1 Relational Databases
2 Relational Algebra and Calculus
3 Introduction to SQL
Terminological confusion

- database
- information system
- database management system
- database model
- data bank
- data base
- database system
- data model
Information system: computer science view

- external media of communication
- application specific methods

database system

information system
example: Geo Information System (GIS)
satellite navigation system (GPS)
database system
virtual map interface
- Every information system uses a database system to manage its data.

- Thus, an information system is ‘more’ than a database system while the later has to fulfill the following main tasks:

  - schema management
  - query processing
  - transaction management
    - access protection
    - integrity control
    - multi-user synchronisation
    - recovery from errors
  - storage management
database system: notion

users and application programs

DBMS: database management system
A helpful „formula“ for remembering the principle idea

„informal formula“:

\[
\text{DBS} = \text{DBMS} + n \times \text{DB}
\]
Thus, the main tasks of a DBS …

- schema management
- query processing
- transaction management
  - access protection
  - integrity control
  - multi-user synchronization
  - recovery from errors
- storage management

… are realized by its DBMS.
• **information system** = database system +
  external media of communication +
  application specific methods

• **database system** DBS = DBMS + n* DB

• **database** DB = a set of data stored according to the concepts of the data model
  supported by the DBMS

• **database management system** DBMS = a system for controlling the access (reading and
  writing access) to the databases

• **data model** = a collection of concepts that determine how a database is structured and
  can be used (e.g., a relational model allows to structure data in "tables").
  The DBMS is responsible for the user being able to see data in structured
  form while the physical representation (in a file) remains hidden.
The DB-market is completely dominated by systems supporting the relational data model today.

Leading (commercial) manufacturers of relational DB-products:

- Oracle
- Sybase
- Microsoft (Access, SQL Server)
- Postgres (Freeware)
- IBM (DB2, Informix)
- MySQL (Freeware)

The notion "relational" is motivated by the mathematical concept of a relation. Relations in mathematics are sets of tuples.

Relational databases are collections of one or more relations.

In practice, relations can be visualized as tables, the rows of which are individual records of data with the same (homogeneous) field structure.

In science, relational databases have a broad range of theoretical foundations.
The idea to organize data in tables is quite old and pretty obvious.

The idea to investigate this representation of data by means of the theory of relations is due to one man, who proposed this view at the end of the 1960s:

Edgar F. Codd

In 1970, he published his seminal paper "A Relational Model of Data for Large Shared Data Banks", in which he fixed all foundations of relational databases with amazing precision and clarity.

Codd died in early 2003.

For this pioneering work Codd received the Turing Award in 1982, the "Nobel price of informatics".
Europe.mdb is a small database for MS Access.

Just now it contains solely two tables: countries and cities in Europe.
Relational **tables are grids**, the fields of which are consisting of columns and rows. (There is a specific terminology for such tables in MS Access.)
Unfortunately, the basic concepts of the relational model are denoted by different terms depending on the context. There are synonymous, but different terminologies in database theory, the standard DB language SQL and MS Access:

<table>
<thead>
<tr>
<th>theory</th>
<th>SQL</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>relation</td>
<td>table</td>
<td>datasheet</td>
</tr>
<tr>
<td>tuple</td>
<td>row</td>
<td>record</td>
</tr>
<tr>
<td>attribute</td>
<td>column</td>
<td>field name</td>
</tr>
<tr>
<td>domain</td>
<td>data type</td>
<td>field data type</td>
</tr>
</tbody>
</table>

Be warned of this „Babylonic confusion“ of terms – we urgently recommend that you always stick to a single system of notions in a consistent manner. It doesn‘t matter which system you use – but never mix them up!
Access table	 „countries“:

datasheet view
Access table: design view

Access table „countries“:

design view

- field size
- format
- input mask
- caption
- default value
- validation rule
- validation text
- Required
- Allow zero length
- Indexed
- Unicode compression
The two different "views" of a table in Access correspond to two fundamental notions of relational databases:

- Schema of a relation: definition of name and structure of the relation
- State of a relation: all tuples currently contained in the relation
- The structure of each state of a relation is defined by its schema. (States are called instances of the schema.)
- In general, the schema remains fixed during state changes.
- Sometimes, however, there are schema modifications as well, followed by immediate state adaptations: schema evolution
- Plural of schema: schemas (not "schemes")!
Schemas and states

- \( \text{schema}_1 \)
- \( \text{schema}_2 \)
- States
- \( \text{instances of schema}_1 \)
- \( \text{instances of schema}_2 \)
- \( \text{current state} \)
There are two basic forms of interaction with a database:

**Reading access:** query mode

**Writing access:** update mode
• A fundamental characteristic of each database management system is the support of one or more query language.

• A query is an expression in this language which . . .
  - . . . is able to express arbitrarily complex search criteria.
  - . . . refers to one or more tables simultaneously.
  - . . . returns one or more records or simply yes/no as an answer.
  - . . . returns records in form of answer tables.

• Every commercial DBMS understands the textual "Structured Query Language" (SQL).

• SQL is the most widely distributed query language for relational DBs.

• SQL is standardized by the American National Standards Institute (ANSI).
Retrieve name, capital and area of all countries larger than 100000 km² in descending order of size!

SQL

```
SELECT countries.country, countries.capital, countries.area
FROM countries
WHERE ((countries.area) > 100000))
ORDER BY countries.area DESC;
```
Retrieve name, capital and area of all countries larger than 100000 km² in descending order of size!

Answers to relational queries are always returned as tables, too.

Thus, they may be „reused“ as input for further queries.

However, these tables are not stored in the DB! They are „virtual“ tables recomputed each time the query is asked.
Interaction with databases: principle

There are two basic forms of interaction with a database:

**Reading access:** query mode

**Writing access:** update mode
State changes: general principles

- "Write" access to a database . . .
  - . . . always results in a state change of the DB.
  - . . . always takes place under control of the DBMS.

- There are three basic forms of write access:
  - insertions of new records into a table
  - deletions of existing records from a table
  - modifications of the value of a particular field in a record of a table

- Insertions and modifications are accepted by the DBMS only if the data types of the resp. fields declared in the schema of the table fit with the values in the new/modified records.

- Caution! The English notion "update" is used in this context with two different meanings – be sure you understand which of them is actually meant:
  - as a synonym for modification
  - as a generalization comprising all three kinds of write access

- Update statements can be expressed in SQL, too.
Some examples of how to formulate update statements in SQL:

- Insert statement with direct reference to the rows to be inserted:

  ```sql
  INSERT INTO cities (City, Country, Population, Year)
  VALUES ('Bonn', 'D', 317000, 2008);
  ```

- Insert Statement with indirect identification of the rows to be inserted:

  ```sql
  INSERT INTO cities
  SELECT capital, population, year
  FROM countries
  WHERE population >= 120000;
  ```

- Update statement with

  ```sql
  UPDATE countries
  SET capital='Bonn'
  WHERE Code='D';
  ```
Integrity constraints and integrity checking

• Primary key definitions and validation rules are special examples of a very important general concept in database design:

  integrity constraints

• In general, an integrity constraint (constraint for short) is a logical condition to be satisfied by each state of the database at all times, i.e., integrity constraints are required to be invariably true during the lifetime of the database.

• In SQL, we will find a rather powerful language for expressing nearly arbitrary such conditions.

• Integrity constraint violations – likely to happen during DB modifications – are controlled automatically by the DBMS. Each insertion, deletion or update of a table is checked for possibly violating any constraint prior to the execution of the resp. modification:

  integrity checking

• If integrity violations are detected, the DBMS either refuses to perform the desired modification or „repairs“ the semantic mistake causing the violation automatically, if possible.

• Key and validation rule violations cannot be „repaired“!
In SQL, most integrity constraints are defined within a `CREATE TABLE`-command:

```
CREATE TABLE Countries
(
    County text,
    Code char,
    Capital text,
    Area number(15) DEFAULT NULL CHECK (> = 0 OR IS NULL),
    Population number(15) NOT NULL CHECK (> = 0 OR IS NULL),
    YEAR date,
    PRIMARY KEY (Country),
    FOREIGN KEY (Capital) REFERENCES Cities
) ;
```
Summary of the notions/concepts you should know:

<table>
<thead>
<tr>
<th>data model</th>
<th>DB query</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB schema</td>
<td>query language</td>
</tr>
<tr>
<td>DB state</td>
<td>subquery</td>
</tr>
<tr>
<td>relation (table,datasheet)</td>
<td>integrity constraint</td>
</tr>
<tr>
<td>attribute (column,field)</td>
<td>check constraint</td>
</tr>
<tr>
<td>tuple (row,record)</td>
<td>primary key</td>
</tr>
<tr>
<td>domain/(field) data type</td>
<td>foreign key</td>
</tr>
<tr>
<td>null value</td>
<td>referential integrity</td>
</tr>
<tr>
<td>default value</td>
<td></td>
</tr>
<tr>
<td>relationship</td>
<td></td>
</tr>
</tbody>
</table>
SQL is based on the languages RA and TRC which are essential for understanding the Semantics of SQL expressions:

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

RA

```
\pi ( \sigma \text{Inhabitants} \geq 1000 \land \text{Name} = \text{Capital} \\
\text{Capital, Inhabitants} ) (~\text{city} \times \text{country} )
```

TRC

```
\{ [y.\text{Capital}, x.\text{Inhabitants}] | \\
city(x) \land \text{country}(y) \land \\
x.\text{Inhabitants} \geq 1000 \land x.\text{Name} = y.\text{Capital} \}
```
1 Relational Databases
2 **Relational Algebra and Calculus**
3 Introduction to SQL
Why additional languages?

- Query languages for databases, such as SQL and Datalog – to be introduced in next chapters – are formal languages, relying on a rigorously defined syntax and semantics.

- Furthermore, we will introduce two variants of logics and set theory, resp., tailored particularly for the manipulation of relations, being special sets and thus requiring special operators and special syntax:
  - **Relational algebra** is the basis of relational query processing.
  - **Relational calculus** is the logical counterpart.

- SQL is based on both, relational algebra as well as relational calculus.

- Datalog is purely based on relational calculus (domain calculus).
Relational Algebra
The (mathematical) concept of a set is of fundamental importance for almost every area of computer science.

"A set is a collection into a whole of definite, distinct objects of our perception or our thought."

Georg Cantor (1845-1918), originator of set theory

The order and number of occurrences of a member in a set is irrelevant (in contrast to SQL):

\{2, 4, 6, 8\}
\{6, 4, 8, 2\}
\{2, 2, 4, 8, 8, 8, 6\}

three denotations of the same set
Elementary set operators

- In set theory, there are three basic operations by which two sets can be combined:

\[
\begin{align*}
\text{union} & \quad A \cup B = \{ e \mid e \in A \text{ or } e \in B \} \\
\text{intersection} & \quad A \cap B = \{ e \mid e \in A \text{ and } e \in B \} \\
\text{difference} & \quad A \setminus B = \{ e \mid e \in A \text{ and } e \not\in B \}
\end{align*}
\]

- Examples:

\[
\begin{align*}
\{1, 2\} \cup \{2, 3\} & = \{1, 2, 3\} \\
\{1, 2\} \cap \{2, 3\} & = \{2\} \\
\{1, 2\} \setminus \{2, 3\} & = \{1\}
\end{align*}
\]
• more binary basic operations of set theory:

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

(Cartesian) product

• generalized construction of products for \( n \) sets (\( n \geq 2 \)):

\[
A_1 \times \ldots \times A_n = \{ (a_1, \ldots, a_n) | a_i \in A_i \}
\]

• The members of a product of \( n \) sets are called \((n)\)-tuples.

• special denominations for tuples:
  • \( n = 1 \): singleton
  • \( n = 2 \): pair
  • \( n = 3 \): triple
  • \( n = 4 \): quadruple
  • \( n = 5 \): quintuple
Example of a product of two sets

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

- A:
  - (●, 1)
  - (●, 5)
  - (●, 7)

- B:
  - (●, 1)
  - (●, 5)
  - (●, 7)

- A × B:
  - (●, 1)
  - (●, 5)
  - (●, 7)
Relations

- In computer science, **sets of tuples** are very important for modeling relationships between objects.

- Sets of tuples are called **relations**.

  Every subset $R$ of a product $A_1 \times \ldots \times A_n$ is called a **relation** over $A_1, \ldots, A_n$.

- Relations are usually denoted in the form of a **table**.

  $A_1 = \{ \text{a, b, c} \}$  
  $A_2 = \{ 1, 2 \}$  
  $A_3 = \{ \%, $ \}$

  $R$

  \[
  \begin{array}{c|cc}
  a & 1 & \% \\
  a & 1 & $ \\
  b & 2 & $
  \end{array}
  \]
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").
What is an 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.
• Relations are (special) sets, and thus operators of set algebra are applicable to relations, too:

<table>
<thead>
<tr>
<th>Set Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>( R \cup S )</td>
</tr>
<tr>
<td>difference</td>
<td>( R - S )</td>
</tr>
<tr>
<td>intersection</td>
<td>( R \cap S )</td>
</tr>
<tr>
<td>product</td>
<td>( R \times S )</td>
</tr>
</tbody>
</table>

- 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)

Union of two inhomogeneous relations . . .

\[ R \cup S \]

. . . results in a set, . . .

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

. . . but not in a relation!
• 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 \textbf{union compatible}.

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

• In addition, \textit{identity of names} of all columns in both relations is often required.

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

\[
\rho_{A \leftarrow B}(R) \quad \text{[In relation R, column A is renamed into B.]} 
\]
In set theory, the **product** of two relations is *always* a binary relation, the elements of which are pairs of tuples.

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

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:

\[
(a,b) \rightarrow (a,b,1,2,3)
\]

\[
(1,2,3) \rightarrow (a,b,1,2,3)
\]

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: \( \pi \)
  - ... certain rows (tuples): \( \sigma \)

- 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 formalising 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!).

\[
\pi_{A,B}
\]

\[
\begin{array}{ccc}
A & B & C \\
1 & 2 & 3 \\
2 & 3 & 4 \\
1 & 2 & 5 \\
\end{array}
\]

\[
\begin{array}{cc}
A & B \\
1 & 2 \\
2 & 3 \\
\end{array}
\]

• 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 (1)

- 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 \( ( \land, \lor, \neg ) \).
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'!

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>x</td>
<td>13</td>
</tr>
</tbody>
</table>

Formulation of this query in relational algebra:

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

```
<table>
<thead>
<tr>
<th>MatrNr</th>
<th>Name</th>
<th>Semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>26120</td>
<td>John</td>
<td>10</td>
</tr>
<tr>
<td>27550</td>
<td>Eve</td>
<td>12</td>
</tr>
<tr>
<td>28117</td>
<td>Bill</td>
<td>27</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>MatrNr</th>
<th>CourseNr</th>
</tr>
</thead>
<tbody>
<tr>
<td>26120</td>
<td>5001</td>
</tr>
<tr>
<td>27550</td>
<td>5001</td>
</tr>
<tr>
<td>27550</td>
<td>4052</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>MatrNr</th>
<th>Name</th>
<th>Semester</th>
<th>CourseNr</th>
</tr>
</thead>
<tbody>
<tr>
<td>26120</td>
<td>John</td>
<td>10</td>
<td>5001</td>
</tr>
<tr>
<td>27550</td>
<td>Eve</td>
<td>12</td>
<td>5001</td>
</tr>
<tr>
<td>27550</td>
<td>Eve</td>
<td>12</td>
<td>4052</td>
</tr>
</tbody>
</table>
```
The natural join is a derived operator in RA as its effect could as well be reached by combining projection, selection and product:

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

<table>
<thead>
<tr>
<th>attr(R) – attr(S)</th>
<th>attr(R) \cap attr(S)</th>
<th>attr(S) – attr(R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1 \ldots A_m</td>
<td>B_1 \ldots B_k</td>
<td>C_1 \ldots C_n</td>
</tr>
</tbody>
</table>

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: \( R \bowtie_{\Theta} S \) mit \( \Theta = (R.A \leq S.C \land S.B > 0) \)

\[
\begin{array}{c|c}
R & S \\
\hline
A & B & B & C \\
1 & a & 5 & 2 \\
3 & b & 0 & 1 \\
2 & a & 1 & 1 \\
\end{array}
\]

- 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: \( R \bowtie S \) mit \( \Theta = (R.A \leq S.C \land S.B > 0) \)

\[
\begin{array}{|c|c|} \hline
R & S \\
\hline
| A | B | & | B | C | \\
\hline
1 & a & | & 5 & 2 | \\
3 & b & | & 0 & 1 | \\
2 & a & | & 1 & 1 | \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|} \hline
A & R.B & S.B & C \\
\hline
1 & a & 5 & 2 \\
1 & a & 1 & 1 \\
2 & a & 5 & 2 \\
3 & b & Null & Null \\
Null & Null & 0 & 1 \\
\hline
\end{array}
\]

- 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: \( \Leftarrow \) resp. \( \Rightarrow \)
• Most sophisticated, but also quite useful operator of RA: division

• formal notation like in arithmetics: \( R \div S \)

• 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?

\[
\begin{array}{|c|c|}
\hline
R & S \\
\hline
A & B \\
\hline
a & 1 \\
a & 2 \\
a & 3 \\
b & 2 \\
b & 3 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
B \\
\hline
1 \\
2 \\
\hline
\end{array}
\]

\[
R \div S = \pi_{\text{attr}(R)} \setminus \text{attr}(S) (R) \setminus \pi_{\text{attr}(R)} \setminus \text{attr}(S) ((\pi_{\text{attr}(R)} \setminus \text{attr}(S) (R) \times S) \setminus R)
\]

Only 'a' appears in R combined with all S-tuples!
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.
The schema of a university database:

**Lectures:**  
{[LecID: integer, Title: string, Credits: integer, Held_By:integer]}

**Professors:**  
{[PersID: integer, Name: string, Position: string, Room: integer]}

**Assistents:**  
{[PersID: integer, Name: string, Research: string, ProfID: integer]}

**Students:**  
{[StudID: integer, Name: string, Semester: integer]}

**attended_by:**  
{[StudID: integer, LecID: integer]}

**PreLecture:**  
{[PredID: integer, SuccID: integer]}
### Schema State

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

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

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

<table>
<thead>
<tr>
<th>Professors</th>
<th>PersID</th>
<th>Name</th>
<th>Position</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2125</td>
<td>Sokrates</td>
<td>C4</td>
<td>226</td>
</tr>
<tr>
<td></td>
<td>2126</td>
<td>Russel</td>
<td>C4</td>
<td>232</td>
</tr>
<tr>
<td></td>
<td>2127</td>
<td>Kopernikus</td>
<td>C3</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>2133</td>
<td>Popper</td>
<td>C3</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>2134</td>
<td>Augustinus</td>
<td>C3</td>
<td>309</td>
</tr>
<tr>
<td></td>
<td>2136</td>
<td>Curie</td>
<td>C4</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>2137</td>
<td>Kant</td>
<td>C4</td>
<td>7</td>
</tr>
</tbody>
</table>
Possible queries:

1. Which students have been studying for more than 9 semesters?
2. What position may be occupied by a professor?
3. Which professors and assistants exist in the database?
4. Which student under those who attend all the lectures in this semester has the smallest student ID?
5. ....
Relational algebra: Example

Which students have already studied more than 9 semesters?

\[ \sigma_{\text{Semester} > 9} (\text{Students}) \]

<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>
</tbody>
</table>
What position may be occupied by a professor?

\[ \pi_{\text{Position}} (\text{Professors}) \]

<table>
<thead>
<tr>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
</tr>
<tr>
<td>C4</td>
</tr>
</tbody>
</table>

Duplicates are eliminated!
Which professors and assistants exists in the database?

\[ \pi_{\text{Name}}(\text{Professors}) \cup \pi_{\text{Name}}(\text{Assistants}) \]

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sokrates</td>
</tr>
<tr>
<td>Russel</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Kant</td>
</tr>
<tr>
<td>Platon</td>
</tr>
<tr>
<td>Aristoteles</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Spinoza</td>
</tr>
</tbody>
</table>
Relational algebra: Example

What lectures are attended by which students?

<table>
<thead>
<tr>
<th>Students</th>
<th>attended_by</th>
</tr>
</thead>
<tbody>
<tr>
<td>StudID</td>
<td>StudID</td>
</tr>
<tr>
<td>Name</td>
<td>LecID</td>
</tr>
<tr>
<td>Semester</td>
<td>Credits</td>
</tr>
<tr>
<td>24002</td>
<td>26120</td>
</tr>
<tr>
<td>Xenokrates</td>
<td>27550</td>
</tr>
<tr>
<td>25403</td>
<td>...</td>
</tr>
<tr>
<td>Jonas</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>LecID</td>
</tr>
<tr>
<td>4001</td>
</tr>
<tr>
<td>4002</td>
</tr>
<tr>
<td>4003</td>
</tr>
</tbody>
</table>
What lectures are attended by which students?

<table>
<thead>
<tr>
<th>StudID</th>
<th>Name</th>
<th>Semester</th>
<th>LecID</th>
<th>Title</th>
<th>Credits</th>
<th>Held_By</th>
</tr>
</thead>
<tbody>
<tr>
<td>27550</td>
<td>Schopenhauer</td>
<td>6</td>
<td>4001</td>
<td>logic</td>
<td>4</td>
<td>2125</td>
</tr>
<tr>
<td>27550</td>
<td>Schopenhauer</td>
<td>6</td>
<td>4002</td>
<td>knowledge theory</td>
<td>3</td>
<td>2126</td>
</tr>
<tr>
<td>28106</td>
<td>Carnap</td>
<td>3</td>
<td>4001</td>
<td>logic</td>
<td>4</td>
<td>2125</td>
</tr>
<tr>
<td>28106</td>
<td>Carnap</td>
<td>3</td>
<td>4002</td>
<td>knowledge theory</td>
<td>3</td>
<td>2126</td>
</tr>
<tr>
<td>28106</td>
<td>Carnap</td>
<td>3</td>
<td>4003</td>
<td>database systems</td>
<td>4</td>
<td>2137</td>
</tr>
<tr>
<td>29120</td>
<td>Theophrastos</td>
<td>2</td>
<td>4001</td>
<td>logic</td>
<td>4</td>
<td>2125</td>
</tr>
<tr>
<td>29120</td>
<td>Theophrastos</td>
<td>2</td>
<td>4002</td>
<td>knowledge theory</td>
<td>3</td>
<td>2126</td>
</tr>
<tr>
<td>29120</td>
<td>Theophrastos</td>
<td>2</td>
<td>4003</td>
<td>database systems</td>
<td>4</td>
<td>2137</td>
</tr>
<tr>
<td>29555</td>
<td>Feuerbach</td>
<td>2</td>
<td>4003</td>
<td>database systems</td>
<td>4</td>
<td>2137</td>
</tr>
</tbody>
</table>
What is the difference to the result of the following expression?

\[ \sigma_{\text{Students.StudID} = \text{attended_by.StudID} \land \text{attended_by.LecID} = \text{Lectures.LecID}} (\text{Students} \times \text{attended_by} \times \text{Lectures}) \]
Which student under those who attend all the lectures in this semester has the smallest student ID?

1st problem: How to find those students who attend all lectures?

- Which relations are needed for this subquery?
  \[
  \Rightarrow \text{attended_by: } \{[\text{StudID: integer, LecID: integer}]\}
  \]
  \[
  \Rightarrow \text{Lectures: } \{[\text{LecID: integer, Title: string, Credits: integer, Held_by:integer}]\}
  \]

- Which relational algebra operators are needed?
  \[
  \Rightarrow \text{Division } \div \text{ (universal quantifier) ,}
  \]
  \[
  \Rightarrow \text{Projection } \pi \text{ (to find all LecIDs)}
  \]

\[
\text{omnistud = attended\_by } \div \pi_{\text{LecID}}(\text{Lectures})
\]
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:

\[ R_2 = \text{Omnistuds} \bowtie \rho_{\text{StudID} > \text{ID}_1} (\text{Omnistuds}) \]

• Determine now the StudID for which no join partner (with MatrNr > Nr1) could be found:

\[ \pi_{\text{StudID}}(\text{Omnistuds}) - \pi_{\text{StudID}}(R_2) \]

In this way, the min- and max-function can be simulated but not the aggregate functions sum, avg, etc…!
last step: put all sub-expressions together for getting the entire query:

\[
\pi_{\text{StudID}}(\text{attended}\_\text{by} \div \pi_{\text{LecID}}(\text{Lectures})) - \\
\pi_{\text{StudID}}((\text{attended}\_\text{by} \div \pi_{\text{LecID}}(\text{Lectures})) \bowtie \\
\rho_{\text{StudID} < \text{ID1}}(\text{attended}\_\text{by} \div \pi_{\text{LecID}}(\text{Lectures})))
\]

Potential for further optimization by evaluating common subexpressions only once!

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

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

<table>
<thead>
<tr>
<th>attended_by</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>StudID</td>
<td>LecID</td>
</tr>
<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>

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

<table>
<thead>
<tr>
<th>LecID</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4001</td>
<td></td>
</tr>
<tr>
<td>4002</td>
<td></td>
</tr>
<tr>
<td>4003</td>
<td></td>
</tr>
</tbody>
</table>

\[ \vdots \]

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

\[
\text{Omnistuds} \bowtie \rho_{\text{StudID} \leftarrow \text{ID1}}(\text{Omnistuds})
\]

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:

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

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 is redundant!

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

\[ \ldots \div \ldots \]

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

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

\[ \ldots \]

\[ \ldots \]  

\[ \ldots \]

our result for the initial query!
1.1 Relational Databases
1.2 Relational Algebra and Calculus
1.3 Introduction to SQL
SQL: History

- SQL (Structured Query Language) is the most popular and well-known relational DB language today.
- Almost every relational DBMS „understands" SQL!
- SQL has been developed in the early 1970s at IBM (as interface to the relational prototype DBMS "System R").
- Original name: SEQUEL (Structured English Query Language)
- First SQL standard: SQL1 in 1986 by ANSI in the USA, revised in 1989
- Considerable extensions in 1992: SQL2 or SQL92, resp.
- New standard in 2008: (only partially realized in commercial systems till now)
- Attention! Nearly every commercial DB product has its own „dialect" of SQL, more or less compatible with the standard.
good new book about the new SQL standard:

Basics of SQL

- SQL has its own terminology of relational concepts:

<table>
<thead>
<tr>
<th>RA</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>relation</td>
<td>table</td>
</tr>
<tr>
<td>attribute</td>
<td>column</td>
</tr>
<tr>
<td>tuple</td>
<td>row</td>
</tr>
<tr>
<td>attribute type</td>
<td>domain</td>
</tr>
</tbody>
</table>

- Tables in SQL are no proper relations, but may contain duplicates and may be ordered. Duplicates can be eliminated by the user, though.

- The name „Structured English Query Language“ indicates that SQL is a keyword-based language which reads like simple English: All keywords are English natural language words. Keywords are „reserved“ and may not be used for other purposes.

- SQL is a purely textual language without graphical elements.

- SQL consists of two sublanguages:
  - a data definition language (DDL) for defining databases schemas
  - a data manipulation language (DML) for expressing queries and updates

© 2013 Manthey, Behrend
Intelligent Information Systems
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 has already been presented in this lecture:
  
  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
- 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**

The result of a table expression is a derived table, while the result of a conditional expression is a truth value.

**Problem (?):** Only table expressions can be directly posed as queries by the user (similar to selection queries in Access)!
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
  ```

  columns of the result table
  „input“ tables
  selection condition

- example:

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

  Find all capitals with more than a million inhabitants!

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

Meaning of a SELECT-FROM-WHERE block:

- FROM
  - product of all input tables
  - choice of certain rows
- WHERE
  - derived tables
- SELECT
  - choice of certain columns
- result table
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:

\[
\begin{align*}
\text{SELECT} & \quad \text{Capital, Inhabitants} \\
\text{FROM} & \quad \text{city, country} \\
\text{WHERE} & \quad \text{Inhabitants} \geq 1000 \land \text{Name}=\text{Capital} ;
\end{align*}
\]

- 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.
Conditions in the WHERE part

- 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:


<table>
<thead>
<tr>
<th>=</th>
<th>&lt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

| <= | >= |

- 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.
Reminder: SQL and TRC

- But SQL may as well be called a TRC-style language.
- Correspondence between TRC and SQL’s SFW-block:

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

```
{[y.Capital, x.Inhabitants] |
  city(x) ∧ country(y) ∧
  x.Inhabitants ≥ 1000 ∧ x.Name = y.Capital} 
```
**TRC variables in SQL**

- The **tuple variables** which have to be used in TRC have been missing in the SQL formulation on the previous slide, due to some shorthand notation to be explained soon.

- However, the full SQL formulation of this query makes use of explicit variables as in TRC which are introduced in the FROM part of the query by means of the keyword `AS`:

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

- Comparing this form with the **TRC version** of the query immediately shows the close correspondence:

```
{[y.Capital, x.Inhabitants] |
  city(x) ∧ country(y) ∧
  x.Inhabitants ≥ 1000 ∧ x.Name = y.Capital)
```
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;
  ```
Implicit and explicit variables (2)

- The most „economic“ style of writing down a query in SQL does not make use of variables at all – at least not explicitly:

```
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:

```
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.
Natural join in SQL: The direct way

The **direct way** is to „translate“ the definition of a natural join based on projection, selection and product, which are the only operators covered by SELECT-FROM-WHERE anyway:

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

in SQL:

```
SELECT A_1, \ldots, A_m, R.B_1, \ldots, R.B_k, C_1, \ldots, C_n
FROM R, S
WHERE R.B_1 = S.B_1 AND \ldots AND R.B_k = S.B_k
```
JOIN operators in SQL

- Instead of „simulating“ a join operator in this potentially quite tedious way, it is possible to explicitly use one of the variants of the JOIN operator in SQL.

- **JOIN operators** can only be used in the FROM part of a block in order to avoid a selection condition altogether (in case of a natural join) or to place it more closely to the operator (in case of an inner join, see next slide).

- In **standard SQL**: special operator for natural join

```
SELECT *
FROM table1 NATURAL JOIN table2
```

or even:

```
SELECT *
FROM ( SELECT *
        FROM table1
        WHERE A > 0 )
        NATURAL JOIN table2
```

or even:

```
SELECT *
FROM ( SELECT *
        FROM table1
        WHERE A > 0 )
        NATURAL JOIN table2
```

**abbreviation for „all columns“**

**embedded query**
Joins in SQL (2)

- Theta join (or inner join) with explicit join-condition (in an extra ON part):

  ```sql
  SELECT X.A
  FROM ( SELECT *
          FROM table1
          WHERE A > 0 ) AS X
  JOIN table2 AS Y
  ON X.A = Y.A
  ```

- *in general:*

  ```sql
  SELECT . . . FROM R JOIN S ON <condition>
  ```

  ```sql
  SELECT . . . FROM R, S WHERE <condition>
  ```
Other RA operators in SQL

- SELECT-FROM-WHERE (SFW-) blocks are the basic units from which each complex SQL query is composed (representing projection, selection and product).

- More complex queries can be constructed by combining simpler queries by means of one of the three RA operators UNION, INTERSECT, or MINUS (called EXCEPT in the SQL standard).

- When using these operators, union compatibility of the operand expression has to be guaranteed.

- example:

Find all cities, which are not in Germany, or which are capitals!

(This includes Berlin!!)
Intersection expressed as join

- The SQL dialect supported by Access does not know the `INTERSECT` operator.

- It is possible, however, to express an intersection as a special case of an (inner or natural) `JOIN` where all columns of the two operand tables are identified.

- **Example**: Find all cities which are capitals as well!

\[
\begin{align*}
\text{(SELECT Name} & \text{FROM cities }) \quad \text{INTERSECT} \\
\text{(SELECT Capital AS Name} & \text{FROM countries)}
\end{align*}
\]

in Standard SQL

\[
\begin{align*}
\text{SELECT Name} & \text{FROM} \\
\text{(SELECT Name} & \text{FROM cities}) \quad \text{JOIN} \\
\text{(SELECT Capital} & \text{FROM countries)} \quad \text{ON Name = Capital}
\end{align*}
\]

in Access SQL

renaming of columns (union compatibility!)
Block nesting and the IN operator

- SFW blocks can be nested in various ways. We already saw an example where an embedded block is used instead of a table name in the FROM part:

```
SELECT Name, Inhabitants
FROM (SELECT Capital FROM countries) JOIN cities ON Name=Capital
WHERE Inhabitants > 1000.
```

- But blocks can be contained in the WHERE part as well, nested by using the `IN` operator (resembling the element operator $\in$ in set theory):

```
SELECT Inhabitants, Name
FROM cities
WHERE Name IN (SELECT Capital FROM countries)
```

- Both formulations are equivalent, thus IN is just a shorthand notation for joins. However, the IN version more properly reflects that 'countries' does not contribute to the target list of the query but is accessed for test purposes only.
NOT IN and MINUS in Access

- The element operator IN can also be used negatively, combined with the (otherwise logical) operator NOT. NOT IN represents the non-element operator $\notin$ in set theory:

$$\text{SELECT Inhabitants, Name}$$
$$\text{FROM cities}$$
$$\text{WHERE Name NOT IN (SELECT Capital FROM countries)}$$

- This is not an abbreviation for a join! However, NOT IN is able to „simulate“ MINUS:

$$\text{(SELECT Name FROM cities) MINUS (SELECT Capital FROM countries)}$$

Access does not support MINUS either, but it has IN and NOT IN (thus difference can be expressed)
Conditional expressions: Overview

- There is a second large class of SQL-expressions: **Conditions** (or: conditional expressions)

- Conditions are **Boolean expressions**, which are either true or false.

- Conditions appear
  - as **selection criteria** in the WHERE-part of a SELECT-block and
  - as **integrity constraints** in CHECK-clauses (to be discussed later).

- There are two fundamental forms of conditions not otherwise expressable in SQL:
  - **comparisons**
  - **existential conditions**

- **Complex conditions** can be composed from simpler conditions by means of the Boolean operators **AND, OR, NOT** as in propositional logic.

- Various **special forms** of conditions can be equivalently expressed by means of the two basic types of conditions (comparisons and existential conditions) and thus are dispensable as far as pure expressive power is concerned.
Comparisons

- **Comparisons** have been discussed in the context of the WHERE-part of an SQL-block on an earlier slide: Attribute values of a tuple can be compared with other other attribute values or with constant values using one of the six comparison operators:

  \[
  \begin{array}{|c|c|}
  \hline
  = & <> \\
  < & > \\
  <= & >= \\
  \hline
  \end{array}
  \]

- Arguments in complex comparisons may be computed by means of a subquery (provided it can be guaranteed that the answer set contains one element only):
  - **simple**: e.g. \( P\text{.age} = 30 \) or \( P\text{.age} > Q\text{.age} \)
  - **complex**: e.g. \( X\text{.age} > ( \text{SELECT } Y\text{.age FROM person Y WHERE Y.name = 'John'} ) \)

- further **special operators** in elementary comparisons in standard SQL:
  - \( X\text{.name LIKE 'Man%' } \) (%: „wildcard“)  
    („pattern matching“ operator: not otherwise expressable)
  - \( X\text{.age BETWEEN 40 AND 50} \)  
    (interval operator; alternatively expressible via '=<' and '=>')
Existential conditions

- Existential conditions are used for checking whether the answer table of a sub-query is empty, or not:

```
SELECT Name
FROM city
WHERE EXISTS ( SELECT River
    FROM city_at_river
    WHERE City = Name )
```

Which cities are situated close to a river?

- Existential conditions can be negated as well: NOT EXISTS

- Positive existential conditions are expressible without an explicit quantifier by eliminating the nesting of SELECT-blocks:

```
SELECT Name
FROM city, city_at_river
WHERE City = Name
```

- This unnesting technique is not applicable for NOT EXISTS cases, however!
Existential quantifier and duplicates

- Avoiding an existential quantifier is potentially dangerous, as EXISTS is not treated in the same way as product construction in the FROM part by some commercial DBMS.

- In Access-SQL, e.g., an existential quantifier causes automatic elimination of duplicates from the answer to the enclosing SELECT expression. The standard (and the book of Date) interpret the semantics of EXISTS differently:

\[
\text{SELECT Name FROM city, city_at_river WHERE City = Name} \quad \rightarrow \quad \text{Name} = \{\text{Bonn, Koblenz}\}
\]

\[
\text{SELECT Name FROM city WHERE EXISTS ( SELECT River FROM city_at_river WHERE City = Name )} \quad \rightarrow \quad \text{Name} = \{\text{Bonn, Koblenz}\}
\]
**Duplicate elimination in SQL**

- SQL answer tables are no relations in the general case: They may be duplicate-free, but this is not guaranteed, even though all input tables of a query are free of duplicate rows.

- Fortunately, duplicates can be explicitly eliminated by using the keyword `DISTINCT` after SELECT:

  ```sql
  SELECT DISTINCT Name
  FROM city, city_at_river
  WHERE City = Name
  ```

- It is recommendable to always use `SELECT DISTINCT` as soon as a „real“ projection occurs, except if the SELECT part refers to a key column only. – There is no convincing reason for working with duplicates in SQL!
Simulation of FORALL via NOT EXISTS

- SQL has no keyword for universal quantification (no 'FORALL'!).

- Universal conditions have to be „simulated“ by means of logical transformations using double negation and existential quantification based on the following law of predicate logic:

  \[ \forall x: F \iff \neg \exists x: \neg F \]

- Example: „Which river runs through every federal state in Germany?“

- In logic, e.g. in TRC, this query can be formalized as follows:

  \[ \{ x.River | \text{river\_through\_state}(x) \land \forall y: (\text{state}(y) \Rightarrow x.\text{State}=y.\text{Name}) \} \]

- If no „forall“ is available, as in SQL, application of the above mentioned law results in the following more complex formulation:

  \[ \{ x.River | \text{river\_through\_state}(x) \land \neg \exists y: \neg (\text{state}(y) \Rightarrow x.\text{State}=y.\text{Name}) \} \]
Simulation of FORALL via NOT EXISTS (2)

- Applying two more transformation laws of propositional logic eliminates the implication and pushes the inner negation even more inward, thus resulting in a slightly more intuitive formalization:

\[
\{ x.River \mid \text{river\_through\_state}(x) \land \neg \exists y : (\text{state}(y) \land \neg x.\text{State}=y.\text{Name}) \}
\]

- If this simple query is to be expressed in SQL, it is necessary to go exactly this way (involving quite a bit of logic) in order to be able to „simulate“ FORALL:

```sql
( SELECT X.River
  FROM river_through_state AS X
  WHERE NOT EXISTS
    ( SELECT *
      FROM state AS Y
      WHERE NOT X.State = Y.Name )
)
```
**Aggregate functions**

- Important class of „built-in“-functions in SQL: **aggregate functions**

<table>
<thead>
<tr>
<th>COUNT</th>
<th>SUM</th>
<th>AVG</th>
<th>MAX</th>
<th>MIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality</td>
<td>Sum</td>
<td>Average</td>
<td>Maximum</td>
<td>Minimum</td>
</tr>
</tbody>
</table>

- Computation of **one** scalar value from a **set** of scalar values (the aggregate) originating from **one** column of **one** table:
Aggregate functions (2)

Examples of aggregate expressions in the SELECT-part:

Compute the overall salary of all C3-professors!

```
SELECT SUM(P.salary) AS Total
FROM professors AS P
WHERE P.Rank = 'C3'
```

Which C3-professors are older than all C4-professors?

```
SELECT P.Name
FROM professors AS P
WHERE P.Rank = 'C3' AND P.Age > (SELECT MAX(Q.Age)
FROM professors AS Q
WHERE Q.Rank = 'C4')
```
Aggregate functions (3)

- often used in connection with aggregate functions:
  extended SELECT-blocks with subdivision of the resultat tables in groups

- syntactic extension: GROUP BY- and (possibly) HAVING-part in SELECT-blocks

- basic idea: The result of the evaluation of SELECT-FROM-WHERE (a table)
  is divided into „subtables“ (groups) with identical values for certain
  grouping columns (specified in the GROUP BY-part)

- optional: Groups not satisfying a certain additional condition (HAVING-part),
  are eliminated.

- Aggregate functions are applied to groups (as aggregates), if GROUP BY has been
  specified:

  e.g.:

  ```sql
  SELECT P.Rank, AVG(P.Age) AS AvgAge
  FROM professors AS P
  GROUP BY P.Rank
  HAVING P.Rank > 'C2'
  ```
### Aggregate functions (4)

Illustration with example data:

```
SELECT P. Rank, AVG(P.Age) AS AvgAge
FROM professors AS P
WHERE P.Name <> 'Ken'
GROUP BY P. Rank
HAVING P. Rank > 'C2'
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>C4</td>
<td>43</td>
</tr>
<tr>
<td>John</td>
<td>C3</td>
<td>33</td>
</tr>
<tr>
<td>Ken</td>
<td>C4</td>
<td>57</td>
</tr>
<tr>
<td>Lisa</td>
<td>C4</td>
<td>39</td>
</tr>
<tr>
<td>Tom</td>
<td>C2</td>
<td>32</td>
</tr>
<tr>
<td>Eva</td>
<td>C3</td>
<td>36</td>
</tr>
</tbody>
</table>

**Group by**

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>C4</td>
<td>43</td>
</tr>
<tr>
<td>Lisa</td>
<td>C4</td>
<td>39</td>
</tr>
<tr>
<td>John</td>
<td>C3</td>
<td>33</td>
</tr>
<tr>
<td>Eva</td>
<td>C3</td>
<td>36</td>
</tr>
<tr>
<td>Tom</td>
<td>C2</td>
<td>32</td>
</tr>
</tbody>
</table>

**Average**

<table>
<thead>
<tr>
<th>Rank</th>
<th>AvgAge</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3</td>
<td>34.5</td>
</tr>
<tr>
<td>C4</td>
<td>41.0</td>
</tr>
</tbody>
</table>

**Where**

```
WHERE P.Name <> 'Ken'
```

**Having**

```
HAVING P. Rank > 'C2'
```
**Sorting tables in SQL**

- **Sorting** of the result table can be specified at the end of a SELECT-block (after `GROUP BY` or `HAVING`, resp., if present at all)

- **Example:**

  ```sql
  SELECT X.Rank, X.Salary
  FROM professors AS X
  ORDER BY X.Rank DESC, X.Salary ASC
  ```

- **“Direction“ of sorting**:  
  - ASC (ascending, default value if unspecified)  
  - DESC (descending)

- The **order** of columns is always respected when sorting, thus introducing multiple sorting criteria.

- Sorting can be specified independent of aggregation.
Null values

- SQL offers a predefined, universal null value **NULL**, intended to represent unknown or missing information in a systematic way.

- Correct usage of **NULL** is difficult, partly because there are a number of inconsequent design decisions in the SQL standard.

- Null values can be interpreted in a number of different ways. Possible interpretations are:
  - Value exists, but is presently unknown.
  - It is known that in this row no value exists in the respective column.
  - It is not known if a value exists or if so, what it is like.

- intended interpretation of **NULL** in SQL: **Value exists, but is unknown!**

- **thus**: **NULL** is denoted a „value“! Each two occurrences of **NULL** represent different „real“ values presently (still) unknown.

- **however**: **NULL** itself doesn‘t have a type but always takes the type of the resp. column under consideration.
Null values (2)

- NULL can, however, **not** be used like a „normal“ value in several cases, e.g.
  - NULL may **not** occur as a parameter of a function (e.g.: X+NULL)
  - NULL may **not** occur in comparisons (e.g.: X=NULL)

- For testing whether a column contains NULL a special syntax is offered:
  
  \[
  \begin{align*}
  X.&\text{column name} & \text{ IS NULL} \\
  X.&\text{column name} & \text{ IS NOT NULL}
  \end{align*}
  \]

- If the evaluation of a subexpression returns NULL, then the entire expression returns NULL as a result, too, e.g.:

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>33</td>
</tr>
<tr>
<td>Tom</td>
<td>NULL</td>
</tr>
</tbody>
</table>

SELECT (65 - Age) AS Rest
FROM person
WHERE Name = 'Tom'

NULL
Null values (3)

- **Exception**: Aggregate functions ignore NULL „on purpose“!

```
Name   Age
------  ----
Jim     33
Tom     NULL
```

- SUM (Age): 33
- COUNT (Age): 1
- AVG(Age): 33

- „Exception from the exception“: COUNT(*) does not ignore NULL tuples!

- In comparisons (and other conditions) NULL leads to usage of a **three-valued logic**, i.e. a logic with three rather than two truth values:
  - TRUE
  - FALSE
  - UNKNOWN

- **Example**: If A=3, B=4 and C=NULL, then . . .
  - A > B AND B > C results in FALSE
  - A > B OR B > C results in UNKNOWN
Null values (4)

- Truth tables of Boolean operators in three-valued logic:

```
<table>
<thead>
<tr>
<th>AND</th>
<th>T</th>
<th>U</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>OR</th>
<th>T</th>
<th>U</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>U</td>
<td>F</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>NOT</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>
```

- Respecting NULL when evaluating SELECT-expressions means:
  - All FROM-rows are eliminated for which the WHERE-part does not return TRUE.
  - consequently: A row is eliminated as soon as the WHERE-part returns FALSE or UNKNOWN !!
Null values (5)

- SQL offers a special syntax for testing the truth value of a condition:

  `<conditional-expression> IS [ NOT ] { TRUE | UNKNOWN | FALSE }`

- Semantics of such IS-expressions: TRUE if and only if the evaluation of the left-hand expression returns the truth value on the right-hand side; FALSE else.

- Consequence: `p IS NOT TRUE` is no longer equivalent with `NOT p`!
  (if `p` is UNKNOWN, then `NOT p` returns UNKNOWN, too)

- Further „logical trap“: `EXISTS` doesn‘t behave like an existential quantifier in three-valued logic
  `EXISTS (<table-expression>)` returns FALSE, if `<table expression>` results in an empty table, TRUE else -- but never UNKNOWN!

- Chapter 16 in Date‘s SQL-book closes with the following (very brief) section:

  **16.6 A RECOMMENDATION**
  Avoid nulls.
### Outer join

- Automatic generation of null values when using an **OUTER JOIN**-operator:
  \[
  \{ \text{LEFT | RIGHT | FULL} \} \ [ \text{OUTER} ] \ \text{JOIN}
  \]

- Semantics: „Normal“ join extended by rows filled up with **NULLs**, containing values which would otherwise not appear in a join.

- Example:

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

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

  ```
  SELECT *
  FROM     p  FULL OUTER JOIN q   ON  p.B >= q.B
  ```

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

  Always contains **INNER JOIN** as subtable!
Outer join (2)

• **LEFT and RIGHT OUTER JOIN**: Only the "non-joining" elements of the left or right table, resp., are filled up with NULLs.

• **Example**:

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

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

```
SELECT * FROM p LEFT OUTER JOIN q ON p.B >= q.B
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>NULL</td>
<td>5</td>
</tr>
</tbody>
</table>
```

- in Access-SQL: Only LEFT JOIN and RIGHT JOIN are supported, no FULL OUTER JOIN; "OUTER" is omitted.
Empty tables (and not so empty ones)

- How does an empty table look like in SQL?

- In set theory, „empty“ means: without elements. Thus, an empty table does not contain any row.

- Don‘t confuse this with a table containing just one row the fields of which all consist of NULL values – such a table is not (really) empty!

- In the datasheet view of Access the difference is clearly visible:

```
<table>
<thead>
<tr>
<th>City</th>
<th>River</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**city_at_river**

- non-empty table, consisting of a „NULL-row“

```
<table>
<thead>
<tr>
<th>City</th>
<th>River</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**city_at_river**

- empty table not containing any row
Boolean queries in SQL

- How to "simulate" a yes/no-query in SQL?
  e.g.: Is there a city with more than 4 million inhabitants?

- With table queries, only an indirect answer is possible:
  An empty answer table is interpreted as „no“.

  SELECT Name
  FROM city
  WHERE Inhabitants > 4000;

  if yes: non-empty answer table
  if no: empty table

- More reasonable, but not (yet) possible as a „stand-alone“ query according to the SQL standard:

  CHECK EXISTS (SELECT Name FROM city WHERE Inhabitants > 4000)
Boolean queries (2)

There is a rather awkward formulation of this query (ab)using the concept of a table query as follows:

\[
\begin{align*}
( & \text{SELECT 'Yes' AS Answer} \\
& \text{FROM city} \\
& \text{WHERE EXISTS (SELECT * FROM city} \\
& \text{    WHERE Inhabitants > 4000))} \\
\text{UNION} \\
( & \text{SELECT 'No' AS Answer} \\
& \text{FROM city} \\
& \text{WHERE NOT EXISTS (SELECT * FROM city} \\
& \text{    WHERE Inhabitants > 4000))}
\end{align*}
\]

Even though this still is a table query, it at least appears like a yes/no query:

\[
\begin{array}{cc}
\text{Answer} & \text{Answer} \\
\text{Yes} & \text{No}
\end{array}
\]
Update operations in SQL: Overview

• already mentioned at the beginning of this section: Update statements are part of the DML-sublanguage of SQL, too.

• SQL offers three basic operations for changing data:
  - INSERT insertion of rows
  - UPDATE modification of values in columns
  - DELETE deletion of rows

• All three types of update operation can be combined with queries for retrieving the rows of a particular table to be inserted/updated/deleted.

• Reminder: There is the danger of a terminology conflict:
  - „Update“: in colloquial DB-style refers to any kind of change
  - UPDATE in SQL: means column value replacement only
**INSERT-Operation**

- Format of insertions:

\[
\text{INSERT INTO} \quad <\text{table-name}> \quad [ \quad ( \quad <\text{list-of-columns}> \quad ) \quad ] \quad <\text{table-expression}> \\
\]

- two variants:
  - **direct** reference to one or more rows to be inserted, e.g.
    - \(\text{INSERT INTO} \quad \text{professors} \quad (\text{Name}, \text{Rank}, \text{Department}) \quad \text{VALUES} \quad (\text{''Cremers‘, ,C4‘, ,III‘})\)

- **indirect** identification of the rows to be inserted via a query, e.g.
  - \(\text{INSERT INTO} \quad \text{professors} \quad \text{SELECT} \quad * \quad \text{FROM} \quad \text{researchers} \quad \text{AS} \quad R \quad \text{WHERE} \quad \text{R.qualification} \quad = \quad \text{’PhD‘}\)
UPDATE- and DELETE-operation

- Format of modifications:

  ```sql
  UPDATE <table-name>
  SET <list-of-assignments>
  [ WHERE <conditional-expression> ]
  ```

- Modifies all rows of "table name" satisfying the WHERE-part according to the assignments of values to columns given in the SET-part.

- Syntax of an individual assignment:

  ```sql
  <column-name> = { <scalar-expression> | DEFAULT | NULL }
  ```

- Example:

  ```sql
  UPDATE professors
  SET Name = 'N.N.'
  WHERE Dept = 'II'
  ```

  assignment (action)

  condition (test in the „old“ state)

- Quite similar: deletions

  ```sql
  DELETE FROM <table-name>
  [ WHERE <conditional-expression> ]
  ```
Schema definition in SQL

- The **DDL**-part of SQL is a language for defining a relational DB **schema**, i.e., a collection of table structures. Before a database can be populated with data, its schema has to be defined.

- SQL offers a number of operations for defining a schema:
  
  CREATE TABLE, CREATE VIEW, CREATE DOMAIN etc.

- In addition to defining the structure (i.e. the type) of the tables, a number of **semantic rules** can be associated with the schema. There are three kinds of such rules:
  
  - **View** definitions (also called deductive rules)
  - **Integrity constraints** (normative rules)
  - **Triggers** (active rules)

- Once a schema has been defined and data have been inserted into the resulting database, it is possible to modify the structure and the rules of a database by means of special operations of the SQL-DDL: **schema evolution**
CREATE TABLE: Principle

- most important DDL-operation: Creation of a new table

```
CREATE TABLE <table-name> ( <lists-of-table-elements> ) ;
```

- "table elements" are
  - definitions of name and data type of each column, and
  - constraints referring to the newly created table.

- Syntax of a table definition:

```
CREATE TABLE <table-name>
    <column-name_1> <type_1> [ <column-constraints_1> ],
    <column-name_2> <type_2> [ <column-constraints_2> ],
    ...
    <column-name_n> <type_n> [ <column-constraints_n> ]
[ <table-constraints> ]
```

Integrity constraints
- for individual columns
- for the entire table
Example:

SQL-statement defining a table `FootballMatch` containing the results of football matches in the national league:

```sql
CREATE TABLE FootballMatch
(
    Date date,
    HomeTeam text,
    GoalsH number(15) DEFAULT NULL CHECK ( >= 0 OR IS NULL),
    GuestTeam text,
    GoalsG number(15) DEFAULT NULL CHECK ( >= 0 OR IS NULL),
    Round number(15) NOT NULL CHECK ( > 0 AND < 35),
    PRIMARY KEY (Date, HomeTeam),
    FOREIGN KEY ( HomeTeam ) REFERENCES Teams,
    FOREIGN KEY ( GuestTeam ) REFERENCES Teams
) ;
```
CREATE TABLE: General structure

Each table definition consists of two parts: The definitions of the individual columns, and (possibly) constraints valid for the entire table:

CREATE TABLE **FootballMatch**

(  
  Date date,  
  HomeTeam text,  
  GoalsH number(15) DEFAULT NULL  
    CHECK ( > = 0 OR IS NULL),  
  GuestTeam text,  
  GoalsG number(15) DEFAULT NULL  
    CHECK ( > = 0 OR IS NULL),  
  Round number(15) NOT NULL  
    CHECK ( > 0 AND < 35),  
  PRIMARY KEY (Date, HomeTeam),  
  FOREIGN KEY ( HomeTeam ) REFERENCES Teams,  
  FOREIGN KEY ( GuestTeam ) REFERENCES Teams  
) ;
CREATE TABLE: Column definitions

Each column definition itself consists of two parts, too:

- the declaration of a column name and a type of its values
- (possibly) special constraints for the values in this column

Syntax of column definitions:

```
<column-name> { <data-type> | <domain> } [ <column-constraints> ]
```

CREATE TABLE  **FootballMatch**

(  
  Date  date,  
  HomeTeam text,  
  GoalsH number(15) DEFAULT NULL  
  CHECK ( >= 0 OR IS NULL),  
  GuestTeam text,  
  GoalsG number(15) DEFAULT NULL  
  CHECK ( >= 0 OR IS NULL),  
  Round number(15) NOT NULL  
  CHECK ( > 0 AND < 35),  
  ...
)

unique within the same table

alternatives
Data types and domains

- As in each programming language: There are various predefined data types for column entries (i.e., for fields of the table).

- **Attention!** Data type names in commercial DBMSs deviate from those of the SQL-standard (See books or tutorials for more details.)

- **in addition:** Application-specific value domains can be introduced via separate domain definitions, e.g.

  ```sql
  CREATE DOMAIN big_eu_capitals AS text(15)
  DEFAULT ''
  CHECK ( VALUE IN ('Paris', 'London', 'Berlin', 'Rome', 'Madrid', 'Brussels', 'Vienna', '' ) )
  ```

- Self-defined domains can be used like basic predefined data types in column definitions.

- **in this introduction:** no further discussion of the domain concept.
CREATE TABLE: Column constraints

Each column definition itself consists of two parts, too:

- the declaration of a column name and a type of its values
- (possibly) special constraints for the values in this column

Syntax of column constraints:

[ NOT NULL | UNIQUE ]
[ PRIMARY KEY ]
[ DEFAULT { <literal> | NULL } ]
[ REFERENCES <table-name> ]
[ CHECK <condition> ]

CREATE TABLE FootballMatch
(
    Date date,
    HomeTeam text,
    GoalsH number(15)  DEFAULT NULL CHECK ( >= 0 OR IS NULL),
    GuestTeam text,
    GoalsG number(15)  DEFAULT NULL CHECK ( >= 0 OR IS NULL),
    Round number(15)  NOT NULL CHECK ( > 0 AND < 35),
    ...
)

left-hand side remains implicit: current column
The second part of a table definition is optional. It consists of one or more table constraints, normally expressing a restriction on several columns:

```
CREATE TABLE FootballMatch
(
    PRIMARY KEY (Date, HomeTeam),
    FOREIGN KEY (HomeTeam) REFERENCES Teams,
    FOREIGN KEY (GuestTeam) REFERENCES Teams
)
```

Syntax of table constraints:

```
[ UNIQUE ( <list-of-column-names> ) ]
[ PRIMARY KEY ( <list-of-column-names> ) ]
[ FOREIGN KEY ( <list-of-column-names> ) REFERENCES <table-name> ]
[ CHECK ( <condition> ) ]
```
Constraints in table definitions

- Table definitions (CREATE TABLE) contain two very similar kinds of constraints:
  - column constraints
  - table constraints (also called: row constraints)

- Column constraints are abbreviations of certain special forms of table constraints where the name of the resp. column remains implicit, e.g.

  - column constraint:
    
    \[
    \text{Type} \quad \text{number}(15) \quad \text{CHECK} \left( > 0 \text{ AND } < 35 \right),
    \]

  - table constraint:
    
    \[
    \text{CHECK} \left( \text{Type} > 0 \text{ AND } \text{Type} < 35 \right)
    \]

- The condition part of such a CHECK constraint has to be satisfied in each admissible (legal, consistent) state of the database.
UNIQUE and NOT NULL

- **UNIQUE**-option: definition of a key (or: candidate key)
  - single-column key:
    in a column definition: \(<\text{column-name}\> \ldots \text{UNIQUE}\)
  - multi-column key:
    separate UNIQUE-clause as table constraint:
    \(\text{UNIQUE} ( <\text{list-of-column-names}>)\)

- **Semantics**: No two rows will ever have the same value in columns belonging to a key.

- **Exception**: Null values – NULL may occur several times in a UNIQUE-column.

- **per table**: Arbitrarily many UNIQUE-declarations are possible.

- In a table with UNIQUE-declarations no duplicates (identical rows) can exist!

- **Exclusion of null values** for individual columns: \(<\text{column-name}\> \ldots \text{NOT NULL}\)
PRIMARY KEY and DEFAULT

- **per table**: At most one (candidate) key can be declared the primary key.
  - single-column primary key:
    in column definition: `<column name> . . . PRIMARY KEY`
  - multi-column primary key:
    separate clause: `PRIMARY KEY ( <list-of-column-names> )`

- **in addition**: No column within a primary key may contain **NULL**!

- PRIMARY KEY is **not** the same as UNIQUE NOT NULL!
  (in addition: Uniqueness of the p. key within the table)

- not a real „constraint“, but rather similar in syntax:
  Declaration of a **default value** for columns of a table:
  Value which is automatically inserted if no explicit value is given during the insertion of a new row, e.g.

```
Type   number(15) DEFAULT 0
```
Foreign key constraints

- second special form of constraint within a table declaration:

  foreign key constraint (aka referential constraint)

- situation: Column(s) of the table declared (called A) reference(s) (i.e., contains values of) a candidate key or primary key of another ("foreign") table B

  condition: A-columns contain only values actually occurring in the referenced B-column(s)!
Foreign key constraints (2)

Syntax of the corresponding constraint (as table constraint):

\[
\text{FOREIGN KEY ( <list-of-column-names> )} \\
\text{REFERENCES <table-name> [ ( <list-of-column-names> ) ]}
\]

if „target columns“ are missing: primary key assumed

e.g.:

```
CREATE TABLE t1
   ( a1 INT PRIMARY KEY, 
     ..... 
 )

b1 references a1
```

```
CREATE TABLE t2
   ( b1 INT REFERENCES t1, 
     ..... 
 )
```

abbreviated form as column constraint
Foreign key constraints (3)

- Complete syntax of a „referential constraint“ provides for various optional extensions:

  FOREIGN KEY  (  <list-of-column-names>  )
  REFERENCES  <base-table-name>  [ (  <list-of-column-names>  ) ]

  [ MATCH  {  FULL  |  PARTIAL  }  ]
  [ ON DELETE  {  NO ACTION  |  CASCADE  |  SET DEFAULT  |  SET NULL  }  ]
  [ ON UPDATE  {  NO ACTION  |  CASCADE  |  SET DEFAULT  |  SET NULL  }  ]

  „referential actions“

  specify what happens in case of integrity violations

- Detailed discussion of all these extensions is beyond the scope of this short introduction.

- Access treats references and referential integrity quite similarly:
  - with change propagation:  ON UPDATE CASCADE
  - with delete propagation:  ON DELETE CASCADE
Global constraints in SQL

- Not supported by any commercial DB system till now, but defined in the SQL standard:

- Assertions serve as a means for expressing global integrity constraints not tied to a particular table, but ranging over several tables.

- Syntax:

  ```
  CREATE ASSERTION <constraint-name> 
  CHECK ( <conditional-expression> )
  ```

- In principle, assertions are sufficient for expressing all imaginable constraints, i.e. all "local" forms of constraints are redundant.

- On the other hand, many constraints can only be expressed via assertions, but not by means of table constraints.

- Example:

  ```
  CREATE ASSERTION lazy_professor 
  CHECK EXISTS 
  ( SELECT * FROM professor 
    WHERE Name NOT IN ( SELECT Teacher 
                        FROM courses ) )
  ```
Integrity checking in SQL

• important topic related to SQL constraints:
  Modalities of checking for constraint violations

• Changes in SQL are usually part of greater units of change called transactions:
  - **Transaction**: Sequence of DML statements viewed as "indivisible units"
  - Transactions are either executed completely, or not at all!
  - Transactions always have to lead to consistent DB states satisfying all integrity constraints stated in the resp. DB schema.
  - more detailed discussion of the concept "transaction": later!

• important motivation for introducing transactions:
  Some transitions from a consistent state into a consistent follow-up state are only possible via inconsistent intermediate steps!

• Consequence for integrity checking during transaction processing:
  Checking of constraints should (more or less always) take place at the end of a transaction!
Integrity checking in SQL (2)

- **in SQL however**: Unless defined otherwise, integrity checking always happens immediately (i.e., directly after the execution of each update).

- **Motivation**: Many simple table constraints can and ought to be checked immediately as they are independent of any other updates.

- **but in particular for „referential cycles“**: Checking at transaction end is inevitable!

  e.g.:

  \[
  \begin{align*}
  \text{C}_1: & \quad \text{„Each course is given by a professor!"} \\
  \text{C}_2: & \quad \text{„Each professor has to give at least one course!"
  \end{align*}
  \]

When hiring a new professor a consistent state can be reached only via a transaction consisting of two individual insertions:

\[
\begin{align*}
\text{INSERT} & \quad \text{INTO} \quad \text{professor} \\
\text{INSERT} & \quad \text{INTO} \quad \text{course}
\end{align*}
\]

Each intermediate state would be inconsistent: **No sequence possible!**
Integrity checking in SQL (3)

- **thus**: two forms of integrity checking in SQL

  - **IMMEDIATE** and **DEFERRED**

- **meaning**: IMMEDIATE-constraints are *immediately* checked, for DEFERRED-constraints checking is deferred to the end of the current transaction.

- **unfortunately**: Without explicitly stating one of these alternatives, IMMEDIATE is assumed (which somehow contradicts the idea of a transaction).

- This default assumption can be changed for individual constraints by declaring them as

  - **INITIALLY DEFERRED**.

- **”INITIALLY“**, because the checking status can be changed dynamically during a running transaction:

  ```sql
  SET CONSTRAINTS { < list-of-constraints > | ALL } { DEFERRED | IMMEDIATE }
  ```

- **in addition**: Some constraints can be declared **NOT DEFERRABLE**. But the even more important NOT IMMEDIATE does not exists in SQL!

- **in summary**: Integrity checking in „full" SQL can be a difficult affair!
Views

- Predefined queries for computation of derived tables as in Access can be declared in an SQL schema as well:

- Views are defined in a separate `CREATE VIEW` statement, simply assigning a name to a query (formulated in SQL-DML), e.g.:

  ```sql
  CREATE VIEW metropolis AS
  ( SELECT ID, Name, Inhabitants, Country
  FROM city
  WHERE Inhabitants >= 1000 ) ;
  ```

- According to the latest edition of the SQL standard, views may even refer to themselves. Such views are called recursive. In this case, the keyword `RECURSIVE` has to be given in front of `VIEW`.

- Recursive views are very useful for traversing data representing graphs such as maps or hierarchies (e.g., „Find all connections from X to Y of arbitrary length!“)
Queries over views

- Queries involving a view are interpreted by expanding the view name, i.e. by textually replacing it by the query associated with it in the view definition:

  CREATE VIEW C4-profs
  AS ( SELECT Name, Dept
       FROM professors
       WHERE Rank = 'C4' )

  SELECT Name
  FROM C4-profs
  WHERE Dept = 'III'

- Note that this technique does no longer work for recursive views, as expansion would never terminate! Other, more elaborate techniques are required in this case, investigated in this lecture later on.
Already in early versions of SQL and in first relational systems an automatic triggering of follow-up changes by the DBMS as a reaction to changes explicitly stated by users or application programs has been suggested.

Declaration of such implicit changes and their combination with triggering events can be done within an SQL schema, too:

- other notion for trigger: active rule
- Name of a DBMS supporting triggers: active DBMS
- Name of the corresponding research area: active databases

In the SQL92-Standard a trigger concept was still missing.

**but**: Most commercial DB products already provide triggers in a rather similar form since many years (ORACLE, DB/2, Sybase, Informix, e.g.).

Since the SQL3-Standard (1999) triggers have been standarized for the first time.
Active rules

- Active rules are called **ECA-rules** as well, thus referring to the three components of such rules:

  - "event"
  - "condition"
  - "action"

- Example of an ECA-rule (in pseudo-code):

  ```
  ON         modify(account(A), V_new)
  IF V_new < credit(A)
  DO         block_account(A)
  ```
Active rules (2)

**general meaning of an active rule:**

*additional, automatically triggered „background activity“*

„surface process“ (e.g. a transaction)

observe

check

react

May recursively trigger other active rules!
example of an SQL-trigger:

CREATE TRIGGER firstCourse
AFTER INSERT ON professors
REFERENCING NEW ROW AS Newcomer
FOR EACH ROW
WHEN (NOT EXISTS
(SELECT *
FROM courses
WHERE Name = Newcomer.Name)
BEGIN ATOMIC
INSERT INTO courses
VALUES (Newcomer.Name, NULL, 4-hrs.);
INSERT INTO exercises
VALUES (Newcomer.Name, NULL, 2-hrs.)
END

"transition variable"