Data Manipulation (DML) and Data Definition (DDL)
Inserting Tuples

- `INSERT INTO REGION VALUES (6,'Antarctica', '')`
- `INSERT INTO NATION (N_NATIONKEY, N_NAME, N_REGIONKEY)
  SELECT NATIONKEY, NAME, 6 FROM ANTARCTIC_NATIONS`

Updating Tuples

- `UPDATE LINEITEM
  SET L_DISCOUNT = L_DISCOUNT + 0.01
  WHERE L_SUPPKEY = 12`

Deleting Tuples

- `DELETE FROM REGION WHERE R_REGIONKEY > 5`
Merging Data (\texttt{MERGE})

- Combination of \texttt{INSERT}, \texttt{DELETE} and \texttt{UPDATE} (SQL:2003)
- Merges two relations in a predetermined manner
- \texttt{MERGE INTO} \texttt{<table1>}
  \texttt{USING} \texttt{<table2>|<fullselect>} \texttt{ON} \texttt{<condition>}
  \texttt{WHEN} \texttt{[NOT]} \texttt{MATCHED} \texttt{THEN} \texttt{<operation>}

- Parameters
  - \texttt{<table1>}: table which should be updated
  - \texttt{<table2>}: new data
  - \texttt{<condition>}: tuple comparison (when are two tuples equal?)
  - \texttt{MATCHED}: Tuples exist in both relations
  - \texttt{NOT MATCHED}: Tuple appear only in the new relation
  - \texttt{<operation>}: Action (UPDATE, INSERT, DELETE)

- (Not fully supported in H2)
Example (MERGE)

- Two relations: PERSONS (left) and NEW_PERSONS (right)

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul</td>
<td>2000</td>
</tr>
<tr>
<td>Fritz</td>
<td>1000</td>
</tr>
<tr>
<td>Fritz</td>
<td>1500</td>
</tr>
<tr>
<td>Dirk</td>
<td>3000</td>
</tr>
</tbody>
</table>

- Merging both relations

```
MERGE INTO PERSONS P
USING NEW_PERSONS N ON P.NAME = N.NAME
WHEN MATCHED THEN UPDATE SET P.SALARY = N.SALARY
WHEN NOT MATCHED THEN INSERT (NAME,SALARY)
VALUES (N.NAME, N.SALARY)
```

- Result (PERSONS)

<table>
<thead>
<tr>
<th>NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paul</td>
<td>2000</td>
</tr>
<tr>
<td>Fritz</td>
<td>1500</td>
</tr>
<tr>
<td>Dirk</td>
<td>3000</td>
</tr>
</tbody>
</table>
DB2 Database Objects

Relation (\textit{TABLE})
- Unordered set of tuples
- Permanent or temporary

View (\textit{VIEW})
- Virtual relation
- e.g. simplification of queries, user-defined view on data

Index (\textit{INDEX})
- Primary or secondary
- Data structure to store / find data
- Additional: preserving uniqueness and sorting

What else
- \texttt{DATABASE} (not in H2), \texttt{TABLESPACE} (not in H2), \texttt{TYPE} (DOMAIN in H2), \texttt{SCHEMA, BUFFERPOOL} (not in H2), \texttt{TRIGGER, USER, CONSTANT, ALIAS}, ...
Definition of Database Objects (DDL)

- e.g. table
- Creating using `CREATE` command

```sql
CREATE TABLE REGION (
    R_REGIONKEY INTEGER NOT NULL PRIMARY KEY,
    R_NAME CHAR(25) NOT NULL,
    R_COMMENT VARCHAR(152)
)
```

- Deleting using `DROP`

```sql
DROP TABLE REGION
```

- Changing using `ALTER`

```sql
ALTER TABLE REGION ADD COLUMN AREA INT
```
Integrity Constraints (constraints)

- Primary Key, Foreign Key, Distinct values (UNIQUE), value-based (CHECK)
- `ALTER TABLE <table> ADD CONSTRAINT <constraint-name>`
  - PRIMARY KEY (<attribute-list>)
  - FOREIGN KEY (<attr-list>) REFERENCES <table>(<attr-list>)
  - UNIQUE (<attribute-list>)
  - CHECK (<predicate>)

- Example
  
  `ALTER TABLE REGION ADD CONSTRAINT MAX5 CHECK (R_REGIONKEY BETWEEN 1 AND 5)`
**View (VIEW)**

- Virtual relation to simplify queries and to define user-specific representations
- Also used for security reasons to hide tuples or attributes
- Specified by SQL queries
- Example
  ```sql
  CREATE VIEW NAT_REG AS
  SELECT N_NAME, R_NAME
  FROM NATION, REGION
  WHERE N_REGIONKEY = R_REGIONKEY
  
  SELECT * FROM NAT_REG
  
<table>
<thead>
<tr>
<th>N_NAME</th>
<th>R_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALGERIA</td>
<td>AFRICA</td>
</tr>
<tr>
<td>MOZAMBIQUE</td>
<td>AFRICA</td>
</tr>
<tr>
<td>MOROCCO</td>
<td>AFRICA</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
  ```
Modifying Views

- INSERT, DELETE and UPDATE possible, if
  - Alternative 1: INSTEAD OF trigger is defined
  - Alternative 2: specific constraints are met


- WITH CHECK OPTION
  - New or updated tuples must fulfill the view constraints

Example

- CREATE VIEW R AS
  SELECT R_REGIONKEY, R_NAME FROM REGION
  WHERE R_REGIONKEY<10
  WITH CHECK OPTION

- INSERT INTO R VALUES (8, 'A') → OK
- INSERT INTO R VALUES (11, 'B') → Error
SQL-statement defining a table `FootballMatch` containing the results of football matches in the national league:

```
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:

```
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 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> | <domain> } [ <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
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 Access-SQL 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.

```
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: Table Constraints

The second part of a table definition is optional. It consists of one or more table constraints, normally expressing a restriction on several columns:

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

Syntax of table constraints:

```sql
[ 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:
    \[
    \text{CHECK} \left( \text{Type} > 0 \text{ AND Type} < 35 \right)
    \]

- 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: `<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`
**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: `PRIMARKEYY ( <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):

```
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,
   ...)
```

b₁ references a₁
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.

- DB systems usually treat references and referential integrity quite similarly:
  - with change propagation:  ON UPDATE CASCADE
  - with delete propagation:  ON DELETE CASCADE
Global Constraints in SQL: Assertions

- Assertions are 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:

  ```sql
  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:

  ```sql
  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!
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.:

C₁: „Each course is given by a professor!"
C₂: „Each professor has to give at least one course!"

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

```
INSERT INTO professor
INSERT INTO course
```

Each intermediate state would be inconsistent: No sequence possible!
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:

\[
\text{SET CONSTRAINTS} \{ \langle \text{list-of-constraints} \rangle \mid \text{ALL} \} \\
\{ \text{DEFERRED} \mid \text{IMMEDIATE} \}
\]

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