Relational Database Systems I

Christoph Lofi
Simon Barthel
Institut für Informationssysteme
Technische Universität Braunschweig
www.ifis.cs.tu-bs.de
• SQL data definition language
• SQL data manipulation language (apart from SELECT)
• SQL $\neq$ SQL
• Some advanced SQL concepts
9.1 Recap

Environment
**BigCompany database server**

Catalog
*human_resources*

Schema
*people*

Table
*staff*

Table
*has_office*

Schema
*taxes*

...*

Schema
*training*

...*

Catalog
*production*

Schema
*products*

...*

Schema
*testing*

...*
9.1 Recap

• Last week, you learned how to query an existing relational database

\[
\text{SELECT} <\text{attribute, function, scalar subquery}> \\
\text{FROM} <\text{table, table subquery}> \\
[\text{WHERE} <\text{condition}>] \\
[\text{GROUP BY} <\text{attribute list}>] \\
[\text{HAVING} <\text{condition}>] \\
[\text{UNION/INTERSECT/EXCEPT} <\text{query}>] \\
[\text{ORDER BY} <\text{attribute list}>]
\]
9.1 SQL DDL

- What’s missing?
  - how to create schemas, tables, ...
  - how to drop schemas, tables, ...
  - how to alter schemas, tables, ...
  - how to insert new tuples into existing tables?
  - how to delete tuples from existing tables?
  - how to update tuples in existing tables?
• **CREATE SCHEMA** creates a **new schema** with a given name for a given **owner**
  – if no schema name is provided, the **current username** is used
  – if no explicit owner is provided, also the **current user** is used

• **Example**
  – **CREATE SCHEMA** heroes **AUTHORIZATION** batman
CREATE TABLE creates a new table with a given name

- contains column definition for each column
- contains additional table-specific structural constraints
9.1 SQL DDL: Tables

- each column has a **name** and a **data type**
- each column may have multiple column **options**
- example
  
  • `CREATE TABLE person (`
  
  name VARCHAR(200),
  
  age INTEGER
  `
  ```

  column def

  column name → data type → column option
### 9.1 SQL DDL: Tables

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>(precision, scale)</td>
</tr>
<tr>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td>(length)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>(length)</td>
</tr>
<tr>
<td>DATE</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td></td>
</tr>
</tbody>
</table>

...
## 9.1 SQL DDL: Tables

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>INTEGER</td>
<td>Integer value depending on system</td>
</tr>
<tr>
<td>Float</td>
<td>FLOAT</td>
<td>Floating point number of approximate precision (the supported precision is implementation-dependent)</td>
</tr>
<tr>
<td>Numeric</td>
<td>NUMERIC((p,s))</td>
<td>A decimal number with (p) digits. ((p-s)) before the decimal point and (s) digits after the decimal point</td>
</tr>
<tr>
<td>Character</td>
<td>CHAR((x))</td>
<td>A textual string of length (x)</td>
</tr>
<tr>
<td>Character varying</td>
<td>VARCHAR((x))</td>
<td>A textual string of length at most (x)</td>
</tr>
<tr>
<td>Date</td>
<td>DATE</td>
<td>Year, month, and day</td>
</tr>
<tr>
<td>Time</td>
<td>TIME</td>
<td>Hours, minutes, and seconds</td>
</tr>
<tr>
<td>Timestamp</td>
<td>TIMESTAMP</td>
<td>A date and a time</td>
</tr>
</tbody>
</table>
### 9.1 SQL DDL: Tables

- **Column definition**
  - Column name
  - Data type
  - Column options

- **Column options**
  - NOT NULL
  - DEFAULT
  - NULL

- **Constraint**
  - Constraint name
  - PRIMARY KEY
  - UNIQUE

- **Check condition**
  - Check (check condition)

- **Reference clause**
  - Reference clause
9.1 SQL DDL: Tables

- **NOT NULL:**
  the NULL value is not allowed for the column

- **Example**
  ```sql
  CREATE TABLE person (  
    name VARCHAR(200) NOT NULL,  
    age INTEGER NOT NULL  
  )
  ```

- **DEFAULT:**
  defines the default value if a value is not explicitly set
  - usually a constant or NULL
  - if omitted, NULL is the default
• **Column constraints**
  
  – restricts possible values for the current column
  
  – may have a **unique name** indicated by **CONSTRAINT** `<name>`
    
    • if name is omitted, system creates a default name
  
  – **CHECK**: user-defined constraint.
    To be valid, values have to satisfy the condition.

  – example
    
    • **CREATE TABLE** `person` (  
      name `VARCHAR(200)`,  
      age `INTEGER CONSTRAINT adult  
           CHECK (age >= 18)`
    )
– **UNIQUE**: no duplicate values are allowed within this attribute
  
  • this option can only be used if the uniqueness constraints concerns only a single attribute
  
  • for multi-attribute uniqueness constraints, there is a different option (later)
  
  • implies **NOT NULL**
    
    – **note**: in DB2, **NOT NULL** has to be specified nevertheless ...
  
  – example
    
    • `CREATE TABLE person (`
      
      name  VARCHAR(200) NOT NULL UNIQUE,
      
      age   INTEGER NOT NOT NULL`
    )
9.1 SQL DDL: Tables

– **PRIMARY KEY**: each table may have a **primary key** (optionally, but recommended) made up of at least one column
  
  • this option can only be used if the primary key consists of only one column
  
  • for multi-column primary keys, you need a different option (later)
    
  • implies **NOT NULL** and **UNIQUE**
    
    – again, DB2 needs an explicit **NOT NULL**

– additionally, a **referential clause** may be specified (see next slides)
9.1 SQL DDL: Referential Integrity

• Rows in tables may refer to rows in other tables to capture relationships

• Of course, you should not be allowed to refer to a non-existing row

– referential integrity between primary keys and foreign keys ensures that references are correct
9.1 SQL DDL: Referential Integrity

Example

Conceptual ER schema

- Hero
  - id
  - real name
  - alias

- Power
  - id
  - name
  - description

- Power has hero
  - strength
  - (0,*)

- Hero has power
  - (0,*)
9.1 SQL DDL: Referential Integrity

Resulting tables

Tables refer to others by primary keys and foreign keys.
9.1 SQL DDL: Referential Integrity

• Referential integrity can be defined using the REFERENCES clause
  – either used by constraints in column options or within table constraints
9.1 SQL DDL: Referential Integrity

• Example

- `CREATE TABLE employee(`
  id `INTEGER NOT NULL PRIMARY KEY,`
  name `VARCHAR(100) NOT NULL`
)`

- `CREATE TABLE managed_by(`
  `employee` `INTEGER NOT NULL REFERENCES employees,`
  `manager` `INTEGER NOT NULL REFERENCES employees`
)`
• Optionally, you may specify what happens if a row that is referenced will be deleted or modified
  
  - **ON DELETE**: if a referenced row is deleted, ...
    
    - **NO ACTION**: ...reject the deletion (that is, it cannot be performed)
    - **SET NULL**: ...delete it and set all referencing foreign keys to **NULL**
    - **CASCADE**: ...delete it along with all rows referring to it

  - **ON UPDATE**: if the primary key of a referenced row is modified, ...
    
    - **NO ACTION**: ...reject the modification (that is, it cannot be performed)
    - **CASCADE**: ...change all values of referencing foreign keys

  - default
    
    - **ON DELETE NO ACTION ON UPDATE NO ACTION**
9.1 SQL DDL: Table Constraints

CREATE TABLE name ( column def )

table constraint

CONSTRAINT const. name

FOREIGN KEY ( column name )

CHECK ( check condition )

PRIMARY KEY ( column name )

reference clause
9.1 SQL DDL: Table Constraints

- Table constraints behave similar to constraints in column options
  - if no name is provided, a name is automatically generated
  - the CHECK condition may contain any Boolean predicate
  - in contrast to column options, table constraints may declare primary keys consisting of multiple attributes
  - foreign keys declare references to primary keys of other tables
    - see referential integrity
9.1 SQL DDL: Table Example

### has_alias
- **hero**
- **alias**

### hero
- **id**
- **real_name**

### has_power
- **hero**
- **power**
- **power_strength**

### power
- **id**
- **name**
- **description**

**Primary key**
- hero.id
- power.id

**Foreign key**
- hero.id in has_power
- alias.id in has_alias

**Relationships**
- hero has alias
- hero has power
- power has strength
CREATE TABLE hero(
  id INTEGER NOT NULL PRIMARY KEY,
  real_name VARCHAR(100)
)

CREATE TABLE power(
  id INTEGER NOT NULL PRIMARY KEY,
  name VARCHAR(100),
  description VARCHAR(255)
)

CREATE TABLE has_alias(
  hero INTEGER REFERENCES hero ON DELETE CASCADE ON UPDATE CASCADE,
  alias VARCHAR(100) NOT NULL,
  PRIMARY KEY (hero, alias)
)

link has_alias to hero
delete alias if hero is deleted
update alias if hero is updated
composed primary key
CREATE TABLE has_power(
  hero INTEGER NOT NULL,
  power INTEGER NOT NULL,
  power_strength INTEGER NOT NULL,
  PRIMARY KEY (hero, power),
  FOREIGN KEY (hero) REFERENCES hero
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  FOREIGN KEY (power) REFERENCES power
    ON DELETE CASCADE
    ON UPDATE CASCADE
)
9.1 SQL DDL: Drop Tables

- For deleting tables, there is the DROP TABLE command

  \[
  \text{DROP TABLE table name }, \quad \text{CASCADE, RESTRICT}
  \]

  - if RESTRICT is used, you may only drop empty tables that are not referenced by any other table
  - if CASCADE is used, all referencing tables are also deleted (including all stored rows)
    - DB2 does not support CASCADE ...
  - if neither is used, the table does not have to be empty, but must not be referenced by another one
  - example
    - DROP TABLE hero CASCADE, power CASCADE
• After a table has been created, you may alter it by adding/removing columns or constraints.
9.1 SQL DDL: Alter Tables

– if you add a **new column** with a **NOT NULL** constraint, you also need to provide a **default value**

– when **dropping** a **column**, you must either choose
  - **CASCADE** to also delete any views, indexes, and constraints dependent on that column
  - **RESTRICT** to allow the drop only if there is no referring column (default)

– if the **name** of a constraint is **auto-generated**, you need to look it up in the **system catalog**

– example
  - **ALTER TABLE** `has_power` **DROP** `power_strength`
  - **ALTER TABLE** `has_power` **ADD COLUMN** `since` **DATE**
• SQL data definition language
• SQL data manipulation language (apart from SELECT)
• SQL ≠ SQL
• Some advanced SQL concepts
9.2 SQL DML

- **Data definition language** (DDL)
  - creating, changing, altering schemas, tables, ...
    - `CREATE SCHEMA`
    - `CREATE TABLE`
    - `ALTER TABLE`
    - `DROP TABLE`

- **Data manipulation language** (DML)
  - querying
    - `SELECT`
  - adding and updating data
    - `INSERT INTO`
    - `UPDATE`
    - `DELETE`
9.2 SQL DML: Insert

• Now we have wonderful, empty tables
• We need to put data into them!
  – INSERT INTO statement
  – you can specify into what columns you want to insert data
    • default: all columns
  – new values are stated as a literal table or inline view (query)
    • of course the attribute domains have to match
9.2 SQL DML: Insert

- A literal table is defined extensionally:

VALUES ("James", "Howlet")

VALUES ("Charles", "Xavier"), ("Jean", "Grey")

VALUES "Wolverine", ("Professor X"), "Phoenix"
9.2 SQL DML: Insert

- \textbf{INSERT INTO} hero(id, real_name) VALUES
  (1, 'Charles F. Xavier'),
  (2, 'Jean Grey')

- \textbf{INSERT INTO} has_alias VALUES
  (1, 'Professor X'),
  (1, 'Onslaught'),
  (2, 'Phoenix'),
  (2, 'Marvel Girl')
• Of course, subqueries may also be used in \textbf{INSERT} statements

\begin{verbatim}
- \textbf{INSERT INTO} heroes_starting_with_a(
  SELECT * FROM hero
  WHERE real_name LIKE 'A%'
)
\end{verbatim}
9.2 SQL DML: Update

- Existing rows can also be changed using the **UPDATE** statement
  - very similar to the **SELECT** statement
  - update **finds rows** fulfilling a **given condition** and **changes** some of its **rows** by assigning **new values**
9.2 SQL DML: Update

\[
\text{UPDATE}\ \text{table name}\ \text{AS}\ \text{alias name} \\
\text{SET}\ \text{column name}\ =\ \text{expression} \\
\text{WHERE}\ \text{search condition}
\]
9.2 SQL DML: Update

- Replace the real name of each hero with NULL
  - `UPDATE hero SET real_name = NULL`

- Multiply all `power_strength` values by 10
  - `UPDATE has_power SET power_strength = power_strength * 10`

- Change the name of hero with id 1
  - `UPDATE hero SET name = 'Charles Francis Xavier' WHERE id = 1`

- Change name and id of Jean Grey
  - `UPDATE hero SET (id, name) = ('3', 'Jean Grey-Summers') WHERE name = 'Jean Grey'`
  - Change of id is propagated to other tables when `ON UPDATE CASCADE` is used in table definition

- Again, subqueries can be used in the `WHERE` clause
The **DELETE statement** is used to delete rows from a table

- deletes all rows satisfying a certain search condition
- example
  - delete *Jean Grey*
    - `DELETE FROM hero WHERE name = 'Jean Grey'`
  - delete all heroes
    - `DELETE FROM hero`
• Again, subqueries can be used here
  
  ```
  DELETE FROM hero h
  WHERE NOT EXISTS ( 
    SELECT * FROM has_alias a
    WHERE a.hero = h.id
  )
  ```
• SQL data definition language
• SQL data manipulation language (apart from \texttt{SELECT})
• \textbf{SQL $\neq$ SQL}
• Some advanced SQL concepts
9.3 SQL ≠ SQL

• First, the **good news:**
  – SQL has been standardized by the ISO in 1987
  – the standard is well-maintained and under active development
  – many *big* database vendors participate in the standardization process
    • IBM, Ingres, Microsoft, Oracle, Sybase, ...
A timeline of SQL standardization:

- **1986**
  - ANSI SQL
  - relations, attributes, views
  - `SELECT ... FROM ... WHERE ...`

- **1987**
  - SQL-86 (ISO/IEC 9075:1986)

- **1989**
  - SQL-89 (SQL1)
    - ≈ SQL-86 + restricted referential integrity
9.3 SQL ≠ SQL

• 1992
  – SQL-92 (SQL2)
  – 3 parts, 1120 pages
  – Entry Level
    • ≈ SQL-89 + CHECK (attribute)
  – Intermediate Level
    • ⊇ Entry Level + domains, CHECK (relation), CASE, CAST, JOIN, EXCEPT, INTERSECT
  – Full Level
    • ⊇ Intermediate Level + assertions, nested select, nested from
9.3 SQL ≠ SQL

• 1999/2000
  – SQL:1999 (SQL3)
  – 5 parts, 2084 pages
  – ≈ SQL-92 + object-orientation, recursive queries, triggers, OLAP, user-defined types, regular expressions
    • Boolean data type
  – computationally complete, object-oriented database programming language, descriptive and procedural
  – core (about 180 features)
    • ≈ SQL92 Entry Level + parts of Intermediate and Full Level
  – 9 Packages (about 240 features)
    • enhanced date/time, enhanced integrity, OLAP, PSM, CLI, basic object support, enhanced object support, trigger, SQL/MM
Recursive queries (SQL:1999):

- How to find all rivers that flow into the North Sea?
- Multiple joins?

<table>
<thead>
<tr>
<th>flows_into</th>
<th>river</th>
<th>mouth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oker</td>
<td>Aller</td>
<td></td>
</tr>
<tr>
<td>Aller</td>
<td>Weser</td>
<td></td>
</tr>
<tr>
<td>Weser</td>
<td>North Sea</td>
<td></td>
</tr>
<tr>
<td>Elbe</td>
<td>North Sea</td>
<td></td>
</tr>
<tr>
<td>Edder</td>
<td>Wietze</td>
<td></td>
</tr>
<tr>
<td>Flöth</td>
<td>Wietze</td>
<td></td>
</tr>
<tr>
<td>Wietze</td>
<td>Aller</td>
<td></td>
</tr>
<tr>
<td>Isar</td>
<td>Danube</td>
<td></td>
</tr>
<tr>
<td>Inn</td>
<td>Danube</td>
<td></td>
</tr>
</tbody>
</table>
9.3 SQL ≠ SQL

• Solution

```sql
WITH RECURSIVE flows_into_ns(river) AS (  
    SELECT river  
    FROM flows_into  
    WHERE mouth = 'North Sea'  
    
    UNION  
    
    SELECT fi.river  
    FROM flows_into AS fi  
    JOIN flows_into_ns AS fins  
    ON fi.mouth = fins.river
)  
SELECT river FROM flows_into_ns
```

• **Note:** DB2 uses a different syntax
  – No `RECURSIVE`, `UNION ALL` instead of `UNION`, no explicit `JOIN`
9.3 SQL ≠ SQL

• 2003
  – SQL:2003
  – 14 parts, 3606 pages
  – **MULTISET** as an explicit construct (with numerous operations, such as: **MULTISET UNION, MULTISET EXCEPT, MULTISET INTERSECT, CARDINALITY**)
  – sequence generators
    • **CREATE SEQUENCE** `<sequence name> AS <type name>`
    [START WITH <value>] [INCREMENT BY <value>]
    [NO MINVALUE | MINVALUE <value>]
    [NO MAXVALUE | MAXVALUE <value>]
    [NO CYCLE | CYCLE]
    • Many databases use non-standard syntax for this task…
  – base type XML for mappings between SQL and XML
  – **CREATE TABLE AS / CREATE TABLE LIKE**
9.3 SQL ≠ SQL

• 2006
  – SQL:2006
  – successor of SQL:2003
  – a new extension for XML handling
    • importing, storing, querying, and manipulating XML data
    • support for XQuery
    • concurrently access (object-)relational data and XML data

• 2008
  – SQL:2008
  – successor of SQL:2006
  – maintenance release (some new but minor features)
• 2011
  – SQL:2011
  – successor of SQL:2008
  – adds support for **time periods**

```sql
CREATE TABLE mutation(
    name VARCHAR(255) PRIMARY KEY NOT NULL,
    person INTEGER REFERENCES person(id),
    from DATE,
    until DATE,
    PERIOD FOR period_of_validity (from, until)));

SELECT name, person FROM mutation
WHERE period_of_validity CONTAINS '2014-12-19';
```

– temporal predicates, temporal primary keys, temporal referential integrity
  • This might be very useful for work planning or scheduling applications
• Well, here are the **bad news**
  – there are still **too many variants** of SQL
    (both **syntactic and semantic differences**)  
      • true application portability remains a challenge
  – the standard has been used to introduce
    **two kinds of features**
    1. features that are well-understood and widely implemented
    2. new and largely untried technologies, hoping that vendors follow the lead and deliver new functionalities
  – **vendors don’t care** too much about the standard
• A common myth among software developers

If your application uses only standard SQL, then it is portable.

• If you don’t believe me, here are some examples ...
• **CREATE TABLE** name (  
  first **VARCHAR**(100),  
  middle **VARCHAR**(100),  
  last **VARCHAR**(100)  
)

**INSERT INTO** name **VALUES** ( 'George', 'Walker', 'Bush' )
**INSERT INTO** name **VALUES** ( 'Horst', '', 'Kr' )
**INSERT INTO** name **VALUES** ( 'Angela', **NULL**, 'Merkel' )

- '' (empty string) means that we know that there is no middle name
- **NULL** means that we don’t know whether there is a middle name

• Sounds like a good design? What do you think?
  - according to the SQL standard, this approach is fine ...
  - ... unless your RDBMS is Oracle ('' is the same as **NULL**)
• What about terminology?
  – the SQL standard defines the following notions
    • Environment
    • Cluster
    • Catalog
    • Schema
  – the reality
    • Database server
    • (unsupported)
    • Database
    • Schema
  – but attention
    • in MySQL, there are no catalogs, schema and database are synonyms
    • in Oracle, there is exactly one schema per user.
      \texttt{CREATE/ALTER SCHEMA x <command>} executes \texttt{<command>}
      on all objects located in schema x
• The **statement terminator** ;

  – according to the SQL standard, (almost) every SQL statement has to be terminated by a semicolon

  – **What’s happening in practice?**

    • many RDBMS treat the terminator as being optional (which is fine, but may cause some problems)

    • some RDBMS either strictly require a terminator or complain if it is present

    • in some RDBMS, this behavior can be configured ...

  – summary:

    **No matter what you do, it causes problems!**
9.3 SQL ≠ SQL

• The BOOLEAN data type

• `CREATE TABLE` customers (  
  id INTEGER PRIMARY KEY,  
  name VARCHAR(100),  
  is_vip BOOLEAN,  
  is_blacklisted BOOLEAN  
)

`SELECT` id, name `FROM` customers  
`WHERE` is_vip AND NOT is_blacklisted

– practice?

• not supported by Oracle, DB2, and MS SQL Server  
  – official workarounds: use CHAR or INTEGER ...

• supported by MySQL and PostgreSQL  
  – where in MySQL BOOLEAN is just a short hand for TINYINT(1) ...
9.3 SQL ≠ SQL

• **Summary**
  - SQL is not SQL
  - in some cases, even **identical SQL statements work differently** on different RDBMS

• **Current trends?**
  - open-source RDBMS (PostgreSQL, MySQL, Derby, ...) typically try to adhere to the standard
    • however, many advanced features are not supported yet
  - recently, DB2 added support for Oracle’s SQL (in DB2 9.7)
• SQL data definition language
• SQL data manipulation language (apart from SELECT)
• SQL ≠ SQL
• Some advanced SQL concepts
9.4 Type Casting

• SQL is a **strongly typed** language
  – Basically, this means that e.g. `INTEGER` is different from `VARCHAR(100)`

• If data types are incompatible, **type casting** may be used to make them compatible
  – `CAST` expression
  – during casting, precision may be lost (e.g. `FLOAT → INTEGER`)
  – example

  • `CAST (power_strength AS NUMERIC(3, 2))`
  • `CAST (alias || real_name AS CHAR(255))`

CAST expression

```
CAST (expression AS data type)
```

`CAST` expression

```
CAST (expression AS data type)
```

CAST

(expression)

AS

data type

NULL

CAST
9.4 Type Casting

• In DB2, possible castings are

<table>
<thead>
<tr>
<th>Source</th>
<th>Possible Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT, INTEGER, DECIMAL, FLOAT</td>
<td>SMALLINT, INTEGER, DECIMAL, FLOAT</td>
</tr>
<tr>
<td>CHAR, VARCHAR, LONG VARCHAR, CLOB</td>
<td>CHAR, VARCHAR, LONG VARCHAR, CLOB, BLOB</td>
</tr>
<tr>
<td>CHAR, VARCHAR</td>
<td>SMALLINT, INTEGER, DECIMAL, DATE, TIME, TIMESTAMP, VARGRAPHICS</td>
</tr>
<tr>
<td>SMALLINT, INTEGER, DECIMAL</td>
<td>CHAR</td>
</tr>
<tr>
<td>DATE, TIME, TIMESTAMP</td>
<td>CHAR, VARCHAR</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>GRAPHICS, VARGRAPHICS, LONG VARGRAPHICS, DBCLOB</td>
<td>GRAPHICS, VARGRAPHICS, LONG VARGRAPHICS, DBCLOB</td>
</tr>
</tbody>
</table>
9.4 Ranking Functions

• Since SQL:2003, there are special functions for working with **result lists**

• Examples
  – output only every other row of the list
  – create a ranking with explicit ranks (1, 2, 3, ...)
  – on what rank position is some given row?
9.4 Ranking Functions

- \texttt{ROW\_NUMBER()} returns the \textbf{position} of each row in the result list.

**Example**

```sql
SELECT name, salary, 
    \texttt{ROW\_NUMBER()}\over ( 
    \texttt{ORDER BY salary DESC}
) \texttt{AS pos}
FROM salary
```

<table>
<thead>
<tr>
<th>person</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simon</td>
<td>45000</td>
<td></td>
</tr>
<tr>
<td>Wolf-Tilo</td>
<td>75000</td>
<td></td>
</tr>
<tr>
<td>Larry</td>
<td>200000000</td>
<td></td>
</tr>
<tr>
<td>Christoph</td>
<td>50000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
<th>pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larry</td>
<td>200000000</td>
<td>1</td>
</tr>
<tr>
<td>Wolf-Tilo</td>
<td>75000</td>
<td>2</td>
</tr>
<tr>
<td>Christoph</td>
<td>50000</td>
<td>3</td>
</tr>
<tr>
<td>Simon</td>
<td>45000</td>
<td>4</td>
</tr>
</tbody>
</table>

Depending on the implementation, the last two rows may switch positions.
example: At which position is Wolf-Tilo?

- SELECT name, salary, 
  "ROW_NUMBER() OVER ( 
    ORDER BY salary DESC 
  ) AS pos 
FROM salary 
WHERE name = 'Wolf-Tilo'

<table>
<thead>
<tr>
<th>person</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simon</td>
<td>45000</td>
<td></td>
</tr>
<tr>
<td>Wolf-Tilo</td>
<td>75000</td>
<td></td>
</tr>
<tr>
<td>Larry</td>
<td>200000000</td>
<td></td>
</tr>
<tr>
<td>Christoph</td>
<td>50000</td>
<td></td>
</tr>
</tbody>
</table>
### 9.4 Ranking Functions

**Example:** Show only rows at even positions.

- `SELECT name, salary, ROW_NUMBER() OVER (ORDER BY salary DESC) AS pos FROM salary WHERE (pos % 2) = 0`

<table>
<thead>
<tr>
<th>person</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simon</td>
<td>45000</td>
<td></td>
</tr>
<tr>
<td>Wolf-Tilo</td>
<td>75000</td>
<td></td>
</tr>
<tr>
<td>Larry</td>
<td>200000000</td>
<td></td>
</tr>
<tr>
<td>Philipp</td>
<td>45000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
<th>pos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wolf-Tilo</td>
<td>75000</td>
<td>2</td>
</tr>
<tr>
<td>Simon</td>
<td>45000</td>
<td>4</td>
</tr>
</tbody>
</table>
9.4 Ranking Functions

- **RANK()** returns the rank of each row in the result list

  - example

    ```sql
    SELECT name, salary, RANK() OVER (ORDER BY salary DESC) AS rank
    FROM salary
    ```

<table>
<thead>
<tr>
<th>person</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simon</td>
<td></td>
<td>45000</td>
</tr>
<tr>
<td>Wolf-Tilo</td>
<td></td>
<td>75000</td>
</tr>
<tr>
<td>Larry</td>
<td></td>
<td>200000000</td>
</tr>
<tr>
<td>Phil</td>
<td></td>
<td>45000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larry</td>
<td>200000000</td>
<td>1</td>
</tr>
<tr>
<td>Wolf-Tilo</td>
<td>75000</td>
<td>2</td>
</tr>
<tr>
<td>Phil</td>
<td>45000</td>
<td>3</td>
</tr>
<tr>
<td>Simon</td>
<td>45000</td>
<td>3</td>
</tr>
</tbody>
</table>
9.4 Ranking Functions

- **DENSE_RANK()** works like **RANK()** but does not skip ranks on ties (as it is usually done)

  - example
    ```sql
    SELECT name, salary, RANK() OVER (ORDER BY salary ASC) AS rank,
           DENSE_RANK() OVER (ORDER BY salary ASC) AS drank
    FROM salary
    ```

<table>
<thead>
<tr>
<th>person</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simon</td>
<td>45000</td>
<td></td>
</tr>
<tr>
<td>Wolf-Tilo</td>
<td>75000</td>
<td></td>
</tr>
<tr>
<td>Larry</td>
<td>200000000</td>
<td></td>
</tr>
<tr>
<td>Philipp</td>
<td>45000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
<th>rank</th>
<th>drank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simon</td>
<td>45000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Philipp</td>
<td>45000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Wolf-Tilo</td>
<td>75000</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Larry</td>
<td>200000000</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
9.4 CASE Expressions

• Very often **codes** are used for storing more complex information
  – retrieving the account information for owner *Clark* with appropriate account descriptions needs a join
  – Indicate all customers with a negative balance with the string *not creditworthy* in the query result

<table>
<thead>
<tr>
<th>account</th>
<th>owner</th>
<th>balance</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clark</td>
<td>Clark</td>
<td>367,00</td>
<td>0</td>
</tr>
<tr>
<td>Louis</td>
<td>-675,00</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Clark</td>
<td>54987,00</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>acc_type</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>checking account</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>savings account</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>credit card account</td>
<td></td>
</tr>
</tbody>
</table>
• The **CASE expression** allows a value to be selected based on the evaluation of one or more conditions (similar to *if-then-else*)
  – comes in two flavors
9.4 CASE Expressions

– the **simple WHEN** clause

  • compares an expression to each case expression one by one
    – if expression is equal to search value, the corresponding result expression is returned

  • if no match is found, then some default (**ELSE** clause) is returned
    – if **ELSE** is omitted, then **NULL** is returned
9.4 CASE Expressions

• **Example:** simple *WHEN* clause
  – directly decode the account type

  ```sql
  SELECT owner, 
  CASE type 
    WHEN 0 THEN 'checking account'
    WHEN 1 THEN 'savings account'
    WHEN 2 THEN 'credit card account'
  END AS verbose_type 
FROM account
  ```

<table>
<thead>
<tr>
<th>account</th>
<th>owner</th>
<th>balance</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clark</td>
<td>367,00</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Louis</td>
<td>-675,00</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Clark</td>
<td>54987,00</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
9.4 CASE Expressions

– the searched **WHEN** clause

• checks search conditions from left to right
• **stop**s as soon as a **search condition evaluates to true**
  – returns the corresponding result then
• if **no condition is true**, the value given by the **ELSE** clause is returned (or **NULL**, if there is no **ELSE** clause)
9.4 CASE Expressions

- **Example:** searched *WHEN* clause
  - retrieve credit rating of customers based on their checking accounts
  - \texttt{SELECT owner,}
    \texttt{CASE}
    \texttt{WHEN balance < 0 \textbf{THEN} 'not credit-worthy'}
    \texttt{WHEN balance = 0 \textbf{THEN} 'questionable'}
    \texttt{ELSE 'credit-worthy'}
    \texttt{END AS credit_worthyness}
  \texttt{FROM account}
  \texttt{WHERE type = 0}

<table>
<thead>
<tr>
<th>account</th>
<th>owner</th>
<th>balance</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clark</td>
<td>367,00</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Louis</td>
<td>-675,00</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Clark</td>
<td>54987,00</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
There are many more SQL statements than we are covering in our lectures.

Moreover, there are many different SQL dialects.

If you don’t want to get mad, do the following:

– don’t care too much about the SQL standard (unless you are actually implementing an RDBMS)
– read the SQL manuals of your RDBMS!
9 More on SQL

- **Example:** DB2’s SQL reference (version 10.5)
• Normalization
• Functional Dependencies
• Normal Forms 
  – 1NF, 2NF, 3NF, BCNF, 4NF, 5NF, 6NF

It's that simple!