  $$\rho_{A \leftarrow B}(R)$$

  [In relation $R$, column $A$ is renamed into $B$.]
Product of Two Relations

- In set theory, the **product** of two relations is **always** a binary relation, the elements of which are pairs of tuples.

  \[
  A \times B = \{ (a, b) \mid a \in A \text{ and } b \in B \}
  \]

- If e.g. tuple \((a,b)\) is an element of the binary relation \(A\) and tuple \((1,2,3)\) is an element of the ternary relation, then the product of \(A\) and \(B\) contains the pair \((a,b,1,2,3)\).

- In relational algebra, however, the product operator is defined in a slightly (but distinctively) different manner: Tuples from both operand relations are **concatenated** into a single tuple before being entered into the product relation:

  \[
  (a,b) \rightarrow (a,b,1,2,3) \quad \text{and} \quad (1,2,3) \rightarrow (a,b,1,2,3)
  \]

- Thus, in RA the product of an \(n\)-ary and an \(m\)-ary relation is an \((n+m)\)-ary (but not a binary relation)!
Renaming While Building Products

- If constructing the product of two relations, **renaming** of columns may be necessary in order to ensure that all columns of the result relation have different names.

- In order to resolve ambiguities, one often uses the name of the origin relation of a column as **prefix** for attributes in the result relation:
Special Operators for Sets of Tuples

- In addition to the set operators (adapted to relations) RA offers another selection of **special operators**, defined for sets of tuples only.

  - **Basic operators** for tuple sets (i.e. relations) are two unary operators for extracting . . .
    - . . . certain *columns*:
      - projection $\pi$
    - . . . certain *rows* (tuples):
      - selection $\sigma$

- Apart from these, there are various **derived operators** based upon projection and selection (in combination with set operators):
  - The various forms of the *join* operator are variants of product:
    - inner/outer join, natural join
  - A very special form of difference is very helpful for formalizing certain variants of universal quantification in set theory: *division*
The projection operator \( \pi \) „officially“ has one relational parameter only, but additionally needs one or more columns of the operand relation as a kind of „auxiliary parameters“ indicating on which columns to project.

In principle, one ought to say that there are very many different projection operators instead of just one: Per combination of columns on which to project there should be one such operator. For simplicity’s sake, however, a „compromise notation“ is used:

All columns not appearing as an index of \( \pi \) are eliminated by projection.
Projection and Duplicate Elimination

- While applying projection it may happen that the result relation contains duplicates – tuples occurring more than once.

- This may even be the case if the input relation itself was free of duplicates (which ought to be the case for each proper relation, as a set, anyway!).

\[ \pi_{A,B} \]

In order to be able to return a relation again, it may be necessary to eliminate duplicates (which may be an expensive task for large relations).

- Projection and union are the only basic operations of RA requiring duplicate elimination.
Selection

- The selection operator $\sigma$ – which is unary in principle, too – needs an „auxiliary parameter“ as well.

- A condition in the syntax of propositional logic, composed of comparisons of column values, called selection condition is added to $\sigma$. All tuples of the input relation not satisfying this condition are eliminated.

- Selection conditions consist of column names of $R$, constants, comparison operators ($=, \neq, <, \leq, >, \geq$) and logical connectives ($\wedge, \lor, \neg$).
Example for using the selection operator:

Find all tuples in relation R, the A-field of which is bigger than the C-field, and the B-field of which is not 'b'!

Formulation of this query in relational algebra:

\[ \sigma_{A > C \land B \neq 'b'}(R) \]
Natural Join

- The „full“ product of two relations is not very useful in most situations. Very often a product is immediately reduced by eliminating rows and columns.

- The most frequently used such variant results in two tables being connected via one or more of their columns on identical values in each of these columns, e.g.:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

- The most natural way of building a table containing all such „connections“ is by looking for identical values in columns with identical name and then to concatenate the tuples thus linked (similarly to building a product relation). Due to the identical values, however, it is sufficient to keep only one copy of the joined columns (rather than two as in a product):
Example of a natural join:
Instead of the 9 tuples of a full product, only 3 „meaningful“ combinations of tuples are kept!
The natural join is a derived operator in RA as its effect could as well be reached by combining projection, selection and product:

\[
R \bowtie S = \pi_{A_1, \ldots, A_m, R.B_1, \ldots, R.B_k, C_1, \ldots, C_n} (\sigma_{R.B_1 = S.B_1 \land \ldots \land R.B_k = S.B_k} (R \times S))
\]

Here, \(\text{attr}(R)\) denotes the set of all column names (attributes) in \(R\).

Obviously, using the special operator results in much higher readability of the expression.
Inner Join

- It is not always clear that concatenation of tuples based on identity is indeed intended. For expressing explicit join conditions there is the inner join:
  - no automatic selection of tuples with identical fields
  - no automatic projection on "relevant" columns

- Example of an inner join: \( R \bowtie S \) with \( \Theta = (R.A \leq S.C \land S.B > 0) \)

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>R.B</th>
<th>S.B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

- The join condition \( \Theta \) is syntactically constructed like a selection condition.
- In Access, only the inner join is supported (no natural join!).
> Outer Join

- There is an *outer join* as well, which extends the inner join by maintaining the information about non-matching tuples in both input relations by „joining“ them with special „null values“ representing the fact that there is no match.

- Example of an outer join: \[ R \bowtie S \text{ mit } \Theta = (R.A \leq S.C \land S.B > 0) \]

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>R.B</th>
<th>S.B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- If only non-matching tuples from one of the partner relations are to be filled up with Null, **left or right outer join** is to be used: \[ \bowtie \text{ resp. } \bowtie \]
Division

- Most sophisticated, but also quite useful operator of RA: division
- formal notation like in arithmetic: \( R \div S \)
- general idea: algebraic counterpart to universal quantification in logic (for all)
- principle of division: Which A-values appear in R combined with all S-tuples?

Only 'a' appears in R combined with all S-tuples!

precise definition of division (nearly unreadable for normal users again):

\[
R \div S := \pi_{\text{attr}(R) - \text{attr}(S)}(R) - \pi_{\text{attr}(R) - \text{attr}(S)}((\pi_{\text{attr}(R)} - \text{attr}(S))(R \times S) - R)
\]
Relational Algebra: Summary

- The following operators are comprised by the relational algebra:

<table>
<thead>
<tr>
<th>base operators</th>
<th>derivable operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>intersection</td>
</tr>
<tr>
<td>difference</td>
<td>join</td>
</tr>
<tr>
<td>product</td>
<td>division</td>
</tr>
<tr>
<td>projection</td>
<td></td>
</tr>
<tr>
<td>selection</td>
<td></td>
</tr>
</tbody>
</table>

- Query languages able to express at least (the effect of) each RA operator are called relationally complete. Thus, RA serves as a measure for the expressive power of DB query languages.

- SQL – to be presented in the next chapter – is a relationally complete language, but exceeds RA in expressivity.
Query processing using relational algebra: An example:

**schema:**

- Lectures: $\{\text{LecID: integer, Title: string, Credits: integer, Held_By: integer}\}$
- Professors: $\{\text{PersID: integer, Name: string, Position: string, Room: integer}\}$
- Students: $\{\text{StudID: integer, Name: string, Semester: integer}\}$
- attended_by: $\{\text{StudID: integer, LecID: integer}\}$

**query:**

Which student under those who attend all the lectures in this semester has the smallest student ID?
Schema state

attended_by

<table>
<thead>
<tr>
<th>StudID</th>
<th>LecID</th>
</tr>
</thead>
<tbody>
<tr>
<td>26120</td>
<td>4001</td>
</tr>
<tr>
<td>27550</td>
<td>4001</td>
</tr>
<tr>
<td>27550</td>
<td>4002</td>
</tr>
<tr>
<td>28106</td>
<td>4001</td>
</tr>
<tr>
<td>28106</td>
<td>4002</td>
</tr>
<tr>
<td>28106</td>
<td>4003</td>
</tr>
<tr>
<td>29120</td>
<td>4001</td>
</tr>
<tr>
<td>29120</td>
<td>4002</td>
</tr>
<tr>
<td>29120</td>
<td>4003</td>
</tr>
<tr>
<td>29555</td>
<td>4003</td>
</tr>
<tr>
<td>30112</td>
<td>4001</td>
</tr>
<tr>
<td>30112</td>
<td>4002</td>
</tr>
<tr>
<td>30112</td>
<td>4003</td>
</tr>
<tr>
<td>31403</td>
<td>4002</td>
</tr>
</tbody>
</table>

Students

<table>
<thead>
<tr>
<th>StudID</th>
<th>Name</th>
<th>Semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>24002</td>
<td>Xenokrates</td>
<td>18</td>
</tr>
<tr>
<td>25403</td>
<td>Jonas</td>
<td>12</td>
</tr>
<tr>
<td>26120</td>
<td>Fichte</td>
<td>10</td>
</tr>
<tr>
<td>26830</td>
<td>Aristoxenos</td>
<td>8</td>
</tr>
<tr>
<td>27550</td>
<td>Schopenhauer</td>
<td>6</td>
</tr>
<tr>
<td>28106</td>
<td>Carnap</td>
<td>3</td>
</tr>
<tr>
<td>29120</td>
<td>Theophrastos</td>
<td>2</td>
</tr>
<tr>
<td>29555</td>
<td>Feuerbach</td>
<td>2</td>
</tr>
</tbody>
</table>

Lectures

<table>
<thead>
<tr>
<th>LecID</th>
<th>Title</th>
<th>Credits</th>
<th>Held_By</th>
</tr>
</thead>
<tbody>
<tr>
<td>4001</td>
<td>logic</td>
<td>4</td>
<td>2125</td>
</tr>
<tr>
<td>4002</td>
<td>knowledge theory</td>
<td>3</td>
<td>2126</td>
</tr>
<tr>
<td>4003</td>
<td>database systems</td>
<td>4</td>
<td>2137</td>
</tr>
</tbody>
</table>

Professors

<table>
<thead>
<tr>
<th>PersID</th>
<th>Name</th>
<th>Position</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>2125</td>
<td>Sokrates</td>
<td>C4</td>
<td>226</td>
</tr>
<tr>
<td>2126</td>
<td>Russel</td>
<td>C4</td>
<td>232</td>
</tr>
<tr>
<td>2127</td>
<td>Kopernikus</td>
<td>C3</td>
<td>310</td>
</tr>
<tr>
<td>2133</td>
<td>Popper</td>
<td>C3</td>
<td>52</td>
</tr>
<tr>
<td>2134</td>
<td>Augustinus</td>
<td>C3</td>
<td>309</td>
</tr>
<tr>
<td>2136</td>
<td>Curie</td>
<td>C4</td>
<td>36</td>
</tr>
<tr>
<td>2137</td>
<td>Kant</td>
<td>C4</td>
<td>7</td>
</tr>
</tbody>
</table>
1st problem: How to find those students who attend all lectures in this semester? Which student under those who attend all the lectures in this semester has the smallest student ID?

• Which relations are needed for this subquery?
  - attended_by: \{\text{StudID: integer, LeclID: integer}\}
  - Lectures: \{\text{LeclID: integer, Title: string, Credits: integer, Held_by:integer}\}

• Which relational algebra operators are needed?
  - Division \(\div\) (universal quantifier),
  - Projection \(\pi\) (to find all LeclIDs)

\[
\text{omnistud} = \text{attended\_by} \div \pi_{\text{LeclID}}(\text{Lectures})
\]
query: Which student under those who attend all the lectures in this semester has the smallest student ID?

2nd problem: How to find now the student with the smallest ID?

- Join the omnistuds relation with itself:
  \[ R_2 = \text{Omnistuds} \bowtie \rho_{\text{StudID} \leftarrow \text{ID}_1} \text{(Omnistuds)} \quad \text{StudID > ID}_1 \]

- Determine now the StudID for which no join partner (with StudID > ID1) could be found:
  \[ \pi_{\text{StudID}}(\text{Omnistuds}) - \pi_{\text{StudID}}(R_2) \]

In this way, the \text{min-} and \text{max}-function can be \textit{simulated} but not the aggregate functions sum, avg, etc...!
query: Which student under those who attend all the lectures in this semester has the smallest student ID?

last step: put all sub-expressions together for getting the entire query:

\[
\left(\pi_{\text{StudID}}\left(\frac{\text{attended_by}}{\pi_{\text{LecID}}(\text{Lectures})}\right) \ominus \pi_{\text{StudID}}\left(\frac{\text{attended_by}}{\pi_{\text{LecID}}(\text{Lectures})}\right)\right) \bowtie \rho_{\text{StudID} \leftarrow \text{ID1}}(\frac{\text{attended_by}}{\pi_{\text{LecID}}(\text{Lectures})}) \]

How to evaluate such a complex relational algebra query? 
⇒ first the most inner expression!

Potential for further optimization by evaluating common sub-expressions only once!
1) Determination of the auxiliary relation Omnistuds:

\[ \pi_{\text{LecID}}(\text{Lectures}) \]

<table>
<thead>
<tr>
<th>LecID</th>
<th>Title</th>
<th>Credits</th>
<th>Held_by</th>
</tr>
</thead>
<tbody>
<tr>
<td>4001</td>
<td>logic</td>
<td>4</td>
<td>2125</td>
</tr>
<tr>
<td>4002</td>
<td>knowledge theory</td>
<td>3</td>
<td>2126</td>
</tr>
<tr>
<td>4003</td>
<td>database systems</td>
<td>4</td>
<td>2137</td>
</tr>
</tbody>
</table>

all auxiliary relations as intermediate results have no relation name anymore
2) Determination of the auxiliary relation Omnistuds:

attended_by $\div \pi_{\text{LecID}}(\text{Lectures})$

auxiliary relation which we have called Omnistuds in our example
3) Determination of the relation with the smallest student ID:

Since the StudID 28106 is the smallest one, it does not satisfy the join condition and no 2-tuples with 28106 as value in the first row are in the resulting relation.
4) Determination of the relation with the smallest student ID:

\[ \pi_{\text{StudID}}(R2) \]

<table>
<thead>
<tr>
<th>StudID</th>
<th>ID1</th>
</tr>
</thead>
<tbody>
<tr>
<td>29120</td>
<td>28106</td>
</tr>
<tr>
<td>30112</td>
<td>28106</td>
</tr>
<tr>
<td>30112</td>
<td>29120</td>
</tr>
</tbody>
</table>

Elimination of duplicates is necessary!
5) Determination of the relation with the smallest student ID:

\[ \pi_{\text{StudID}}(\text{Omnistuds}) - \pi_{\text{StudID}}(R2) \]

Projection redundant!

\[
\begin{array}{c}
\text{StudID} \\
28106 \\
29120 \\
30112 \\
\end{array}
\]

\[
\begin{array}{c}
\text{StudID} \\
29120 \\
30112 \\
\end{array}
\]

\[
\begin{array}{c}
\text{StudID} \\
28106 \\
\end{array}
\]

our result for the initial query!
Foundations of Information Systems

2 SQL - The Database Language
SQL – Foundations
**SQL (Structured Query Language)**
- Descriptive Query Language

**Consists of**
- DQL (*data query language*)
- DML (*data manipulation language*)
- DDL (*data definition language*):
- DCL (*data control language*): access right management

**SQL Standard**
- Part 1: SQL/Framework (Standard Description)
- **Part 2: SQL/Foundation**
- Part 3: SQL/CLI (*call level interface*)
- Part 4: SQL/PSM (*persistent storage modules*)
- ...
SELECT Statement

SELECT [DISTINCT] <attribute-list> FROM <table-name> [AS <alias>], …
WHERE <predicate-list>
GROUP BY <attribute-list>
HAVING <predicate-list>
ORDER BY <attribute> [ASC|DESC],… → Projection, Duplicate elim.
→ Cartesian Product
→ Selection on tuple level
→ Grouping
→ Selection on group level
→ Sorting

Select Conditions Examples

- Patter search in Strings
  WHERE R_NAME [NOT] LIKE 'A%'
- Range search
  WHERE R_REGIONKEY [NOT] BETWEEN 1 AND 3
- NULL-Werte
  WHERE R_COMMENT IS [NOT] NULL
Example (SAMPLE)

- Name and salary of employees belonging to department A00

```
SELECT FIRSTNME, LASTNAME, WORKDEPT, SALARY
FROM EMPLOYEE
WHERE WORKDEPT = 'A00'
```

- Result

<table>
<thead>
<tr>
<th>FIRSTNME</th>
<th>LASTNAME</th>
<th>WORKDEPT</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHRISTINE</td>
<td>HAAS</td>
<td>A00</td>
<td>52750</td>
</tr>
<tr>
<td>VINCENZO</td>
<td>LUCCHESI</td>
<td>A00</td>
<td>46500</td>
</tr>
<tr>
<td>SEAN</td>
<td>O'CONNELL</td>
<td>A00</td>
<td>29250</td>
</tr>
</tbody>
</table>
**Example (SAMPLE)**

- Average salary per department

```sql
SELECT DEPTNAME, AVG(SALARY) AS AVG_SALARY
FROM DEPARTMENT D, EMPLOYEE E
WHERE E.WORKDEPT = D.DEPTNO
GROUP BY DEPTNAME
```

- Result

<table>
<thead>
<tr>
<th>DEPTNAME</th>
<th>AVG_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADMINISTRATION SYSTEMS</td>
<td>25153.33</td>
</tr>
<tr>
<td>INFORMATION CENTER</td>
<td>30156.66</td>
</tr>
<tr>
<td>MANUFACTURING SYSTEMS</td>
<td>24677.77</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>20998</td>
</tr>
<tr>
<td>PLANNING</td>
<td>41250</td>
</tr>
<tr>
<td>SOFTWARE SUPPORT</td>
<td>23827.5</td>
</tr>
<tr>
<td>SPIFFY COMPUTER SERVICE DIV.</td>
<td>42833.33</td>
</tr>
<tr>
<td>SUPPORT SERVICES</td>
<td>40175</td>
</tr>
</tbody>
</table>

aggregation function (e.g. COUNT, SUM, MIN, MAX)
Example (TPCH)

- Total revenue per country, if this value is over 50 Mio.; sorted

```
SELECT N_NAME, SUM(O_TOTALPRICE) AS TURNOVER
FROM ORDERS, CUSTOMER, NATION
WHERE O_CUSTKEY = C_CUSTKEY
  AND C_NATIONKEY = N_NATIONKEY
GROUP BY N_NAME
HAVING SUM(O_TOTALPRICE) > 50000000
ORDER BY N_NAME
```

- Result

<table>
<thead>
<tr>
<th>N_NAME</th>
<th>TURNOVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARGENTINA</td>
<td>12,999,7977,11</td>
</tr>
<tr>
<td>EGYPT</td>
<td>6,648,2178,24</td>
</tr>
<tr>
<td>JORDAN</td>
<td>2,739,41626,19</td>
</tr>
<tr>
<td>MOROCCO</td>
<td>10,937,39712,6</td>
</tr>
</tbody>
</table>
A Systematic Approach to SQL
Queries and Updates in SQL: Overview

- SQL data manipulation language: statements for "manipulating" data

- two forms of manipulation:
  - formulation and evaluation of queries
  - execution of updates

- The format of simple queries is as follows:
  SELECT-FROM-WHERE

- But the SQL query language (as part of the SQL-DML) can do much more!

- goal of this section: introduction to the foundations of this powerful language

- You can become an expert in SQL by much more intense training and self-studies only!

- only at the end of this section: treatment of update statements in SQL (INSERT, DELETE, UPDATE etc.)
In SQL, there are two types of queries:

- **Table expression**
- **Conditional expression**

**Problem (?)**: Only table expressions can be directly posed as queries by the user!
Table Expressions: Basic Structure

- Basic component of any SQL query: SELECT-FROM-WHERE blocks

- Syntactic structure in the simplest case:

  ```sql
  SELECT (list_of_column_names) 
  FROM (list_of_table_names) 
  WHERE condition
  ```

- Example:

  ```sql
  SELECT Name, Inhabitants 
  FROM city, country 
  WHERE Inhabitants >= 1000 AND Name=Capital ;
  ```

- In SQL, upper or lower case does not matter for table and column names.
Meaning of SELECT Blocks: Principle

FROM
product of all input tables

WHERE
choice of certain rows

SELECT
choice of certain columns

result table

derived tables
Reminder: SQL and Relational Algebra

- Already mentioned: 'SELECT-FROM-WHERE' blocks are "syntactic sugar" for the most common kind of RA expression composed of projection, selection and product:

\[
\pi_{\text{Capital, Inhabitants}} \left( \sigma_{\text{Inhabitants} \geq 1000 \land \text{Name} = \text{Capital}} \left( \text{city} \times \text{country} \right) \right)
\]

- As each join is in fact an abbreviation of a particular project-select-product expression, the effect of each join can be obtained by means of a single SELECT-FROM-WHERE block in SQL, even though in a somewhat more tedious way.
The **WHERE part** of an SFW-block is – in its basic form – nothing but a selection condition composed of individual comparisons of column values of the tables mentioned in FROM with other column values or constants.

- **Comparisons** make use of the following six comparison operators:
  - $=$
  - $\neq$
  - $<$
  - $>$
  - $\leq$
  - $\geq$

- Comparisons can be logically combined by the three basic **junctors** of propositional logic, written in keyword notation:
  - AND
  - OR
  - NOT

- Arbitrary nesting is possible (using brackets). There are more complex conditions which we will introduce later.
Implicit and Explicit Variables

- SQL offers various **shorthand notations** for tuple variables, aimed at making queries more readable (and more similar to natural language style).

- Instead of explicitly introducing variables for each relation (via AS), **table names** themselves may be used as variables (as long as each relation appears only once in the query):

```sql
SELECT country.Capital, city.Inhabitants
FROM city, country
WHERE city.Inhabitants >= 1000 AND city.Name = country.Capital;
```

- Both styles, separate variables and table names as pseudo-variables, can be **mixed** within a query:

```sql
SELECT C.Capital, city.Inhabitants
FROM city, country AS C
WHERE city.Inhabitants >= 1000 AND city.Name = C.Capital;
```
The most „economic“ style of writing down a query in SQL does not make use of variables at all – at least not explicitly:

```sql
SELECT Capital, Inhabitants
FROM city, country
WHERE Inhabitants >= 1000 AND Name = Capital;
```

However, this „variable-free“ style always comes along with an implicit assumption about bindings of column names to table names as in the formulation with variables. For „decoding“ this implicit style, the resp. DB schema has to be consulted:

```sql
SELECT Capital, Inhabitants
FROM city, country
WHERE Inhabitants >= 1000 AND Name = Capital;
```
Implicit and Explicit Variables (3)

- Using variables is *unnecessary* in most cases – however, you are advised to use them (despite the extra effort) whenever the implicit bindings of columns to tables are confusing for you (or others, reading your query).

- As soon as a table is mentioned more than once in a query, however, it is unavoidable to use variables in order to resolve possible ambiguities, e.g.:

```
SELECT X.From, Y.To
FROM    link AS X, link AS Y
WHERE   X.To = Y.From
```

  referring to a table called 'link' with columns (From, To)

- Without the variables it would be unclear, which of the two adjoining links is actually meant, as they have identical column names. Using the variables as prefix to the columns (as in TRC) resembles column renaming in RA.