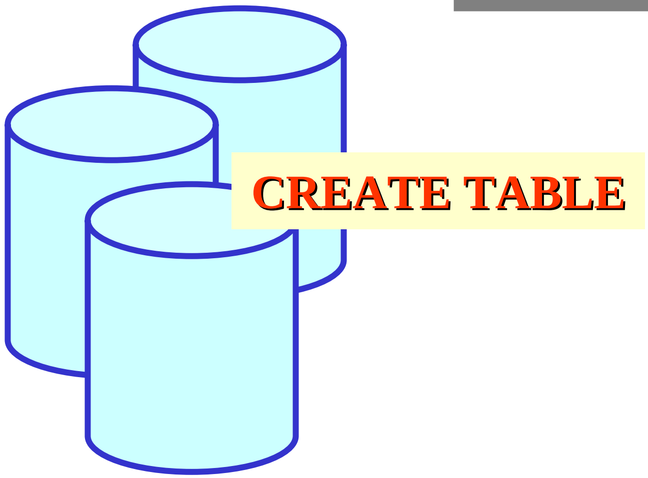
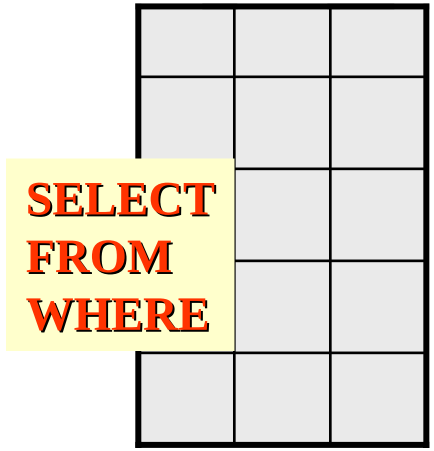
Foundations of Information Management (WS 2008/09)

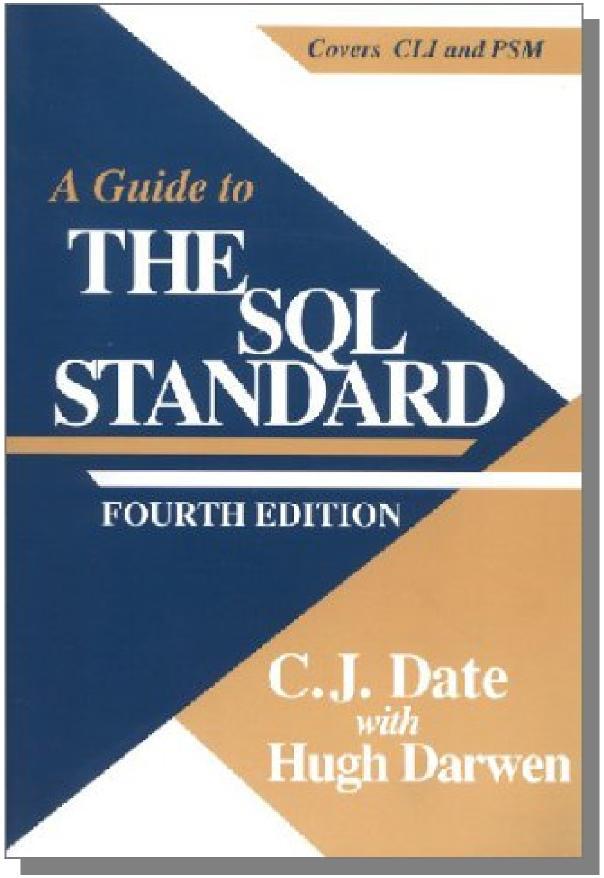


– Chapter 3 –

An Introduction to SQL



- 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 of the standard over the years:
  SQL2 or SQL92, resp., SQL3 or SQL:1999, SQL:2003
- Attention ! Nearly every commercial DB product has its own „dialect" of SQL. None of them implements it completely and exactly.
- . . . and: SQL is a „huge“ language – more than 1500 pages of standard text.



„Classical" (but arguably still the best) source about SQL:

Chris Date, Hugh Darwen:
„A Guide to the SQL Standard"

ISBN 0-201 964-260
Addison Wesley, 1997 (4th edition)
~ € 44

Good new book about the new SQL standard:
Melton/Simon: "SQL:1999 Understanding Relational Language Components", Academic Press, 2002

- SQL has its own terminology of relational concepts:

Access			SQL
	datasheet	table	
	field	column	
	record	row	
	data type	domain	

- 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.

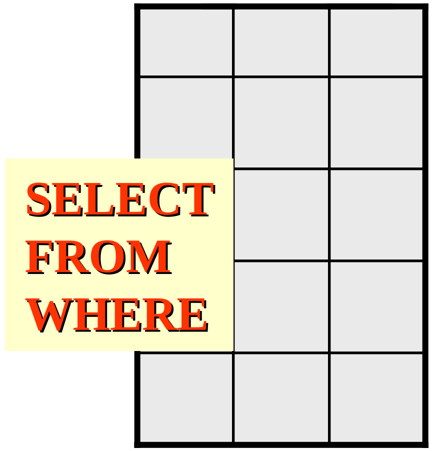
- SQL is a purely textual language without graphical elements.

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

Foundations of Information Management (WS 2008/09)

Data Manipulation in SQL

– 3.1 –



- SQL data manipulation language: statements for „manipulating" data
- Two forms of manipulation:
    - Evaluation of queries
    - Execution of updates
- The format of simple queries has already been addressed in connection with Access:

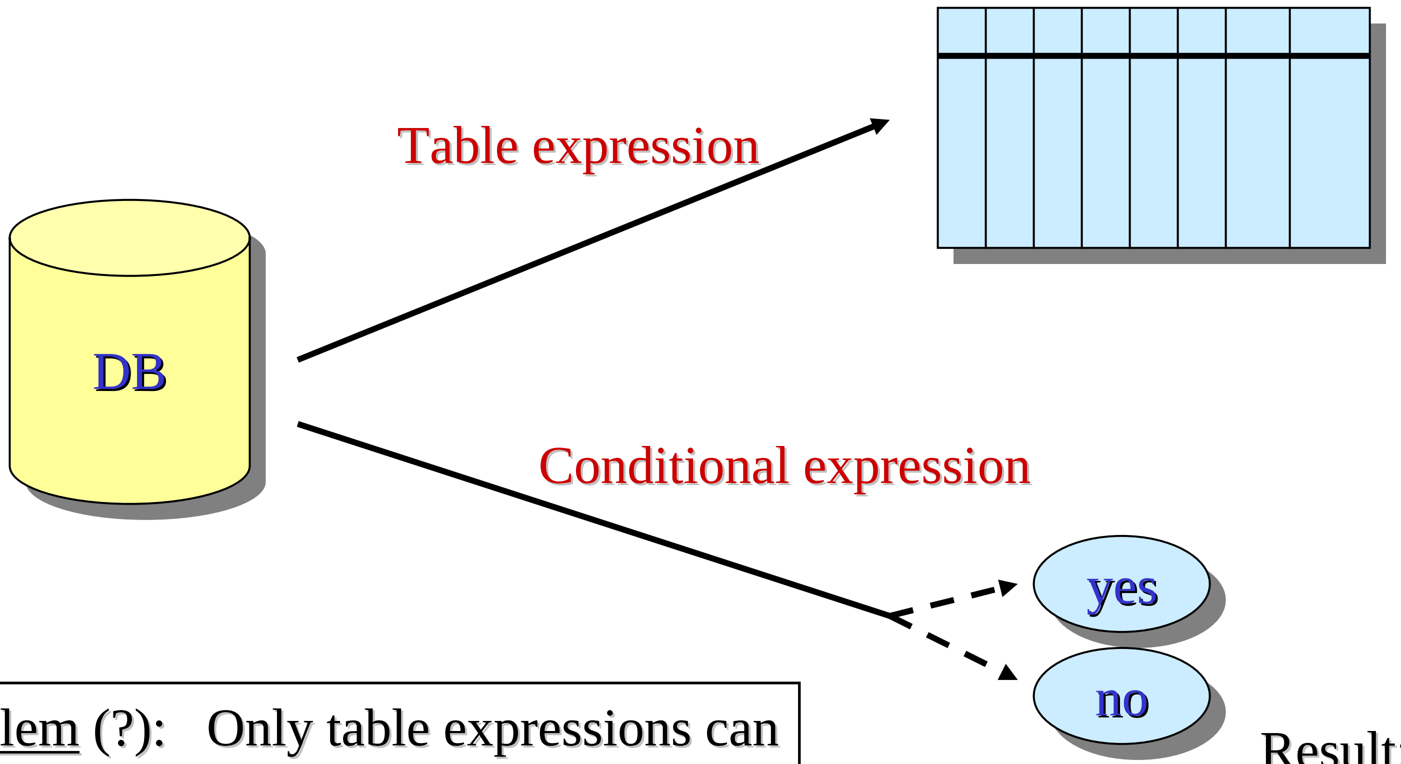
  SELECT-FROM-WHERE statements

- 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 !
- At the end of this section: Treatment of update statements in SQL

  (INSERT, DELETE, UPDATE etc.)

In SQL, there are two types of queries:

Result:
derived table



Result:
truth value

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

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

- Syntactic structure in the simplest case:

```
SELECT   (list_of_column_names)    ← columns of the result table
FROM     (list_of_table_names)     ← „input“ tables
WHERE    condition                 ← selection condition
```

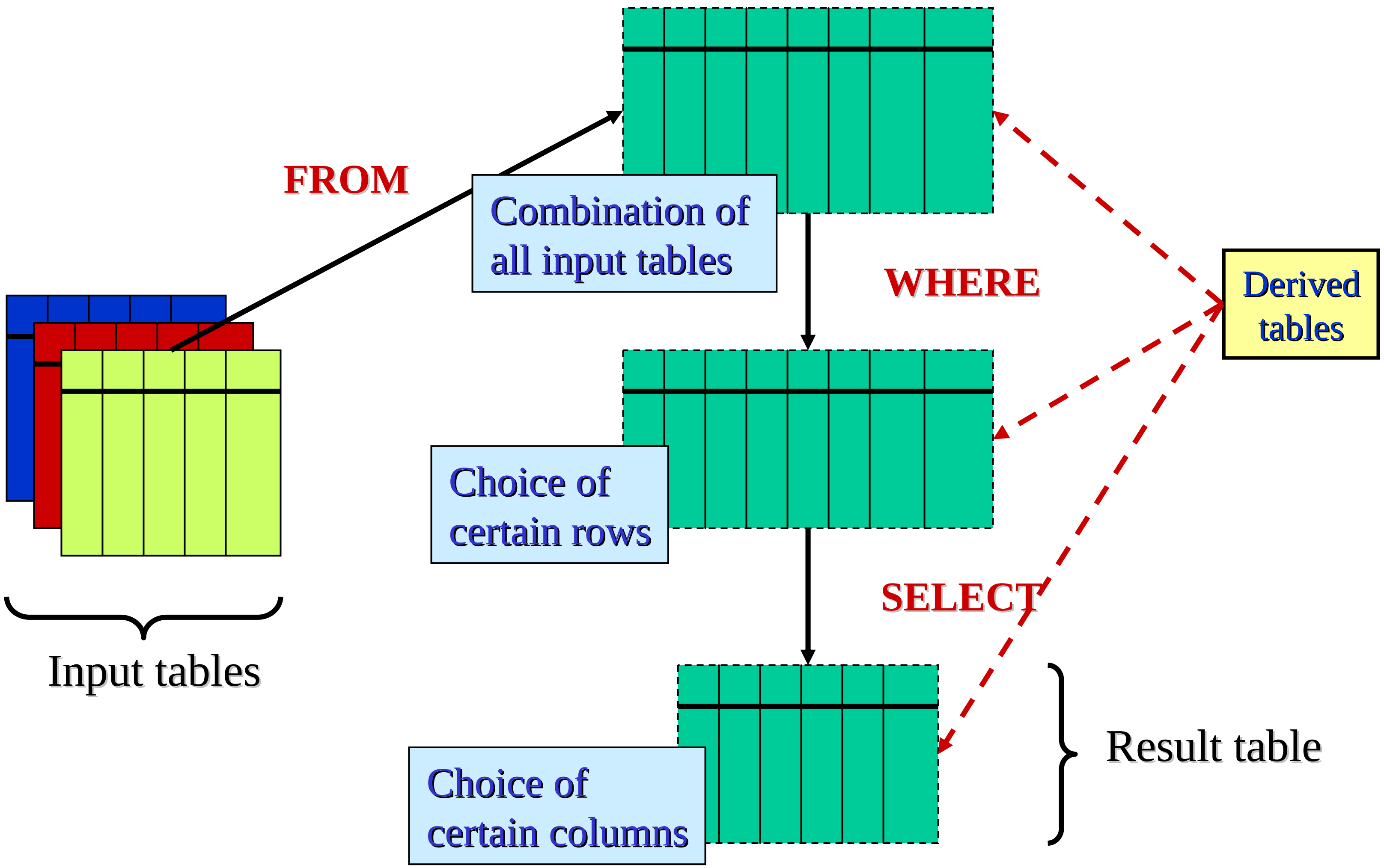
- Example:

```
SELECT   capital, population
FROM     cities, countries
WHERE    population >= 1000 AND
         city=capital ;
```

Find all capitals with more than a million population !

- In SQL, upper or lower case does not matter for table and column names.

Meaning of a SELECT-FROM-WHERE block:



Let us apply this 3-step process for evaluating the example query over the „Europe“ DB:

```
SELECT   capital, population
FROM     cities, countries
WHERE    population >= 1000 AND
         city=capital ;
```

Microsoft Access - [cities : Tabelle]

city	country	population	year
Amsterdam	NL	731288	2000
Andorra la Vella	AND	20787	2001
Ankara	TR	3203362	2000
Athens	GR	772072	1991
Barcelona	E	1503884	2001
Belgrade	SRB	1597599	1997
Berlin	D	3386667	2000
Bern	CH	122469	2001
Birmingham	GB	1010400	2000
Bratislava	SK	447345	2000
Bremen	D	540950	2002

Datensatz: 11 von 77

77 cities

Microsoft Access - [countries : Tabelle]

country	code	capital	area	population
Austria	A	Vienna	83850	8023244
Albania	AL	Tirane	28750	3249136
Andorra	AND	Andorra la Vella	450	72766
Belgium	B	Brussels	30510	10170241
Bulgaria	BG	Sofia	110910	8612757
Bosnia and Herzegovina	BIH	Sarajevo	51233	2656240
Belarus	BY	Minsk	207600	10415973
Switzerland	CH	Bern	41290	7207060
Cyprus	CY	Nicosia	9250	744609
Czech Republic	CZ	Prague	78703	10321120
Germany	D	Berlin	356910	83536115

Datensatz: 1 von 46

46 countries

```
SELECT    capital, population
FROM      cities, countries
WHERE     population >= 1000 AND
          city=capital ;
```

In the first step, a huge table containing all 46 * 77 = 3542 combinations of rows on citites and rows in countries is formed – at least conceptually.

city	country	population	year	country	code	capital	area	population
				3542 rows !!				

This huge table is called the product of cities and countries. Never form a product unless you make sure to „cut it down“ immediately after – or unless you actually want it that big!

```
SELECT    capital, population
FROM      cities, countries
WHERE     population  >= 1000 AND
          city=capital ;
```

In the next step, all those rows are eliminated from the enormous product table which do not satisfy the WHERE-condition – only 46 capitals remain, of which 16 are big enough.

city	country	population	year	country	code	capital	area	population

16 rows !!

Still, all the columns remain – even though most of them are irrelevant for the query.

```
SELECT    capital, population
FROM      cities, countries
WHERE     population >= 1000 AND
          city=capital ;
```

Finally, the unwanted columns are eliminated – i.e., only capital and population remain. OOPS, there is a problem: We do know which population copy to take (the one originating from *cities*) but the DB system doesn't know! Let us postpone the problem!

~~city~~	~~country~~	**population**	~~year~~	~~country~~	~~code~~	**capital**	~~area~~	~~population~~
			16 rows !!					

The elimination of columns is called the projection operation, by the 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:

=	<>
<	>
=<	>=

- Comparisons can be logically combined by the three basic operators of propositional logic (called junctors), written in keyword notation:

  AND OR NOT

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

- The purpose of the WHERE-part is to select those rows, which satisfy the condition for inclusion into the answer table.

Theory of propositions and their connecting operators: Propositional logic

> A **proposition (statement)** is a sentence (a linguistic entity) of which it is reasonable to say that it is true or false.

(Aristotle, Greek mathematician, 384-322 B.C.)

Examples of elementary statements:

• $5 < 6$	true
• 8 is a prime number	false
• The moon is made of cheese	false

- Compound statements are composed from other statements by means of logical operators (connectors).

- Binary (dyadic) operators of propositional logic:

Conjunction	$\wedge$	"and"	(also: &)
Disjunction	$\vee$	"or"	(also: |)

- Unary (monadic) operator:

Negation	$\neg$	"not"

- Elementary statements as well as other compound statements may be connected via such operators in order to form arbitrarily nested complex propositions, e.g.:

((5 < 6) $\wedge$ (8 is a prime number)) $\vee$ $\neg$ (The moon is made of cheese)

- Operators are syntactical "tools", by which the meaning ("semantics") of compound statements can be derived from the meaning of their parts.

- How this is to be done is determined by so-called truth tables:

e.g. Truth table for conjunction

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

T: true
F: false

- To be read this way: If A is true and B is false, then the statement $A \wedge B$ has the truth value false.

- Truth tables of the binary operators:

or:

A	B	A $\vee$ B
T	T	T
T	F	T
F	T	T
F	F	F

and:

A	B	A $\wedge$ B
T	T	T
T	F	F
F	T	F
F	F	F

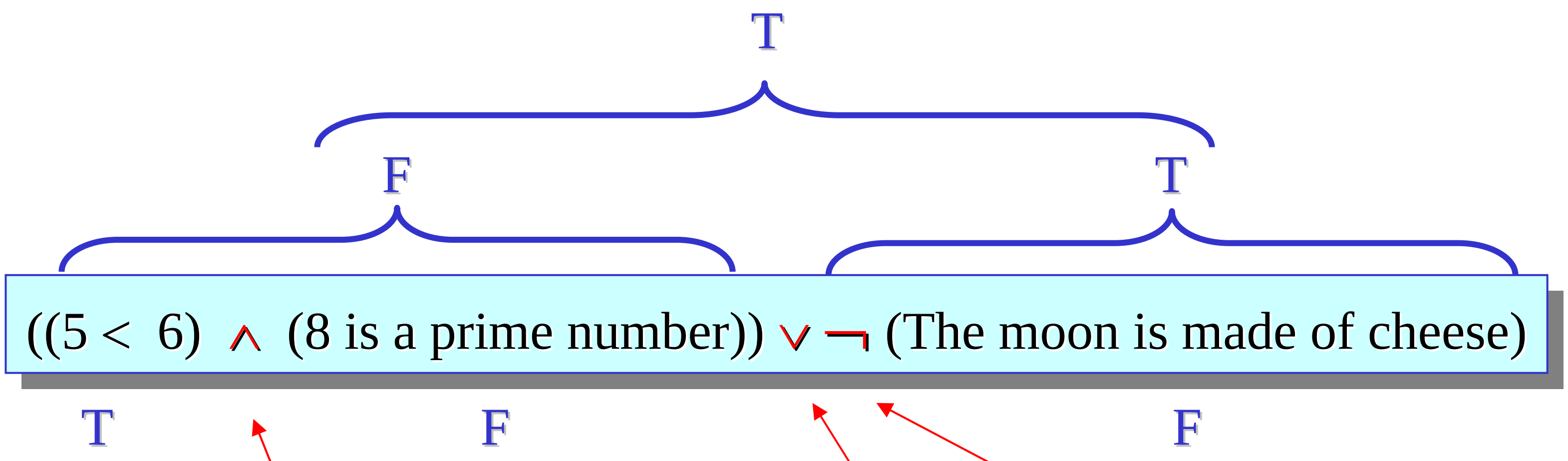
- A bit unusual: Propositional *or* is not exclusive, it is not a real alternative, but "subsumes" propositional *and*.

- Truth table of negation:

not:

A	$\neg$ A
T	F
F	T

Truth values of compound statements can be derived systematically from the truth values of their parts, using the meaning of the operators involved, e.g.:



and:

A	B	A ∧ B
T	T	T
T	F	F
F	T	F
F	F	F

or:

A	B	A ∨ B
T	T	T
T	F	T
F	T	T
F	F	F

not:

A	¬ A
T	F
F	T

Implicit and explicit references to input tables

- In the example query, three names are mentioned (capital, year , city) which refer to certain columns of the two input tables cities and countries. If you remember the schema of each table, you know which column comes from which table. SQL remembers!

```
SELECT    capital, year

FROM      cities, countries

WHERE     year  >= 2000 AND
          city=capital ;
```

- By prefixing column names with their „parent table“ name, you can make these references explicit in a query. Doing so helps readers unfamiliar with the schema. Access always uses this style when generating SQL from QBA queries:

```
SELECT    countries.capital, cities.year
FROM      cities, countries
WHERE     cities.year >=  2000 AND
          cities.city = countries.capital ;
```

- In the original version of the example, *population* was used instead of year. However, both tables have a *population* row! Thus, even the SQL system does not know to which row I am actually referring to – the one in *cities*, or the one in *countries*:

```
SELECT    capital, population

FROM      cities, countries

WHERE     population >= 1000 AND
          city=capital ;
```

?

- For resolving ambiguities like this, it is even necessary to use table names as prefixes in order to explicitly determine the intended reference table – mixing implicit and explicit referencing is possible:

```
SELECT    capital, cities. population
FROM      cities, countries
WHERE     cities. population >=  1000 AND
          city = capital ;
```

- If you prefer, you can introduce shorthands – or even other names – for refering to the input tables in the from clause, e.g. X for cities and Y for countries (just like variables in mathematical formulas):

```
SELECT    Y.capital, X.population
FROM      cities  AS  X, countries AS  Y
WHERE     X.population >=  1000
          AND X.city = Y.capital;
```

- Such shorthands (or alternative names) are called alias names (or aliases for short). They are declared in the FROM-part using the keyword AS.

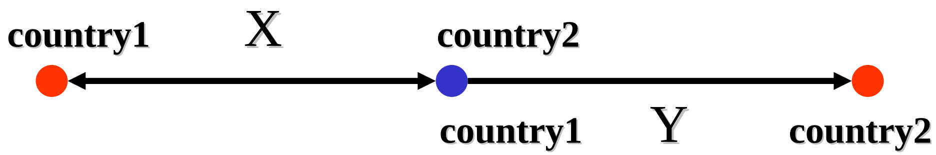
- You can mix styles of referencing: explicit (table name or alias name) or implicit, as long as it is possible to uniquely determine which column refers to which table, e.g.:

```
SELECT    CT.capital, population
FROM      cities, countries AS  CT
WHERE     population >=  1000
          AND city = CT.capital;
```

- Using aliases is unnecessary in most cases – however, you are advised to use them (despite the extra effort) whenever the implicit references 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 un-avoidable to use aliases in order to resolve possible ambiguities, e.g.:

```
SELECT   X.country1, Y.country2
FROM     common_border AS X, common_border AS Y
WHERE    X.country2 = Y.country1
```



- Without the variables it would be unclear, which of the three adjacent countries is actually meant, as they have identical column names. Using the aliases as prefix to the column names resolves the ambiguity.

- In order to properly understand further fundamental operators in SQL, it is necessary to get a basic idea of a particular variant of mathematical set theory, called relational algebra.

- The mathematical concept of a **set** is of fundamental importance for almost every area of computer science.

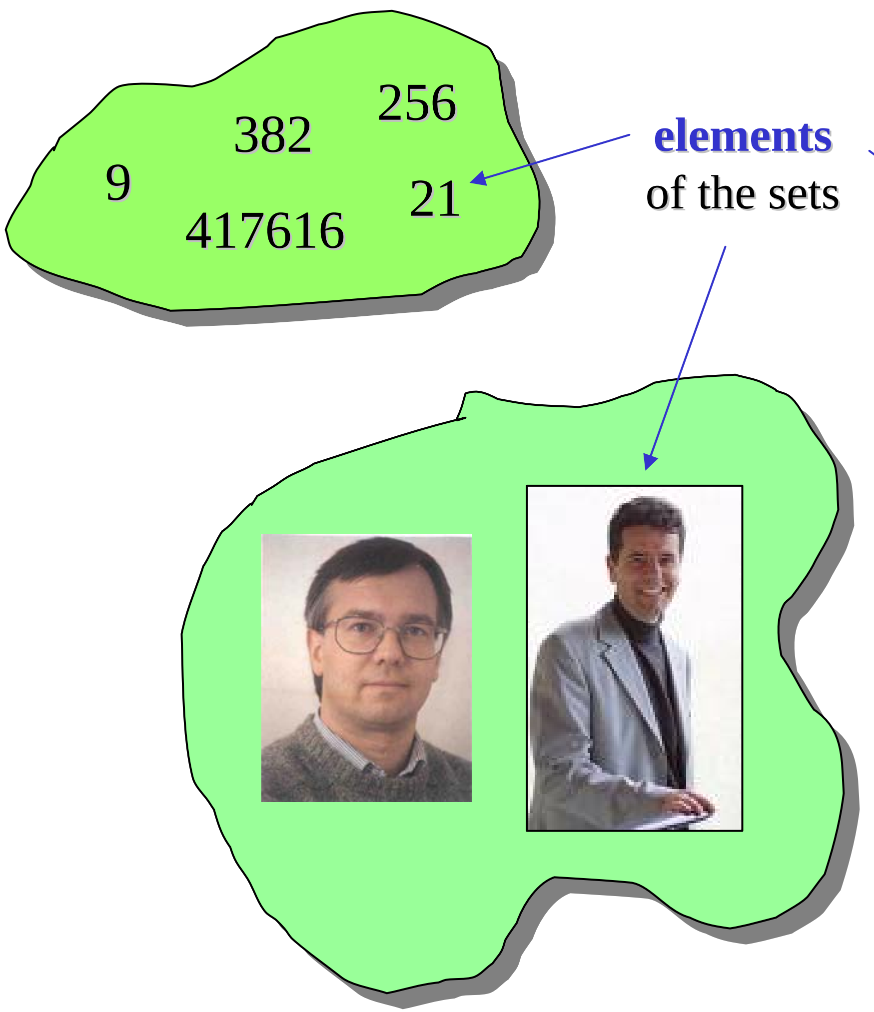
- The notion of a set is "defined" in an informal way, as is common practice in mathematics (by intuition, “naive set theory").

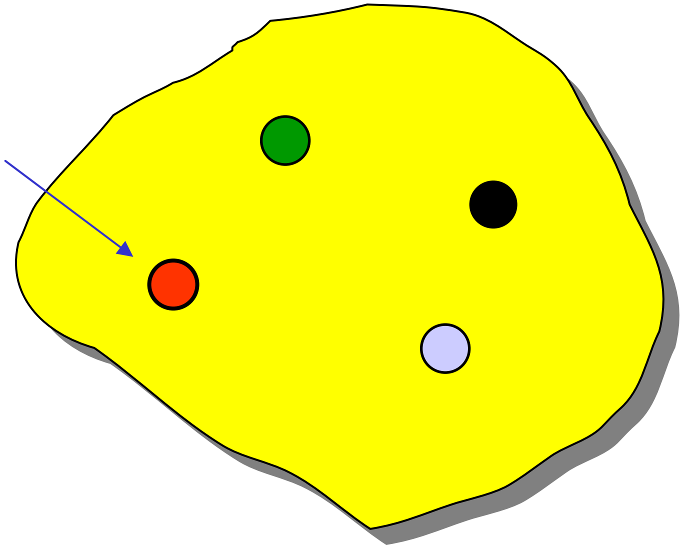
> "A set is a collection into a whole of definite, distinct objects of our perception or our thought."

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

- Sets are composed of elements (Cantor’s “distinct objects”). In a set, each element appears exactly once (i.e., there are no duplicates). The order in which elements appear does not matter (i.e., sets are unordered).

Almost **everything** can be in a set.

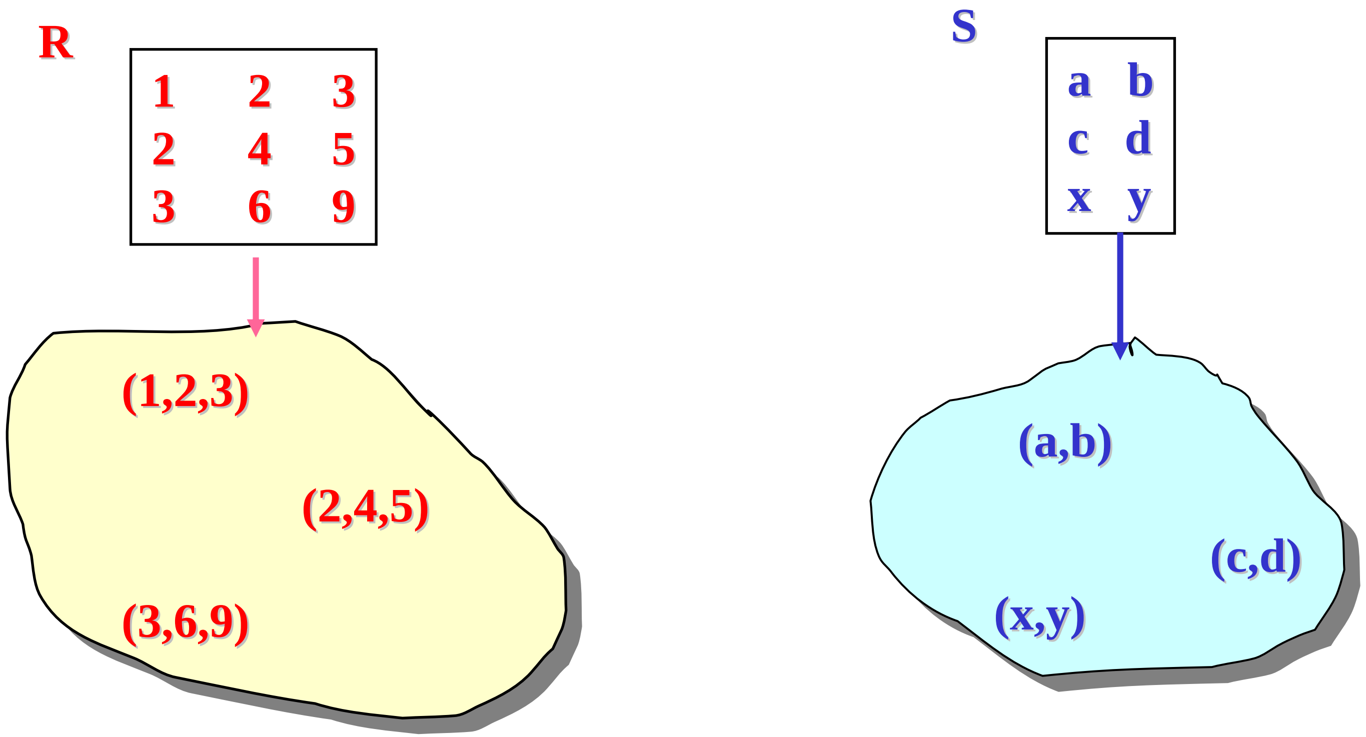




Sets are rather collections of abstract ideas or individuals ("objects") than shopping bags.

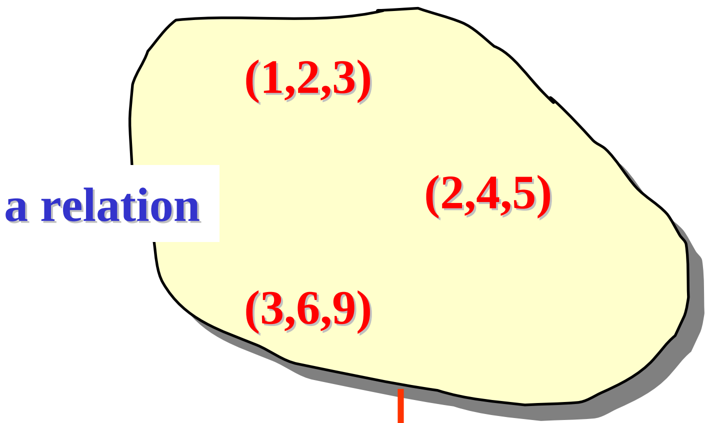
Database relations (visualized in tabular form) can be regarded as sets, too. Their elements are called **tuples** in mathematics:

R is a set of 3-tuples (or triples), S is a set of 2-tuples (or pairs).

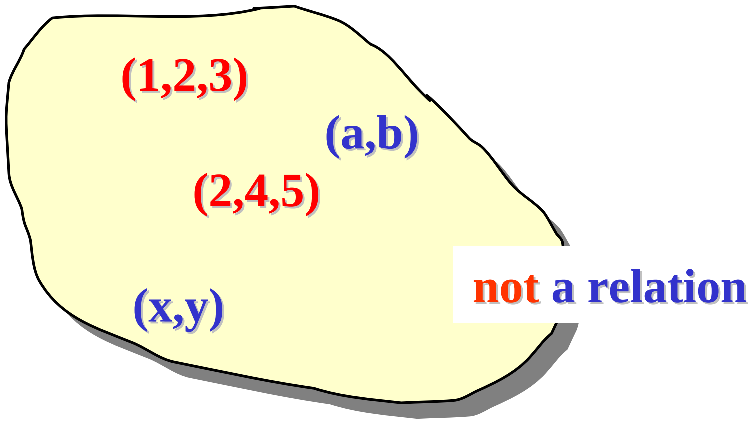


- Not every set of tuples is a relation, though!

- Relations are homogeneous, i.e., contain elements (tuples) of the same kind:
  - same number of components (called arity in mathematics)
  - same type of components at respective positions

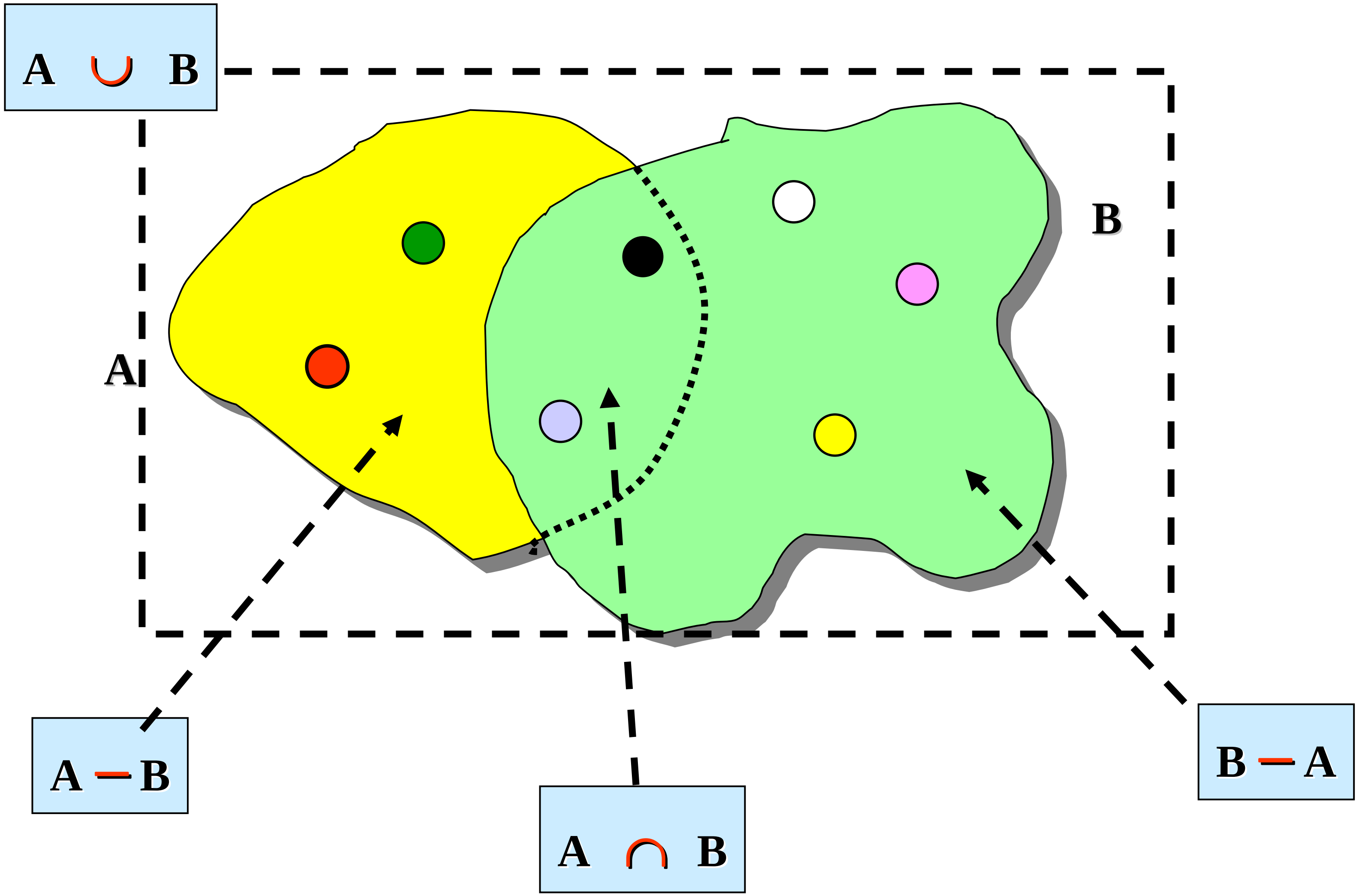


1	2	3
2	4	5
3	6	9



1	2	3
2	4	5

a	b
x	y


A ∪ B
A
B
A − B
A ∩ B
B − A

- There are three basic operators for combining sets in general, i.e. for constructing a new result set from the elements of two input sets A and B:

union	$A \cup B$	contains all elements of A together with those in B
intersection	$A \cap B$	contains all elements from A which are in B, too.
difference	$A - B$	contains all elements from A which are not in B

- If applying them to sets that are relations, we have to make sure that both input sets are „of the same kind" (i.e. have the same arity and type), otherwise the result set would not be a proper relation again. Thus, the three set operators „behave" a bit differently if used as relational algebra operators.

- In SQL, there are keyword for the three operators, differing slightly from their names in set theory:

UNION INTERSECT MINUS

R

1	2	3
2	4	5
3	6	9

Q

1	2	3
2	6	7
3	6	9
4	1	5

R **UNION** Q

1	2	3
2	4	5
3	6	9
2	6	7
4	1	5

duplicates eliminated!

R **INTERSECT** Q

1	2	3
3	6	9

R **MINUS** Q

2	4	5

Q **MINUS** R

2	6	7
4	1	5

R

A	B	C
1	2	3
2	4	5
3	6	9

Q

A	B	C
1	2	3
2	6	7
3	6	9
4	1	5

```
(SELECT   A, B   FROM   R)
      UNION
(SELECT   A, B   FROM   Q)
```

A	B
1	2
2	4
3	6
2	6
4	1

In SQL, set operators may be applied to any two expressions denoting relations of the same kind.

- Each clause of a SELECT-FROM-WHERE block corresponds to an operator of relational algebra, too:

projection	$\pi_{A, B}$	eliminates all columns except A and B
selection	σ_{cond}	eliminates all rows except those satisfying condition *cond*
product	R × S	set of all combinations of tuples from R and S

- Example:

```
SELECT    capital, cities.population
FROM      cities, countries
WHERE     cities.population  >= 1000 AND
          city=capital ;
```

$\pi_{\text{capital, citites.population}}$

cities × countries

$\sigma_{\text{cities.population >= 1000 AND city = capital}}$

- The order of evaluation matters in SQL: 1) product 2) selection 3) projection

- Don‘t be fooled by SELECT corresponding to the projection part rather than selection!

The JOIN operator in SQL and RA

- There is a special notation for situations, where two tables connected via a product are logically linked via a selection condition involving one column from each table, too:

```
SELECT    countries.capital, cities.population
FROM      cities, countries
WHERE     cities.population  >= 1000 AND
          cities.city=countries.capital ;
```

join symbol in RA:

$\bowtie$

- Such linking conditions are called join conditions, and the operation is called a **join** in RA. A join may appear in the FROM part in place of the comma (indicating product). The join condition is moved to the FROM part, too:

```
SELECT    countries.capital, cities.population
FROM      cities  JOIN  countries ON cities.city=countries.capital
WHERE     cities.population  >= 1000;
```

- Note that JOIN is allowed in a FROM part only, not as an independent operator such as, e.g. UNION. – In Access, this form of join is called the INNER JOIN.

- If only columns from one of the two tables involved is required in the result table of a query, the other table can be accessed in an inner block nested in the WHERE part:

```
SELECT    countries.capital
FROM      cities, countries
WHERE     cities.population  >= 1000 AND
          cities.city=countries.capital ;
```

↓

```
SELECT    countries.capital
FROM      countries
WHERE     countries.capital IN
               (SELECT cities.city
                FROM   cities
                WHERE  cities.population  >= 1000);
```

- The keyword **IN** (connecting a column name and a subquery) corresponds to the operator $\in$ representing the ***is element of*** relationship between an object and a set in set theory.

- In Access, the MINUS operator expressing set difference is unknown. However, an identical result can be obtained using block nesting and the NOT IN operator:

```
(SELECT  city     FROM cities)
     MINUS
(SELECT  capital FROM countries)
```

Which cities are no capitals?

```
SELECT  city
FROM    cities
WHERE  city  NOT IN  (SELECT capital FROM countries)
```

- Apart from being a bit more intuitive, this formulation shows more explicitly that set difference is not a symmetric operation: R MINUS S $\neq$ S MINUS R

- In addition, the nesting style indicates clearly that the rows „surviving" in the difference all come from the left operand table.

- Access does not support the INTERSECT operator either, as it can be simulated by means of a join on all columns of the two tables returning only those rows that have identical values in all of these columns:

```
(SELECT  city      FROM cities  WHERE  population > 1000)
    INTERSECT
(SELECT  capital FROM countries)
```

```
SELECT   city
FROM     cities  JOIN countries  ON  city = capital
WHERE    cities.population > 1000
```

- In this case, the order of the input tables does not matter. The above is equivalent to:

```
SELECT   city
FROM     countries  JOIN cities  ON  city = capital
WHERE    cities.population > 1000
```

- Within a SELECT-FROM-WHERE block, two tables can be combined in two ways:
  - by simply listing the tables in the FROM part separated by a comma: product
  - by explicit JOIN in connection with a join condition in the ON part
- Two independent subqueries can be combined using one of the three set operators in infix notation: UNION, INTERSECT, and MINUS.
- A SELECT-FROM-WHERE block can be nested within the WHERE part of another block by means of the (NOT) IN operator, comparing a column in the outer block with a column in the SELECT part of the inner block.
- JOIN-ON is not strictly necessary, as it can be expressed by product and selection.
- MINUS can be expressed using NOT IN and nesting.
- INTERSECT can be expressed by a JOIN on all columns.
- Thus, SELECT-FROM-WHERE blocks with IN-style nesting and UNION are sufficient for expressing almost all multi-table queries (this is the SQL subset supported by Access).

- SQL answer tables are no relations in the sense of set theory and relational algebra: Projection and union may produce duplicate answers which are <u>not</u> automatically eliminated in SQL!

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

```
SELECT DISTINCT   Name
FROM        countries
WHERE       Name in (SELECT country
                     FROM cities
                     WHERE population > 1000)
```

→

Name
France
Germany
~~Germany~~
. . .

- It is recommendable to <u>always</u> 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!

Aggregate functions

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

COUNT	Cardinality
SUM	Sum
AVG	Average
MAX	Maximum
MIN	Minimum

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

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‘
```

advisable in order to have a column name for each column in the answer table

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‘ )
```

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

- Syntactic extension: GROUP BY-clause (after SELECT-FROM-WHERE)

- 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)

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

  e.g.:

```
SELECT      P.Rank,  AVG( P.Age )  AS  AvgAge
FROM        professors  AS  P
GROUP BY    P.Rank
```

- If no explicit grouping is specified, the entire table is assumed as one big „group“.

Illustration with example data:

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

Name	Rank	Age
Jim	C4	43
John	C3	33
~~Ken~~	~~C4~~	~~57~~
Lisa	C4	39
Tom	C2	32
Eva	C3	36

GROUP BY

Name	Rank	Age
Jim	C4	43
Lisa	C4	39
John	C3	33
Eva	C3	36
Tom	C2	32

AVG

Rank	AvgAge
C2	32.0
C3	34.5
C4	41.0

- Sorting of the result table can be specified at the end of a SELECT-block (after GROUP BY, if present at all)

- Example:

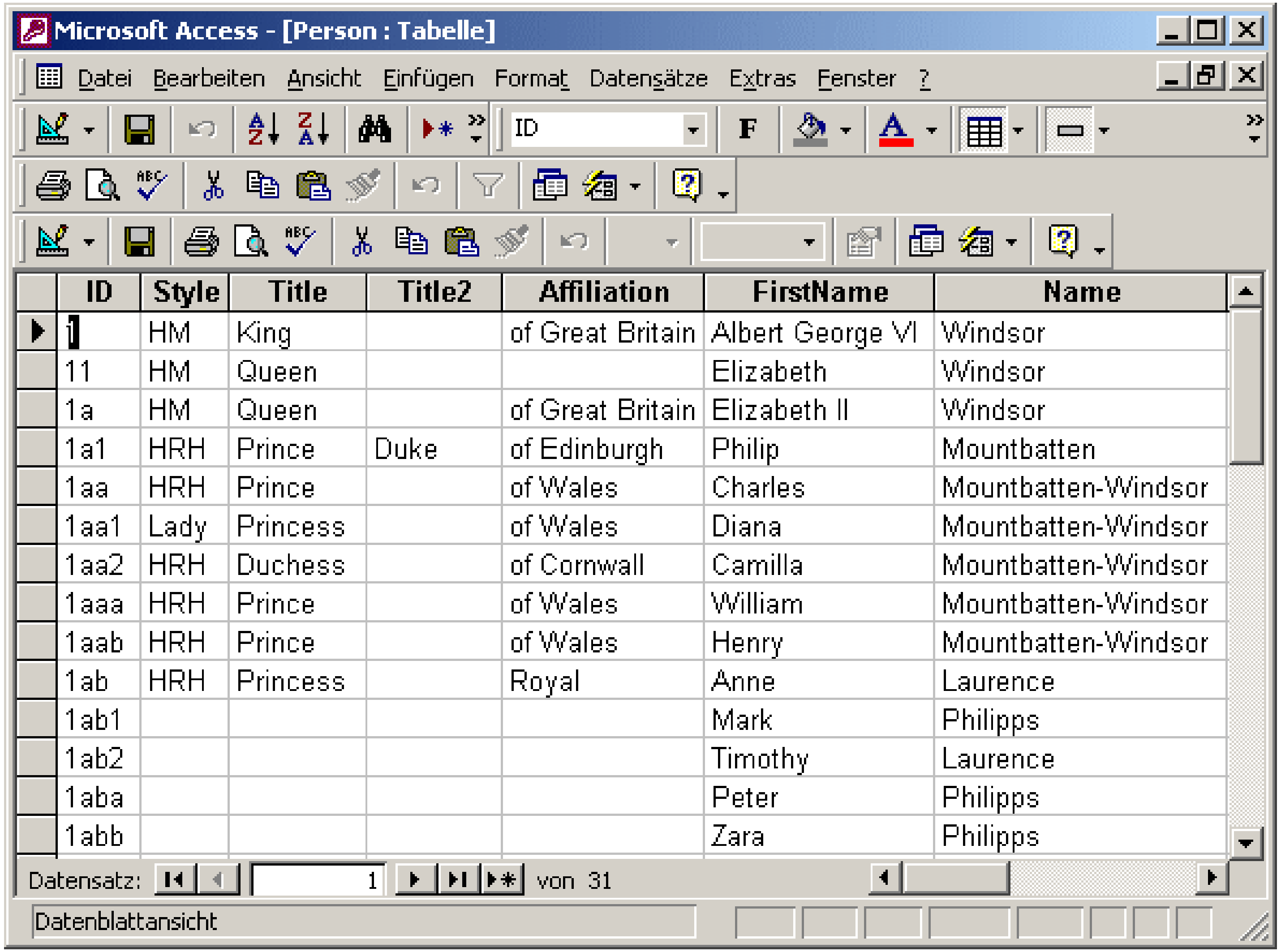
```
SELECT   X.Rank, X.Salary
FROM     professors AS X
ORDER BY X.Rank DESC ,
         X.Salary ASC
```

- „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.

In tables of a relational database there might be numerous empty cells – for various reasons and with different meanings. In SQL, there is the feature of a **null value** associated with this.



ID	Style	Title	Title2	Affiliation	FirstName	Name
1	HM	King		of Great Britain	Albert George VI	Windsor
11	HM	Queen			Elizabeth	Windsor
1a	HM	Queen		of Great Britain	Elizabeth II	Windsor
1a1	HRH	Prince	Duke	of Edinburgh	Philip	Mountbatten
1aa	HRH	Prince		of Wales	Charles	Mountbatten-Windsor
1aa1	Lady	Princess		of Wales	Diana	Mountbatten-Windsor
1aa2	HRH	Duchess		of Cornwall	Camilla	Mountbatten-Windsor
1aaa	HRH	Prince		of Wales	William	Mountbatten-Windsor
1aab	HRH	Prince		of Wales	Henry	Mountbatten-Windsor
1ab	HRH	Princess		Royal	Anne	Laurence
1ab1					Mark	Philipps
1ab2					Timothy	Laurence
1aba					Peter	Philipps
1abb					Zara	Philipps

- SQL offers a predefined, universal null value, intended to represent unknown or missing information in a systematic way. The keyword NULL represents such values.

- 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 values in SQL: Value exists, but is unknown!

- Thus: Nulls are called „values“! Each two occurrences of a null value represent different „real" values presently (still) unknown.

- However: Nulls themselves don‘t have a type but always take the type of the resp. column under consideration.

- In queries, emptiness of a particular cell can be tested by using the keyword NULL. Note that NULL does not represent „the“ null value (as there are infinitely many of them), but simply serves as a test condition applied to a particular field of a particular row.

- One immediate consequence of this particular interpretation of empty cells alias null values is that NULL may not be used in comparisons, i.e. the following are not allowed in SQL:

  Name = NULL   Age > NULL

- Instead, there is a special test operator IS which can be used to express checks for „nullness“ (i.e. emptiness of cells), e.g.:

  Name IS NULL   Age IS NOT NULL

- Moreover, you cannot join rows on empty cells, as two different occurrences of a null value (in two different rows) are different by definition, and thus cannot be identified (or compared).

- Aggregate functions ignore NULL „on purpose“!

person

Name	Age
Jim	33
Tom	NULL

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

AVG(Age): 33

- Access offers a built-in function (nz), though, for transforming all NULLs in a field by 0 when used in a query, e.g. nz([age],0) (nz stands for null-to-zero)

- 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

  Whenever NULL occurs during evaluation, UNKNOWN may result, depending on the logical operators involved (details are beyond the scope of this chapter).

- Example: If A=3, B=4 and IS NULL C, then . . .
  - A > B AND B > C results in FALSE
  - A > B OR B > C results in UNKNOWN

- Automatic generation of null values when using an OUTER JOIN-operator:

  { LEFT | RIGHT | FULL } [OUTER] JOIN

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

- Example:

p

A	B
1	2
1	3
2	1

q

B	C
2	5
3	4
5	3

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

A	p.B	q.B	C
1	2	2	5
1	3	2	5
1	3	3	4
2	1	NULL	NULL
NULL	NULL	5	3

Always contains INNER JOIN as subtable !

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

- <u>Example</u>:

p (left)

A	B
1	2
1	3
2	1

q (right)

B	C
2	5
3	4
5	3

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

A	p.B	q.B	C
1	2	2	5
1	3	2	5
1	3	3	4
2	1	NULL	NULL
~~NULL~~	~~NULL~~	~~5~~	~~3~~

- In Access-SQL: Only LEFT JOIN and RIGHT JOIN are supported, no FULL OUTER JOIN; "OUTER" is omitted.

- 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:

city_at_river

City	River

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

city_at_river

City	River

empty table not containing any row

- 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 ;
```

Name
Paris
London

if yes:
non-empty answer table

Name

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 )
```

- 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 the general sense refers to any kind of change
  - UPDATE in SQL means column value replacement only

- Recommendation: Try update statements in Access and observe how action queries of type insertion/modification/deletion are automatically transformed into SQL statements, and vice versa.

- Format of insertions:

```
INSERT INTO <table-name> [ ( <list-of-columns> ) ] <table-expression>
```

- Two variants:
  - Direct reference to one or more rows to be inserted, e.g.

Notation of a tuple in SQL

keyword for direct specification of rows

```
INSERT INTO professors (Name, Rank, Department)
       VALUES ( ‚Cremers‘, ‚C4‘, ‚III‘)
```

  - Indirect identification of the rows to be inserted via a query, e.g.

```
INSERT INTO professors
       SELECT *
       FROM   researchers AS R
       WHERE  R.qualification = ‚PhD‘
```

- Format of modifications:

```
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:

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

- Example:

```
UPDATE   professors
SET      Name = ‚N.N.‘
WHERE    Dept = ‚II‘
```

← assignment (action)
← condition (test in the „old" state)

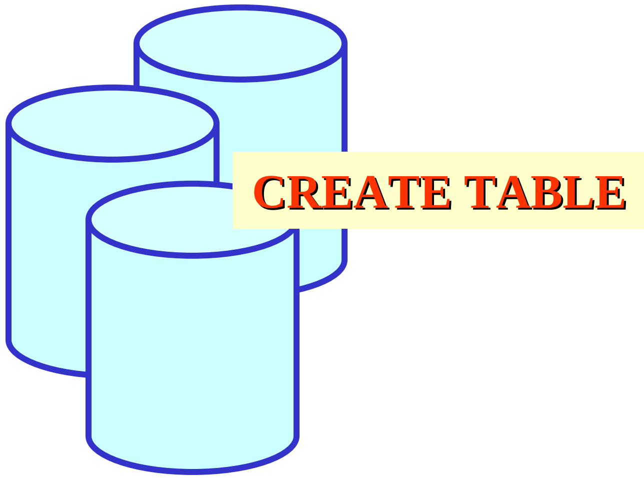
- Quite similar: Deletions

```
DELETE FROM  <table-name>
[ WHERE  <conditional-expression> ]
```

Foundations of Information Management (WS 2008/09)

Data Definition in SQL

– 3.2 –



- 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

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

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

Unique within one and the same schema

- "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:

Table name

Table elements

```
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
     ) ;
```

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

Column definitions

```
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
    ) ;
```

Table 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

```
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),
   . . .
   )
```

Syntax of column definitions:

<column-name> <data-type> [<column-constraints>]

unique within the same table (refers to <column-name>)

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

```
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

Syntax of column constraints:

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

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> ) ]
```

- 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:

      Type number(15) CHECK (> 0 AND < 35),

    - table constraint:

      CHECK (Type > 0 AND Type < 35)

- The condition part of such a CHECK constraint has to be satisfied in each admissible (legal, consistent) state of the database.

- UNIQUE-option: definition of a key (or: candidate key)
  - single-column key:
    in a column definition: <column-name> . . . UNIQUE
  - multi-column key:
    separate UNIQUE-clause as table constraint:
    UNIQUE (<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: <column-name> . . . NOT NULL

- 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)

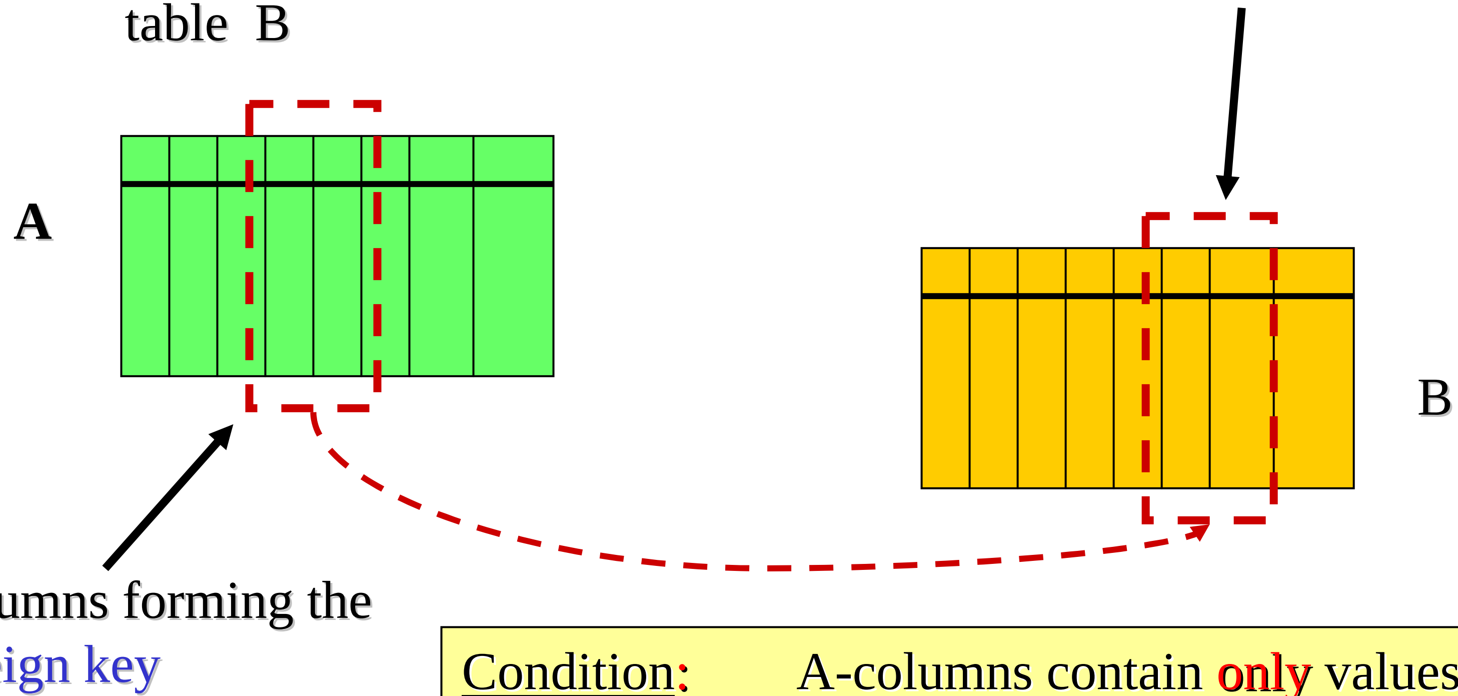
- Not a real „constraint", but rather similar in syntax:
  Declaration of a default value for columns of a table:
  Value which is automatically inserted if no explicit value is given
  during the insertion of a new row, e.g.

Type number(15) DEFAULT 0

- 2nd 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 a another („foreign“) table B



Condition: A-columns contain only values actually occurring in the referenced B-column(s)!

Syntax of the corresponding constraint (as table constraint):

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

If „target columns“ are missing: primary key assumed

e.g.:

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

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

abbreviated form as column constraint

b_1 references a_1

- 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

Assertions

- 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 table.
- 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    NOT EXISTS
           ( SELECT  *  FROM  professor
             WHERE   Name  NOT  IN  ( SELECT  Teacher
                                      FROM     courses ) ;
```

- 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!

b-it

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

- <u>Motivation</u>: 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!

<u>e.g.</u>:

C_1: „Each course is given by a professor!"

C_2: „Each professor has to give at least one course!"

course ⇄ professor

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

INSERT INTO professor
INSERT INTO course

Each intermediate state would be inconsistent: No sequence possible !

- 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:

  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 !

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

Views

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

Name

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

Query

- 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 RECUR-SIVE 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 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:

e.g.:

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

```
SELECT   Name
FROM     C4-profs
WHERE    Dept = ‚III‘
```

➡

```
SELECT   X.Name
FROM     ( SELECT Name, Dept
           FROM    professors
           WHERE Rank = ‚C4‘ )     AS X
WHERE    X.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 within the special area of deductive database research.

- 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:

Trigger

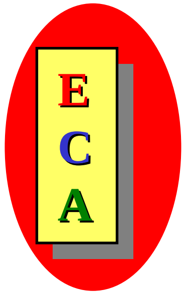
- Other notion for trigger: Active rule
- Name of a DBMS supporting triggers: Active DBMS
- Name of the corresponding research area: Active databases

- 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.).

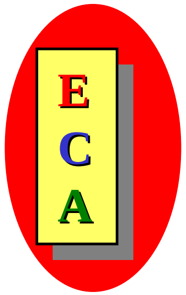
- In the new SQL3-Standard (1999) triggers have been standarized for the first time.

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



"event"
"condition"
"action"

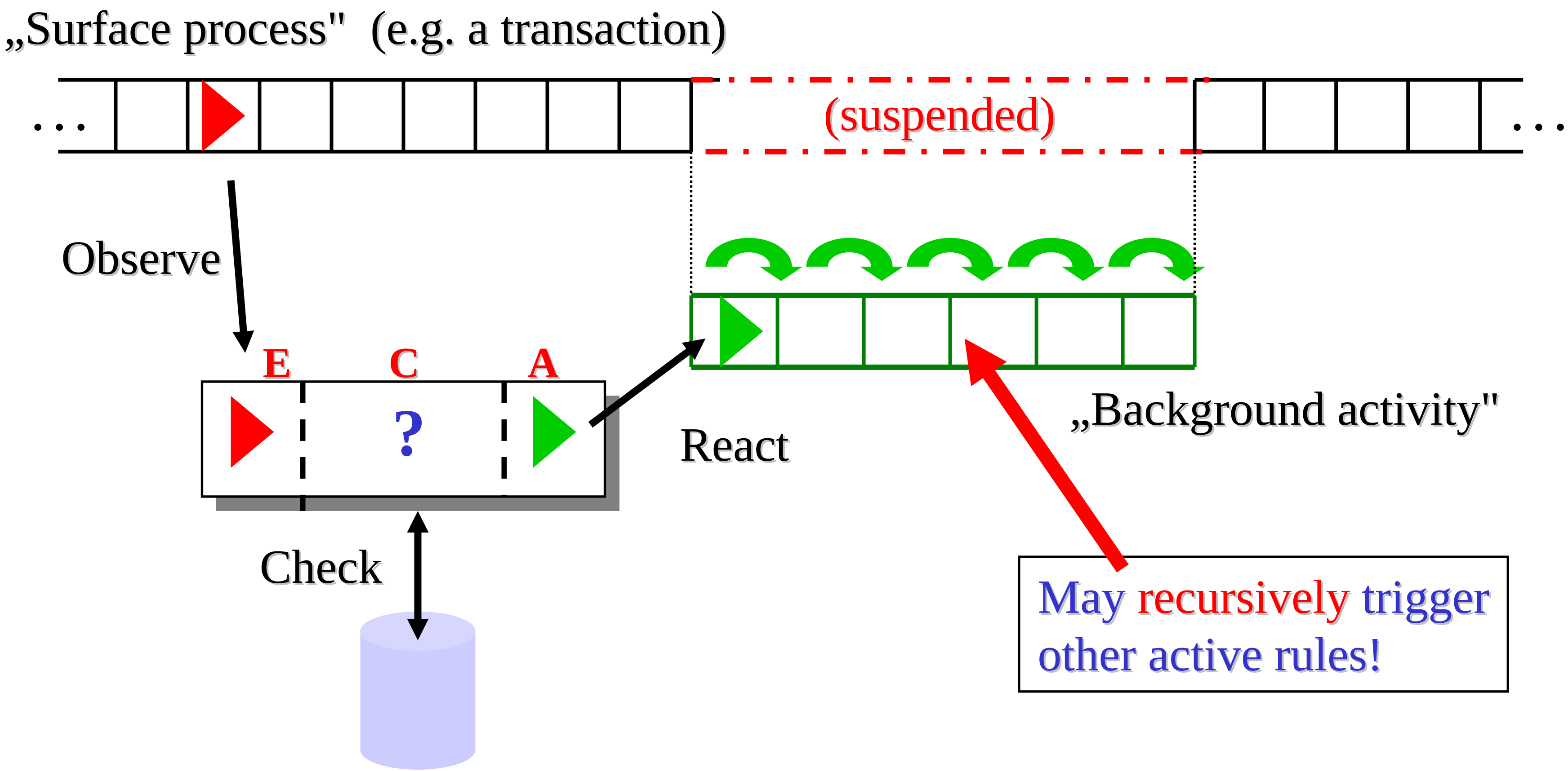
- Example of an ECA-rule (in pseudo-code):




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

General meaning of an active rule:

Additional, automatically triggerd „background activity"



Example of an SQL trigger:

Trigger name

Time of triggering

Triggering update

```
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
```

Condition

Action

„Transition variable"