5. Languages of relational databases

5.1. Tables of the illustrative example - Bank

- the relational DB is perceived by the user as a collection of tables

Client

clientNo	name	street	town
440726/0672	Jan Novák	Cejl 8	Brno
530610/4532	Petr Veselý	Podzimní 28	Brno
601001/2218	Ivan Zeman	Cejl 8	Brno
510230/048	Pavel Tomek	Tomkova 34	Brno
580807/9638	Josef Mádr	Svatoplukova 15	Brno
625622/6249	Jana Malá	Brněnská 56	Vyškov

Account

accNo	balance	clientNo	branch
4320286	52000	440726/0672	Jánská
1182648	10853	530610/4532	Palackého
2075752	126350	440726/0672	Palackého

Branch

name	assets
Jánská	10000000
Palackého	5000000

Transaction

accNo	transNo	date	amount
4320286	1	10.10.1998	3000
4320286	2	12.10.1998	- 5000
2075752	1	14.10.1998	- 2000
2075752	2	14.10.1998	10000

5.2. SQL

5.2.1. Introduction

- **Language history**
  - **1975 - Sequel in System R**
  - **1986 - ANSI standard, 1986 - ISO standard - SQL/86, dominant role of SQL dialect by IBM (DB2)**
  - **1989 - Integrity Addendum - SQL/89,**
  - **1992 - SQL/92, three levels of compliance (Entry/Intermediate/Full)**
  - **1996 – part concerning persistent stored modules (PSM/96)**
  - **1999 – SQL1999 – object-relational features**
  - **2002 – multimedia and data mining support**
  - **2003 – SQL2003 – OLAP support, XML support (SQL/XML)**
  - **2006 – extended support of XML in SQL/XML (XQuery)**
  - **2008 – finished SQL/XML and other parts (~4000 p.).**
  - **SQL/MM – parts: Framework, Full Text, Spatial, Still image, Data mining**

- **many SQL dialects SQL, the base is SQL/92 (at least Entry) + additional extensions**

Adapted from http://www.jcc.com/sql.htm

Part	Explanation
Part 1 - SQL/Framework	Information common to all parts of the standard. (90 p.)
Part 2 - SQL/Foundation	Data definition and data maniputlation syntax and semantics, SQL embedded in non-object programming languages, od 2003 i SQL/OLAP. (1366 p.)
Part 3 - SQL/CLI	(1995) Call Level Interface: Corresponds to ODBC. (405 p.)
Part 4 - SQL/PSM	(1996) Persistent Stored Modules: Stored routines, external routines, and procedural language extensions to SQL. (191 p.)
Part 9 - SQL/MED	Management of External Data: SQL access to non-SQL data sources (files). (486 p.)
Part 10 - SQL/OLB	(1999) Object Language Bindings: Specifies the syntax and semantics of embedding SQL in Java™ (SQLJ). (415 p.)
Part 11 - SQL/Schemata	(2003) Information and Definition Schemas. (298 p.)
Part 13 - SQL/JRT	(2003) Java Routines and Types: Routines using the Java™ Programming Language (Persistent Stored SQLJ). (208 p.)
Part 14 - SQL/XML	(2003) SQL and XML. (447 p.)

- **There are three possible contexts for SQL (binding styles):**

  - direct SQL
  - embedded SQL
  - module language
- Main statement categories:
  - database objects definition (DDL – Data Definition Language)
  - data manipulation (DML – Data Manipulation Language)
    - for direct SQL
    - for embedded SQL
  - authorization (access control)
  - data integrity
  - transaction control

5.2.2. Data definition

- **Basic statements:**
  - **CREATE - creating a database object**
  - **DROP – removing a database object**
  - **ALTER – changing properties of a database object**
- **Creating a base table (really existing in the database)**

```
CREATE TABLE base_table_name
  (column_def, …
  [table_int_con_list]
  )
```

→ it creates a new empty table and stores its description into a system catalogue

- **Column definition**

```
column_name type [default_value] [column_int_con_list]
```

- **Integrity constraint definition (int_con above)**

  Integrity constraints are constraints put on the values in columns of the table so that the data integrity is not violated.

- **Repeating important concepts related to table integrity constraints (more formally see chapter 3 Relational data model):**

Candidate key **– column(s) of a table the value of which must be unique in the table (and in RM also irreducible).**

Primary key **– one of candidate keys that will be used to „address“ rows of the table. It must satisfy constraints for candidate keys and the value must not be empty/missing.**

Alternative key **– a candidate key that is not a primary one.**

Foreign key **– column(s) the value of which must match a value of the referenced candidate key in the referenced table (in dle RM the value can also be null). We use it to create links (by means of explicit values) between rows of two, not necessarily different, tables. The correspondence of a foreign key and the referenced candidate key values is referred to as** ***referential integrity*****.**

- **Integrity constraints (declarative)**

```
[CONSTRAINT name] constraint
```

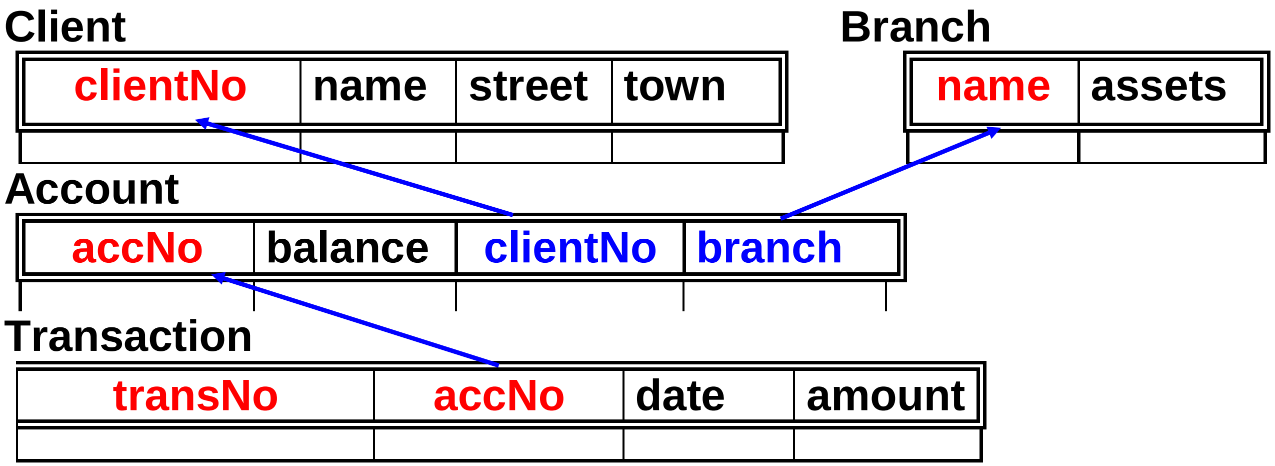
  - **Column constraints**

```
NULL, NOT NULL
CHECK (conditional expression)
PRIMARY KEY
UNIQUE
FOREIGN KEY REFERENCES table [(col_name)] [event
ref_action]
```

  - **Table constraints**

```
PRIMARY KEY (col_name, …)
UNIQUE (col_name, …)
FOREIGN KEY (col_name, …) REFERENCES
   tabulka [(col_name, …)] [event ref_action]
CHECK (conditional expression)
```

Ex)



Client

clientNo name street town

Branch

name assets

Account

accNo balance clientNo branch

Transaction

transNo accNo date amount

```
CREATE TABLE Account
   (accNo       NUMERIC(7,0),
   balance      NUMERIC(10,2) DEAFULT 0,
   clientNo     CHAR(11) NOT NULL,
   branch       CHAR(20) NOT NULL,
   CONSTRAINT PK_account PRIMARY KEY (accNo),
   CONSTRAINT FK_account_clientNo FOREIGN KEY
     (clientNo) REFERENCES Client ON DELETE CASCADE,
   CONSTRAINT FK_account_branch FOREIGN KEY (branch)
     REFERENCES Branch)
```

➢ Data types

- *string:*

CHARACTER(n), CHARACTER VARYING(n), BIT(n), BIT VARYING(n)

- *numeric*

- NUMERIC(p, q), DECIMAL(p, q),

- INTEGER, SMALLINT, FLOAT(p), REAL, DOUBLE PRECISION

- *date and time:* DATE, TIME, TIMESTAMP
- *interval:* INTERVAL

SQL/99 introduces additional pre-defined data types, e.g.:

- *string:*

NATIONAL CHARACTER(n), NATIONAL CHARACTER VARYING(n), CHARACTER LARGE OBJECT, BINARY LARGE OBJECT

- *boolean:*

BOOLEAN

Note: There are abbreviations for the data types, e.g. CHAR, VARCHAR, NCHAR, NVARCHAR, CLOB, BLOB.

SQL dialects provides additional types, e.g. NUMBER by Oracle, TINYINT in MySQL.

Okomentoval(a): [t1]: Rozdíl oproti NUMERIC – DECIMAL musí mít délku minimálně p číslic (skutečná délka je implementačně závislá), NUMERIC přesně p.

Okomentoval(a): [C2]: INTEGER a SMALLINT mohou být reprezentovány binárně nebo dekadicky.

Okomentoval(a): [C3]: p je přesnost vyjádřená celkovým počtem číslic.

Okomentoval(a): [C4]: U Oracle DATE obsahuje i čas.

Okomentoval(a): [C5]: Časové typy mohou obsahovat pole YEAR, MONTH, DAY, HOUR, MINUTE,SECOND, lze specifikovat i časovou zónu vymezenou posuvem.

Okomentoval(a): [C6]: Intervaly mohou být dvou typů: year-month a day-time, včetně podmnožin, tj. např. INTERVAL YEAR, INTERVAL HOUR TO MINUTE. Podle toho obsahují příslušná pole.

Okomentoval(a): [C7]: NATIONAL říká, že příslušný datový typ je založený na specifické znakové sadě definované implementací jako „národní sada“. Pro Oracle je to UNICODE AL16UTF16 and UTF8.

Okomentoval(a): [C8]: Na 1 byte.

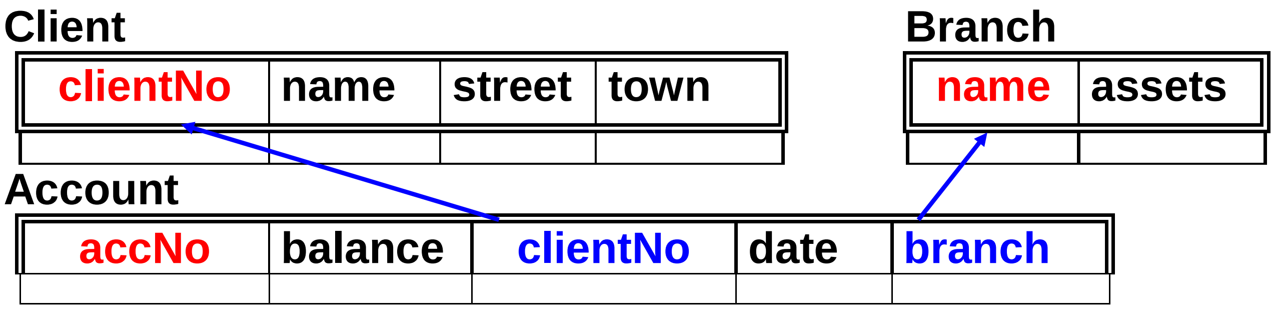
- **Literals**
  - ***string:*** **'string1' for characters, B'001010', X'01AFB0' for bits**
  - ***numeric:*** **12345.67, -25.7E-3**
  - ***date and time:*** **DATE '2005-02-27', TIME '10:00:27.5'**

Note: Oracle stores not only date but also time in a DATE type and there are several formats for date and time. It is possible to use a function TO_DATE, e.g. TO_DATE('98-DEC-25 17:30','YY-MON-DD HH24:MI').

Okomentoval(a): [C9]: U hexadecimálního může být velké nebo malé písmeno.

- **Operations with a table that can violate referential integrity and constraints for primary and alternate keys**

Ex)



	SELECT	INSERT	DELETE	UPDATE
Client	-	PK, AK, NULL	RI Account.clientNo	PK, AK, NULL RI Account.clientNo
Account	-	PK, AK, NULL, RI Client, Branch	-	PK, AK, NULL, RI Client, Branch

⇒ operations DELETE and UPDATE on a referenced table can violate referential integrity

⇒ it is possible to define DBMS behaviour (referential action) for them

- **Referential event - update (ON UPDATE), delete (ON DELETE) <u>of a referenced table</u>**
- **Referential action - NO ACTION, CASCADE, SET DEFAULT, SET NULL**

- Altering a base table definition

```
      ALTER TABLE base_table_name action
```

  - actions: - adding (ADD), dropping (DROP) a column, altering column’s default value (ALTER); adding (ADD), dropping (DROP) an integrity constrain

→ it modifies the table and changes corresponding information in the system catalogue

- Dropping a base table

```
      DROP TABLE base_table_name [RESTRICT|CASCADE]
```

Okomentoval(a): [z12]: Týká se pohledů, omezení FK, triggerů atd.

→ it drops the base table including all information about it in the system catalogue

- Views (VIEW)(see later)
  - tables derived from other tables, they need not exist in the database
- Other typical database objects (not included in SQL/92, some later)
  - ➢ Index

```
   CREATE [UNIQUE]INDEX index_name ON
   base_table_name (col_name [ASC|DESC], … )
```

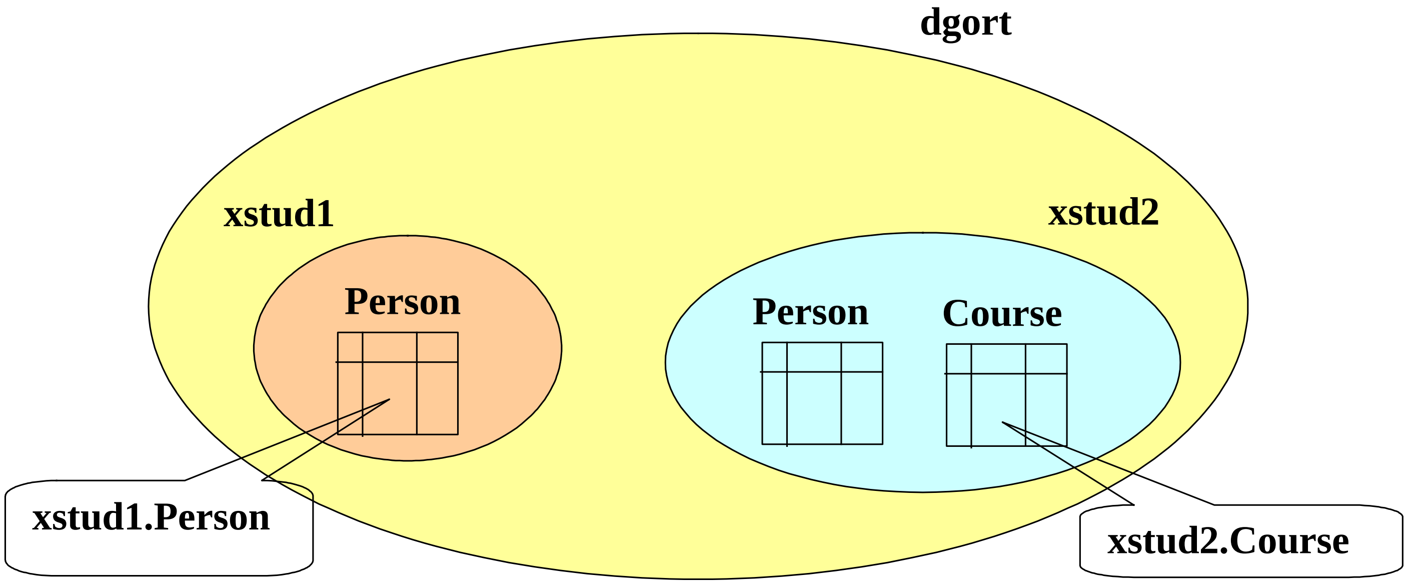
➢ Synonyms

```
CREATE SYNONYM synonym_name FOR table_name
```

- allow to increase location independence

Okomentoval(a): [t13]: Užitečné pro zajištění nezávislosti na kvalifikaci jména tabulky (schéma, uzel sítě).

Ex)



CREATE SYNONYM Other_person FOR xstud1.Person;

CREATE SYNONYM Other_person FOR xstud1.Person@dbgort.fit.vutbr.cz;

- **Sequence generators (sequence, counter, autoincrement)**

Ex) Oracle

```
CREATE SEQUENCE person_seq
START WITH 1000
INCREMENT BY 1;
```

MySQL

```
CREATE TABLE Account (
accNo UNSIGNED INT AUTO_INCREMENT,
… );
```

Oracle 12c

```
CREATE TABLE Osoba (
osobaID INT DEFAULT osoby_seq.NEXTVAL PRIMARY KEY,
… );
CREATE TABLE Osoba (
osobaID INT GENERATED AS IDENTITY PRIMARY KEY,
… );
```

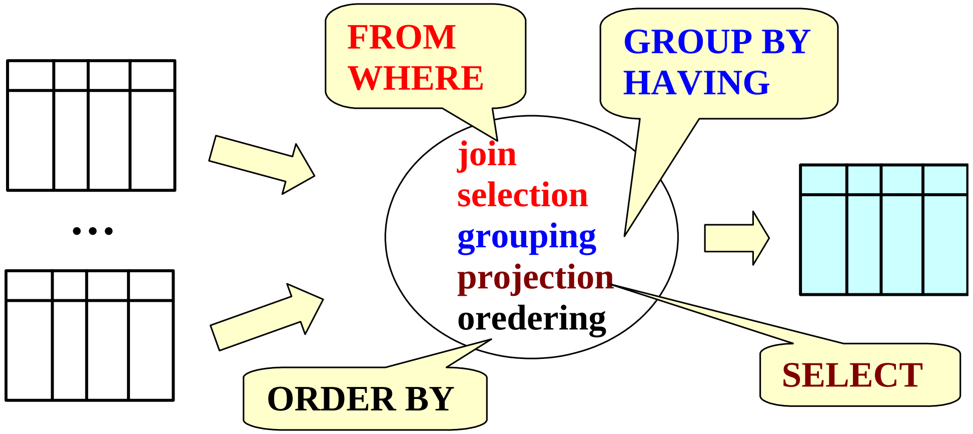
Okomentoval(a): [JZ14]: S případnými parametry jako u sekvence.
Ve skutečnosti SŘBD vytvoří odpovídající sekvenci.
GENERATED
[ALWAYS | BY DEFAULT [ON NULL]]
AS IDENTITY [(identity_options)]

5.2.3. Data manipulation

- statements: SELECT, UPDATE, DELETE, INSERT
- operands are base tables or views, the result is a table

- SELECT statement

```
  SELECT [ALL|DISTINCT] item [[AS] col_alias], …
     FROM table_expression [[AS] [table_alias]], …
     [WHERE condition]
     [GROUP BY col_name_from_FROM|number, …]
     [HAVING condition]
-------------------------------------------------------
     [ORDER BY col_name_from_SELECT|number [ASC|DESC]],
     …
```



- **Simple queries (over one table)**

  "Who are the clients of the bank?"

  ```
  SELECT clientNo, name
  FROM Client
  ```

  - **We can use the symbol ‘*’in SELECT clause with the meaning ‘all columns’**

  ```
  SELECT *
  FROM Client
  ```

  - **We use the keyword DISTINCT to eliminate duplicate rows**

  "Which towns are the clients of the bank from?"

  ```
  SELECT DISTINCT town
  FROM Client
  ```

  - **WHERE clause specifies conditions for row selection**

  "Which accounts are managed in branch Janska?"

  ```
  SELECT accNo
  FROM Account
  WHERE branch='Janska'
  ```

- Renaming
  - It is possible to introduce new names (aliases) for columns of the result (in SELECT clause)
  - It is possible to introduce new names of tables or table expressions (in FROM clause)

```
expression [AS] alias
```

  - The new names from SELECT clause can by employed only in ORDER BY clause, the new names of tables in all clauses of the SELECT statement
- Items in the SELECT clause are scalar expressions (a column name is a simple form of such expression)

"How much are assets of branches in USD at the rate of 25 CZK/$?"

```
SELECT name, assets/25 assets_v_USD
FROM Branch
```

- The rows of the result can be ordered - ORDER BY

```
SELECT name, assets/25 assets_v_USD
FROM Branch
ORDER BY assets_v_USD
```

or 2

- It is possible to query over several tables (JOIN operation)

"Which clients have their account in branch Janska?"

```
SELECT DISTINCT C.*
FROM Client C, Account A
WHERE C.clientNo=A.clientNo AND A.branch='Janska'
```

"Which clients did their transaction on 12.10.1998?"

```
SELECT C.clientNo, C.name, T.accNo, T.amount
FROM Client C, Account A, Transaction T
WHERE C.clientNo=A.clientNo AND A.accNo=T.accNo
   AND A.branch='Janska' AND T."date"='1998-10-12'
```

- Join types:
  - ○ inner
    - ◊ based on a general condition (Θjoin)
    - ◊ based on equality (equijoin)
    - ◊ natural (natural join)
  - ○ outer

Ex) T1

A	B	X
0	a	x
1	a	x
3	c	z

T2

X	C	D
x	1	0
x	2	0
y	3	1

Result?

T1 JOIN T2 ON A<C

T1 JOIN T2 ON A=D

T1 NATURAL JOIN T2

T1 NATURAL LEFT JOIN T2

- It is also possible to join instances of the same table

"Are there any clients living at the same address?"

```
SELECT C1.name, C1.clientNo, C2.name, C2.clientNo,
   C1.street, C1.town
FROM Client C1, Client C2
WHERE C1.town=C2.town AND C1.street=C2.street AND
   C1.clientNo>C2.clientNo
```

- The FROM clause can contain table expressions, e.g. join expression of a form:

```
table CROSS JOIN table   |
table [NATURAL] [join_type] JOIN table
            ON condition | USING (column, …)]
```

  - join types: INNER|(LEFT|RIGHT|FULL)[OUTER]|UNION

"Which clients have their account in branch Janska?"

```
SELECT DISTINCT clientNo, name, street, town
FROM Client NATURAL JOIN Account
WHERE branch='Janska'
```

Okomentoval(a): [C15]: USING je pro equijoin stejně pojmenovaných sloupců (obecně ale ne všech). Výsledný sloupec je jen jednou. Od NATURAL JOIN se tedy liší tím, že podmínka spojení zahrnuje obecně jen některé stejně pojmenované sloupce.

Okomentoval(a): [t16]: Takto je to v Oracle:
join_type
The join_type indicates the kind of join being performed:
Specify INNER to indicate explicitly that an inner join is being performed. This is the default.
Specify RIGHT to indicate a right outer join.
Specify LEFT to indicate a left outer join.
Specify FULL to indicate a full or two-sided outer join. In addition to the inner join, rows from both tables that have not been returned in the result of the inner join will be preserved and extended with nulls.
You can specify the optional OUTER keyword following RIGHT, LEFT, or FULL to explicitly clarify that an outer join is being performed.
JOIN
The JOIN keyword explicitly states that a join is being performed. You can use this syntax to replace the comma-delimited table expressions used in WHERE clause joins with FROM clause join syntax.

ON condition
Use the ON clause to specify a join condition. Doing so lets you specify join conditions separate from any search or filter conditions in the WHERE clause.

USING column
When you are specifying an equijoin of columns that have the same name in both tables, the USING column clause indicates the columns to be used. You can use this clause only if the join columns in both tables have the same name. Do not qualify the column name with a table name or table alias.

In an outer join with the USING clause, the query returns a single column which is a coalesce of the two matching columns in the join. The coalesce functions as follows:

COALESCE (a, b) = a if a NOT NULL, else b.

Therefore:

A left outer join returns all the common column values from the left table in the FROM clause.
A right outer join returns all the common column values from the right table in the FROM clause.
A full outer join returns all the common column values from both joined tables.

- Aggregate functions

```
COUNT (*)|
AVG|MAX|MIN|SUM|COUNT ([ALL|DISTINCT] col_name)
```

"How many clients does the bank have?"

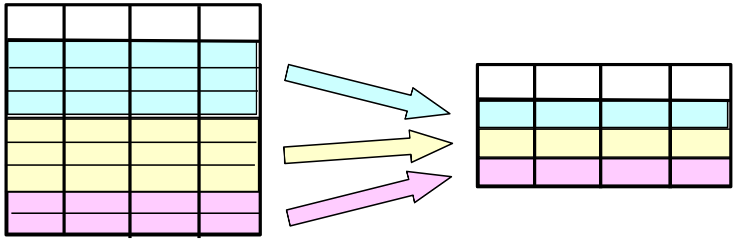
```
SELECT COUNT(*) number_of_clients
FROM Client
```

- it is not possible to nest aggregate functions (Oracle allows it)

- GROUP BY clause

"How much money is on account in each branch?"

```
SELECT branch, SUM(balance) total_on_accounts
FROM Account
GROUP BY branch
```



- restrictions for expressions in the SELECT clause (only aggregate functions, elements from GROUP BY, and constants)

"How many accounts and how much on them does every client have?"

```
SELECT C.name, C.clientNo, COUNT(*) accounts,
   SUM(balance) total
FROM Client C, Account A
WHERE C.clientNo=A.clientNo
GROUP BY C.clientNo, C.name
```

- **HAVING clause**
  - It is similar to WHERE but applied on aggregated groups

"Which clients have on their accounts more than 100000 CZK?"

```
SELECT C.name, C.clientNo, SUM(balance) total
FROM Client C, Account A
WHERE C.clientNo=A.clientNo
GROUP BY C.clientNo, C.name
HAVING SUM(balance)>100000
```

- **WHERE clause / conditional expressions**
  - **It can contain predicates (possibly with a NOT operator and conjuncted with AND, OR logic conjunctions):**
    - **Comparison**

```
row_contructor comparison_op row_contructor |
row_contructor comparison_op {ANY|SOME|ALL}
   (table expression)
```

"Which clients living out of Brno have more money in the bank than clients living in Brno?"

```
SELECT C.name, C.clientNo, SUM(balance) total
FROM Client C, Account A
WHERE C.clientNo=A.clientNo AND C.town<>'Brno'
GROUP BY C.name,C.clientNo
HAVING SUM(balance)>ALL
   (SELECT SUM(balance)
    FROM Client C, Account A
    WHERE C.clientNo=A.clientNo AND C.town='Brno'
    GROUP BY C.clientNo)
```

- How to test missing information

```
col_name IS [NOT] NULL
```

"Is the address of any client incomplete?"

```
SELECT *
FROM Client WHERE street IS NULL OR town IS NULL
```

- Predicate BETWEEN

```
expression [NOT] BETWEEN expression AND expression
```

- e BETWEEN c1 AND c2 is equivalent to: *e ≥ c1 AND e ≤ c2*

"Which transactions were done in October 1998?"

```
SELECT C.name, C.clientNo, A.accNo,
 T."date",T.amount
FROM Client C, Account A, Transaction T
WHERE C.clientNo=A.clientNo AND A.accNo=T.accNo
 AND T.date BETWEEN '1998-10-01' AND '1998-10-31'
```

- **Predicate EXISTS**

```
   [NOT] EXISTS (table_expression)
```

  - **It tests whether the table is not empty**

"Which clients have their account only in branch Janska?"

```
SELECT DISTINCT C.*
FROM Client C, Account A
WHERE C.clientNo=A.clientNo AND A.branch='Janska' AND
   NOT EXISTS(SELECT *
              FROM Account A
              WHERE C.clientNo=A.clientNo AND
                    U.branch<>'Janska')
```

  - **We usually use a subquery with '*' in the SELECT clause**

- **Predicate LIKE**

```
   string_expr [NOT] LIKE pattern [ESCAPE esc_char]
```

- **Pattern is a string expression, it can contain wild characters:**
  - ○ **_ - an arbitrary character,**
  - ○ **% - any number (possibly zero) of arbitrary characters**
- ***esc_character* - it cancels the meaning of a wild character**

Ex) str LIKE ‘_%’ ESCAPE '\‘

"Which clients have their first name Jan?"

```
SELECT *
FROM Client
WHERE name LIKE 'Jan %'
```

- **Predicate IN**

```
   row_constructor [NOT] IN (table_expr)|
   salar_expr [NOT] IN (list_of_scalar_expressions)
```

"Are there any clients from Brno or Prague?"

```
SELECT *
FROM Client
WHERE town IN ('Prague', 'Brno')
```

"Which clients did there transactions in October 1998?"

```
SELECT *
FROM Client
WHERE clientNo IN
    (SELECT clientNo FROM Account
     WHERE accNo IN
        (SELECT accNo  FROM Transaction
         WHERE date BETWEEN '1998-10-01'
            AND '1998-10-31'))
```

- **SQL WITH clause (from SQL-99)**
  - **It specifies tables that can be used in the following select statement.**

```
WITH table_name AS (table_expression), …
   SELECT_statement
```

- **it allows us to simplify complex queries**

```
WITH averages AS
   (SELECT custNo,AVG(balance) AS average
    FROM Account
    GROUP BY custNo)
SELECT *
FROM averages
WHERE average >= ALL
      (SELECT average
       FROM averages);
```

- Operators for set operations

```
table_expression UNION|EXCEPT|INTERSECT [ALL]
         table_expression [ORDER_BY_clause]
```

Ex) Assume another table Loan

loanNo	clientNo	branch	amount	paid_off

"Which clients have either account or loan managed in branch Janska?"

```
SELECT C.name, C.clientNo
FROM Client C, Account A
WHERE C.clientNo=A.clientNo AND A.branch='Janska'
UNION
SELECT C.name, C.clientNo
FROM Client C, Loan L
WHERE C.clientNo=L.clientNo AND L.branch='Janska'
```

- Duplicates are eliminated, to see them use ALL.

- **INSERT statement**

```
   INSERT INTO table_name [(col_name, …)] source
```

→ It inserts zero, one or more rows into a table

➢ Possible sources are:

- The row of default values (from CREATE TABLE)

```
DEFAULT_VALUES
```

- The row of values from a list

```
VALUES(scalar_expr|NULL|DEFAULT, …)
```

```
INSERT INTO Client
VALUES('440726/0672','Jan Novak','Cejl 8','Brno')
```

- The result of a subquery

```
table_expression
```

"Insert rows of clients who have their accounts in Janska into a CJ table."

```
INSERT INTO CJ
   SELECT DISTINCT C.*
   FROM Client C, Account A
   WHERE C.clientNo=A.clientNo AND A.branch='Janska'
```

- **DELETE statement (searched)**

```
   DELETE FROM table_name
      [WHERE condition]
```

→ it deletes the rows that satisfy the condition

"Delete information about clients without an account."

```
DELETE FROM Client
WHERE clientNo NOT IN (SELECT clientNo FROM Account)
```

  - The difference of DELETE FROM T and DROP TABLE T.

- **UPDATE statement (searched)**

```
   UPDATE table_name
      SET col_name = scalar_expr|NULL|DEFAULT, …
      [WHERE condition]
```

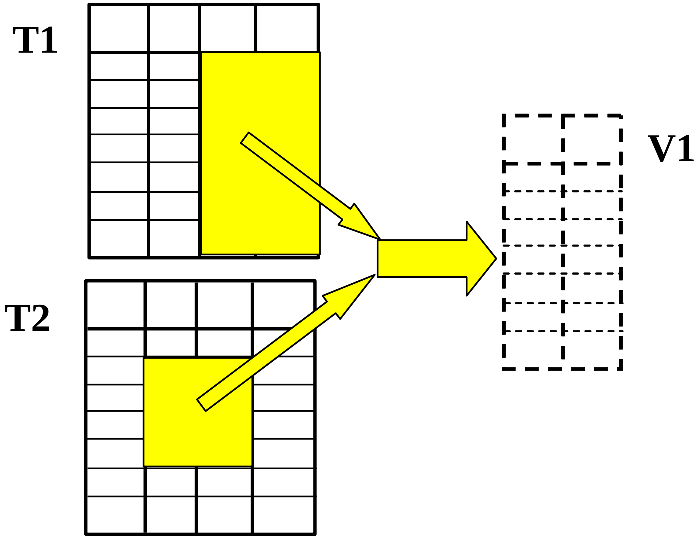
→ the statement changes values in specified columns of rows of the table satisfying the condition

"Update balance for deposit 1000 CZK on account number 100."

```
UPDATE Account
SET balance=balance+1000
WHERE accNo=100
```

5.2.4. Views

Named derived tables which do not exist in the database.



- **Creating a view**

```
  CREATE VIEW view_name [(col_name, …)]
     AS table_expression
     [WITH CHECK OPTION]
```

→ the statement inserts the view definition into a system catalogue

- **The columns must have unique names (possibly renamed).**

Ex) A view for clients who have an account in branch Janska.

```
CREATE VIEW Janska AS
   SELECT C.*
   FROM Client C, Account A
   WHERE C.clientNo=A.clientNo AND
         A.branch ='Janska'
WITH CHECK OPTION
```

- Dropping a view

```
DROP VIEW view_nameu [RESTRICT|CASCADE]
```

→ the statement deletes the information about the view from the system catalogue

- Manipulations over views

When the view is used, the DBMS employs the view definition and performs operation on base tables.

Ex)

```
SELECT * FROM Janska WHERE town = 'Brno'
```

⇒

```
SELECT C.*
FROM Client C,Account A
WHERE C.town = 'Brno' AND
      C.clientNo=A.clientNo AND A.branch='Janska';
```

- Updateability of views

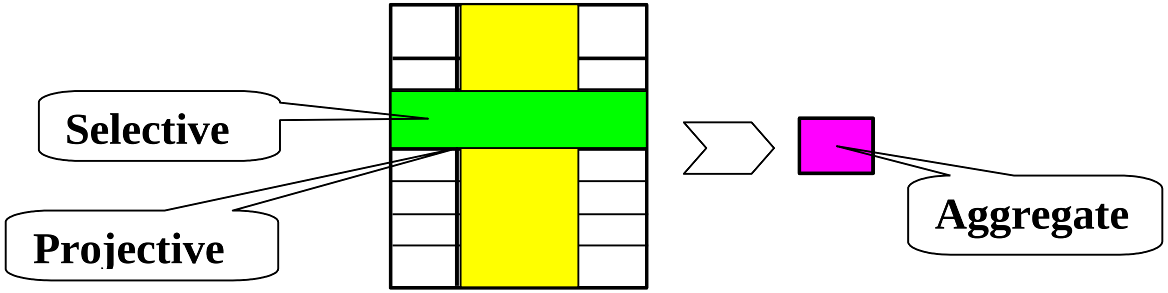
DBMS must be able to transform uniquely operations for row insertion, deletion, and updating to corresponding operations on base tables.

Ex)

```
CREATE VIEW Number_of_accounts (name,count)
      AS SELECT branch, COUNT(*)
            FROM Account
            GROUP BY branch;
```

Views with DISTINCT, GROUP BY, HAVING, aggregate functions and joining several tables allow reading only.

Ex) Updateability of a view on one base table



- A selective view

```
CREATE VIEW Brnensti AS
  SELECT* FROM Client WHERE town='Brno'
```

- A projective view *without a primary key column included*

```
CREATE VIEW Brnensti1 AS
  SELECT name, street FROM Client WHERE town='Brno'
INSERT INTO Brnensti1
   VALUES ('Josef Vlk','Koliste 55')
```

- **A projective view *with a primary key column, not included columns allow NULL***

```
CREATE VIEW Brnensti2 AS
 SELECT clientNo,name FROM Client WHERE town='Brno'
INSERT INTO Brnensti2
   VALUES ('112233/4444','Josef Vlk')
```

- **An aggregate view**

```
CREATE VIEW Number_of_accounts (name, count) AS
SELECT branch, COUNT(*) FROM Account GROUP BY branch;
INSERT INTO Number_of_accounts VALUES ('Panska',20)
```

- **The WITH CHECK OPTION clause**
  - **It is checked that manipulations with the view does not violate the view definition.**

```
CREATE VIEW Brnensti AS
   SELECT * FROM Client WHERE town=’Brno’
WITH CHECK OPTION

UPDATE Brnensti
SET town=’Praha’
WHERE clientNo=…
```

- **Materialized views**

  Views whose data is stored (as a copy of original data in the database). It is possible to refresh the content of the view.

```
CREATE MATERIALIZED VIEW MFromBrno
REFRESH ON COMMIT AS
   SELECT* FROM Customer WHERE town='Brno'
```

  - **The motivation**
    - **Increased performance, restricted access to original data.**

- Main application areas:
  - Data warehouses – summarizing views.
  - Distributed databases – data replication in nodes.
  - Mobile databases – materialization of views used by mobile clients.

➢ Using views (in general, not only materialized)

Advantages:

- Access restriction, logical structure of the database is hidden (security)
- Complexity of a query can be hidden (simplification)
- The way of accessing data can be hidden (it is independent on possible future change)

5.2.5. Access to the system catalogue

The system catalogue of a relational database has the same interface as a user part of the database with some restrictions, i.e. it contains tables and views.

Ex) Oracle: views: ALL_, DBA_, USER_

```
USER_TABLES(TABLE_NAME,TABLESPACE_NAME,...),
USER_TAB_COLUMNS
       (TABLE_NAME,COLUMN_NAME,DATA_TYPE,...)
```

"What columns do the table Client have?"

```
SELECT column_name
FROM User_tab_columns
WHERE table_name = 'CLIENT'
```

- The catalogue data can only be read, other manipulations are done indirectly by means of the CREATE, ALTER, DROP statements.
- Sometimes there are also special statements to access metadata,

Ex) MySQL: SHOW TABLES; DESCRIBE Client;

5.2.6. Working with missing information

- the need in the real-world applications
- solution: - one selected value of the domain of the attribute
  - special "value“ (NULL in SQL)
- it influences operations (e.g. A+B, A>B)
  ➔ ***three-valued logic*** (3VL) - {true, false, unknown}

- Rules
  - ➢ scalar expressions – the result is NULL if any operand is NULL
  - ➢ comparison – the result is *unknown* if any operand is NULL
  - ➢ aggregate functions – NULL behaves as a neutral value. If values are missing in all rows the result of COUNT is zero, NULL for other aggregate functions
  - ➢ WHERE clause – only rows for which the condition is *true* are selected
  - ➢ comparison of rows

a	NULL	c
a	NULL	c

neither *r1* = *r2*, nor *r1* <> *r2*,
for DISTINCT it is *true*

- **Testing of missing information**

```
col_name IS [NOT] NULL
```

Okomentoval(a): [t17]: POZOR ne jm_sloupce = NULL.

- **Outer join (OUTER JOIN - right, left, full), outer union**

Př) T1

A	B	X
0	a	x
1	a	x
3	c	z

T2

X	C	D
x	1	0
x	2	0
y	3	1

Result?

T1 NATURAL LEFT JOIN T2

T1 NATURAL RIGHT JOIN T2

T1 NATURAL FULL JOIN T2

T1 UNION JOIN T2

"How many accounts and much money on them does every client of the bank have?"

```
SELECT name, clientNo, COUNT(accNo)
       count_of_accounts, SUM(balance) total
FROM Client NATURAL LEFT JOIN Account
GROUP BY name,clientNo
ORDER BY total DESC
```

5.2.7. Other SQL statements

- **Access control (see chap. 7)**

  GRANT, REVOKE

- **Session control (see chap. 9)**

  CONNECT, DISCONNECT, SET CONNECTION, …

- **Transaction processing (see chap. 9)**

  COMMIT, ROLLBACK, SET TRANSACTION (isolation level,…),…

- **Others**

5.3. SQL in application programs

- **Three binding styles SQL as defined in the standard:**
  - ➢ **direct SQL**
  - ➢ **embedded SQL**
  - ➢ **module language – it makes it possible to create so called SQL modules for a particular programming language. The module can contain SQL procedures and functions; each of them contains one SQL statement. An SQL module can be compiled separately of other program modules. Its routines can be called from a given programming language by the same way as other procedures and functions.**

- **SQL/92 is not computationally complete – it does not provide control structures**
- **Embedded SQL is more powerful than the direct SQL, all statements of the direct SQL can be employed in the embedded SQL.**

5.3.1. Embedded SQL

- **Principles:**
  - **Embedded SQL statements have a form**

```
 EXEC SQL SQL_statement
```

and are ended by the rule of a host language (e.g. ';' for C).

  - **Any statement of the direct SQL can be used in host environment.**
  - **It is possible to refer host variables (called "bind") - start with ':'.**
  - **The referred bind variables must be declared in a declaration section:**

```
 EXEC SQL BEGIN DECLARE SECTION
......
END DECLARE SECTION
```

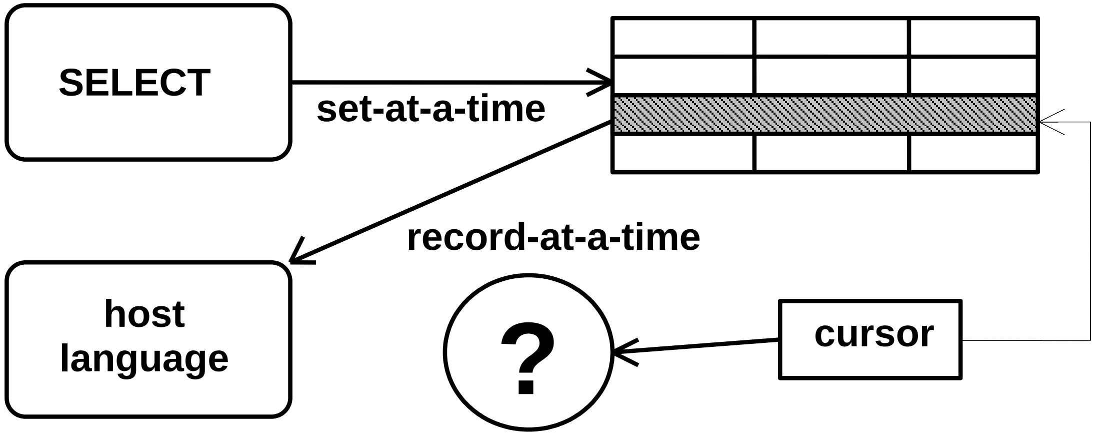
  - **Any program containing embedded SQL must declare a host variable SQLCODE or SQLSTATE, the value of which set the DBMS after executing every SQL statement.**
  - **Host variables must be of an appropriate type with respect to their use.**
  - **Host variables and column names can have the same name.**

- The value of SQLCODE or SQLSTATE should be tested after each SQL statement; there is a WHENEVER statement that simplifies the testing.:

```
EXEC SQL WHENEVER condition action
```

  - the condition is SQLERROR or NOT FOUND, the action is CONTINUE or GO TO label.

- The concept of cursor



- **Statements that do not require a cursor**
  - **A single-row SELECT**

```
  SELECT … INTO host_var[INDICATOR indik], … FROM …
```

  - **INSERT, searched UPDATE and DELETE**
- **Statements related to cursor:**
  - **Cursor declaration**

```
    DECLARE [INTENSIVE|SCROLL] cursor_name CURSOR
          FOR select_statement_possibly_with_ORDER_BY
          [FOR READONLY| UPDATE[OF column]]
```

    - **rules for a cursor to be able to allow updating are similar to those for views udateability**

Ex)

```
DECLARE Janska CURSOR FOR
  SELECT C.clientNo, C.name, C.street, C.town
  FROM Client C, Account A
  WHERE C.clientNo=A.clientNo AND branch='Janska'
```

- Query execution

```
OPEN cursor_name
```

Ex)

```
OPEN Janska
```

- Accessing rows

```
FETCH [[NEXT|PRIOR|FIRST|...] FROM] cursor_name
           INTO list_of_variables
```

Ex)

```
FETCH Janska INTO :clientNo, :name, :street, :town
```

- Closing (deactivation) of a cursor

```
CLOSE cursor_name
```

Ex)

```
CLOSE Janska
```

- Positional UPDATE and DELETE

```
... WHERE CURRENT OF cursor_name
```

Ex) Using a cursor in PL/SQL (Oracle)

```
DECLARE CURSOR z IS
   SELECT clientNo, name FROM Client WHERE town =
      'Brno';
…
BEGIN
   OPEN z;
   LOOP
     FETCH z INTO cN, n;
     EXIT WHEN z%NOTFOUND;
     …
   END LOOP;
   CLOSE z;
END;
```

5.3.2. Dynamic SQL

It makes it possible to assemble an SQL in runtime.

- **Preparing a statement for execution**

```
PREPARE statement_name FROM string|variable
```

  - ***prepareable statement* – a single-row SELECT without INTO, INSERT, searched UPDATE and DELETE, a specification part of a cursor declaration (a select statement)**
  - **it is possible to use a wild character '?' for parameters**

- **Statement execution**

```
EXECUTE statement_name [INTO …][ USING input_values]
```

- **Space release**

```
DEALOCATE PREPARE statement_name
```

- **Preparing and immediate execution**

```
EXECUTE IMMEDIATE string|variable
```

- **Prepared\ statement in cursor definition**

```
DECLARE cursor_name CURSOR FOR statement_name
```

Ex) Oracle Pro*C

```
sprintf(s1,”%s”,”UPDATE Client SET name=? WHERE
   clientNo=?”);
EXEC SQL PREPARE statement FROM :s1;
EXEC SQL EXECUTE statement USING :new_name,:clNo;
```

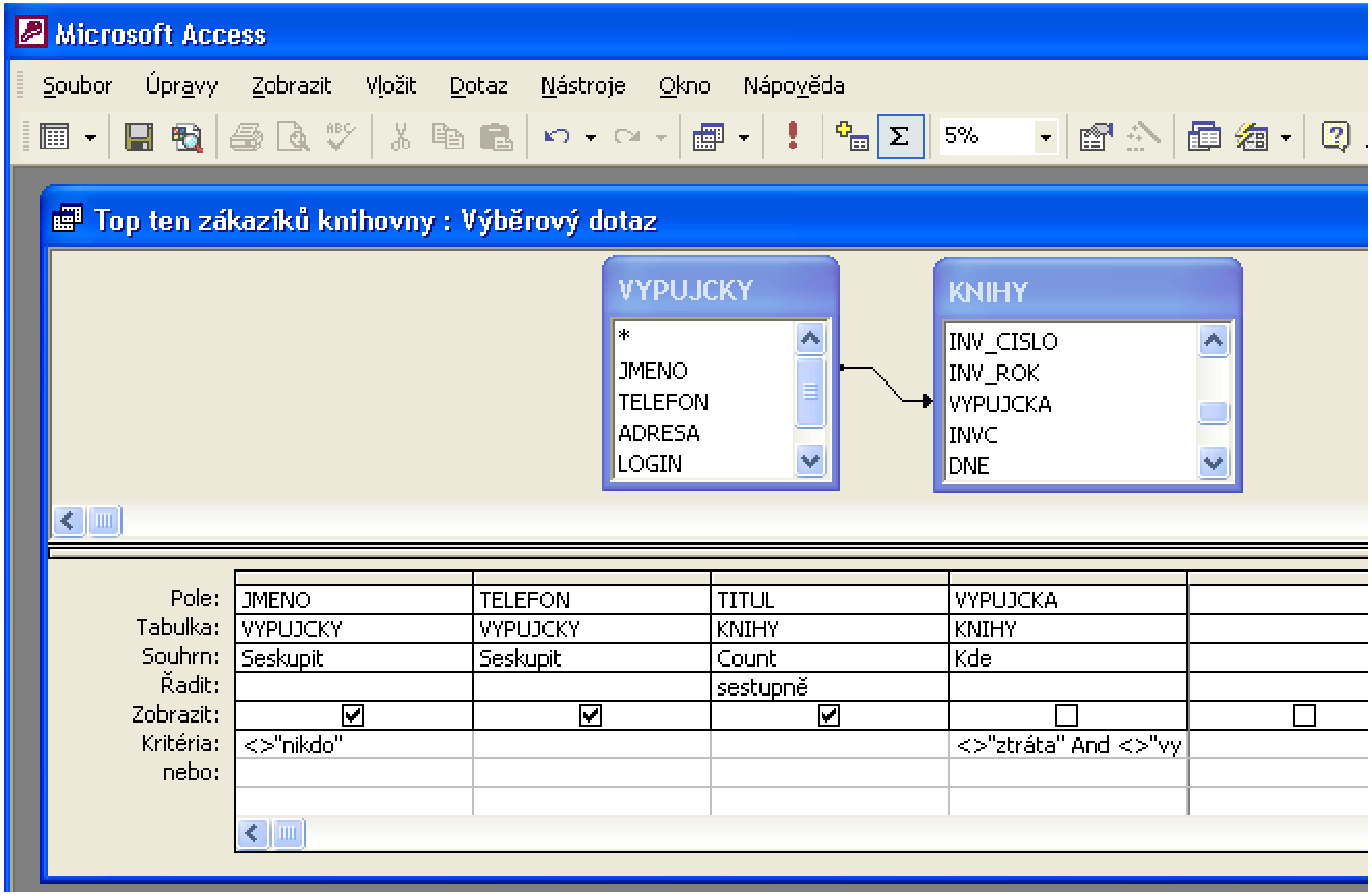
- **Flexibility versus effectiveness**
  - **possibility to assemble statements in runtime**
  - **compilation only in runtime ⇒ late checks, late binding**

5.4. Other relational languages

5.4.1. Query By Example

- developed by IBM in 70s,
- it is an interactive way of querying rather than writing statements
- it is usually available as a development tool or a tool for and users (not necessary to know SQL)
- originally the user interface of a tool was table-oriented, now it is a form-oriented using GUI (sometimes called GQBE (Graphical Query By Example)

Ex) Microsoft Access



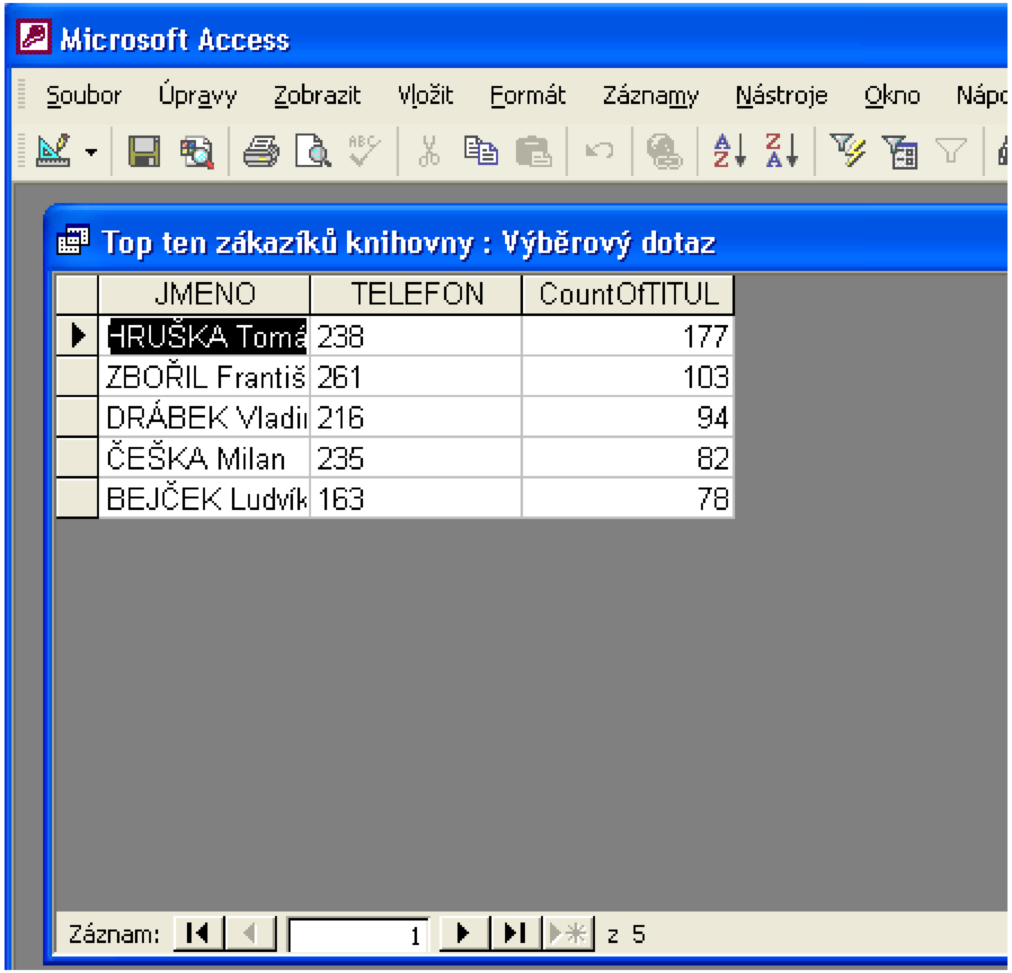
Pole:	JMENO	TELEFON	TITUL	VYPUJCKA	
Tabulka:	VYPUJCKY	VYPUJCKY	KNIHY	KNIHY	
Souhrn:	Seskupit	Seskupit	Count	Kde	
Řadit:			sestupně		
Zobrazit:	☑	☑	☑	□	□
Kritéria:	<>"nikdo"			<>"ztráta" And <>"vy	
nebo:					

The generated query:

```
SELECT TOP 5 PERCENT VYPUJCKY.NAME, VYPUJCKY.TELEFON,
Count(KNIHY.TITUL) AS CountOfTITUL
FROM VYPUJCKY LEFT JOIN KNIHY ON VYPUJCKY.NAME =
KNIHY.VYPUJCKA
WHERE (((KNIHY.VYPUJCKA)<> "ztráta" AND
(KNIHY.VYPUJCKA)<> "vyřazeno"))
GROUP BY VYPUJCKY.NAME, VYPUJCKY.TELEFON
HAVING (((VYPUJCKY.NAME)<> "nikdo"))
ORDER BY Count(KNIHY.TITUL) DESC;
```

Okomentoval(a): [C18]: Priklad je nad realnymi tabulkami (daty) - Kirchnerove knihovna (cela knihovna je cca 8 tabulek).
Dotaz zobrazi "top ten" vypujcovatelu (osklivy patvar). Na prvnim obrazku je cely desktop, druhy je vyrez, ktery obsahuje vsechno podstatne, treti vysledek.
Nad okenkem s grafikou dotazu je tech zadanych 5%, vedle je znak sigma jako zadost o sumacni dotaz (tak se to rika?). Cara mezi tabulkami udava join, v danem pripade je pouzit left join (zcela zbytecne - mel by vyznam, pokud bych chtel celou tabulku a zaroven videt i zakazniky, kteri nemaji nic vypujceno).
3 a 4 obrazek ukazuji vysledek dotazu (celu desktop a podstatny vyrez).

The result of the query:



Microsoft Access
Soubor Úpravy Zobrazit Vložit Formát Záznamy Nástroje Okno Nápc

Top ten zákazíků knihovny : Výběrový dotaz

JMENO	TELEFON	CountOfTITUL
HRUŠKA Tomá	238	177
ZBOŘIL Františ	261	103
DRÁBEK Vladi	216	94
ČEŠKA Milan	235	82
BEJČEK Ludvík	163	78

Záznam: 1 z 5

References

1. Date, C., J., Darwen, H.: A Guide to the SQL Standard. Fourth Edition. Addison-Wesley, 1997.
2. Silberschatz, A., Korth H.F, Sudarshan, S.:Database System Concepts. Fourth Edition. McGRAW-HILL. 2001, str. 135 – 225.