Relational Database Systems 1
Wolf-Tilo Balke
Joachim Selke
Institut für Informationssysteme
Technische Universität Braunschweig
www.ifis.cs.tu-bs.de

Overview
• Implementing the relational model
• SQL
  – Queries
  – Data manipulation language (next lecture)
    • SELECT
    • INSERT
    • UPDATE
    • DELETE
  – Data definition language (next lecture)
    • CREATE TABLE
    • ALTER TABLE
    • DROP TABLE

Exercise 6.1
• Give relational algebra expressions for the following queries in natural language:
  a) Create a list of all hotels that includes the total number of rooms for each hotel.
     \[ \pi_{\text{hotelName}, \text{count(roomNo)}}(\text{hotelNo}, \text{hotelName} \downarrow \text{count(roomNo)}) (\text{Hotel} \bowtie \text{Room}) \]
    • Of course, this does not work for hotels without any rooms. However, let’s assume that there is no such hotel.
  b) What’s the average room price at the Balke Inn?
     \[ \pi_{\text{average(price)}}(\text{hotelName} = \text{Balke Inn}) (\text{Hotel} \bowtie \text{Room}) \]

Exercise 6.2
• Describe (using natural language) the relations that would be produced by the following tuple relational calculus expressions:
  a) \{ H.hotelName | hotel(H) \land H.city = ‘Braunschweig’ \}
     List the names of all hotels in Braunschweig!
  b) \{ H.hotelName | hotel(H) \land \exists R ( Room(R) \land H.hotelNo = R.hotelNo \land R.price > 50) \}
     List the names of all hotels offering a room that costs more than €50!

Exercise 6.1 Solutions
• A hotel database:
  Hotel(hotelNo, hotelName, city)
  Room(roomNo, hotelNo → Hotel, type, price)
  Booking(hotelNo → Hotel/Room, guestNo → Guest, dateFrom, dateTo, roomNo → Room)
  Guest(guestNo, guestName, guestAddress)

c) How many different people made a reservation for a room whose price is higher than the average room price at the respective hotel?
\[
\begin{align*}
\text{AvgPrice} &:= \pi_{\text{hotelNo, average price}}(\text{hotelNo} \downarrow \text{average(price)} (\text{Hotel} \bowtie \text{Room})) \\
\text{ExpensiveRooms} &:= \pi_{\text{hotelNo, roomNo}}(\text{price} > \text{average price}(\text{Room} \bowtie \text{AvgPrice})) \\
\text{RichGuests} &:= \pi_{\text{guestNo}}(\text{Booking} \bowtie \text{ExpensiveRooms}) \\
\text{Result} &= \text{count(\text{guestNo})} \cdot \text{RichGuests}
\end{align*}
\]
Exercise 6.2

c) \{ H.hotelName | Hotel(H) \land \exists B \exists G (Booking(B) \land Guest(G) \land H.hotelNo = B.hotelNo \land B.guestNo = G.guestNo \land G.guestName = "Wolf-Tilo Balke") \}

List the names of all hotels in that Wolf-Tilo Balke has or had a reservation!

Exercise 6.2

d) \{ H.hotelName, G.guestName, B.dateFrom, B.dateTo | Hotel(H) \land Guest(G) \land Booking(B) \land Booking(B) \land H.hotelNo = B.hotelNo \land G.guestNo = B.guestNo \land B.guestNo = G.guestNo \land B.dateFrom = B.dateTo \}

List the names of all hotels and guests that have or had different reservations at the same hotel; for each hotel and guest, also list all possible pairs of starting dates for the corresponding reservations.

• Hint: (hotelNo, guestNo, dateFrom) is the primary key of Booking, which makes each result line a pair of reservations.

Exercise 6.3

• Generate the relational algebra, tuple relational calculus, and domain relational calculus expressions for the following queries:

  a) List all hotels.

  RA: \( \pi_{hotelName}(Hotel) \)
  TRC: \{ h.hotelName | Hotel(h) \}
  DRC: \{ \{ h1 | \exists h1, h2, h3 | Hotel(h1, h2, h3) \} \}

  b) List all single rooms with a price below €20 per night.

  RA: \( \pi_{hotelName, city, roomNo}(\sigma_{type = 'single' \land price < 20}(Room \bowtie Hotel)) \)
  TRC: \{ h.hotelName, h.city, r.roomNo | Hotel(h) \land Room(r) \land r.type = 'single' \land r.price < 20 \land h.hotelNo = r.hotelNo \}
  DRC: \{ h2, h3, r1 | \exists h1, r1 | Room(r1, h1, 'single', r1) \land r1 < 20 \land Hotel(h1, h2, h3) \}

  c) List the price and type of all rooms at the Balke Inn.

  RA: \( \pi_{roomNo, type, price}(Room \bowtie \sigma_{hotelName = 'Balke Inn'}(Hotel)) \)
  TRC: \{ r.roomNo, r.roomType, r.price | Room(r) \land \exists h | (Hotel(h) \land h.hotelName = 'Balke Inn' \land h.hotelNo = r.hotelNo) \}
  DRC: \{ r1, r2, r3 | \exists h1, h2 | (Room(r1, h1, r2, r3) \land Hotel(h1, 'Balke Inn', h2)) \}

  d) List the details (guestNo, guestName, and guestAddress) of all guests currently staying at the Balke Inn.

  RA: \( Guest \bowtie \sigma_{dataFrom \leq TODAY \land dataTo \geq TODAY}(Booking) \bowtie \sigma_{hotelName = 'Balke Inn'}(Hotel) \)
  TRC: \{ g | \exists b, h | (Booking(b) \land Hotel(h) \land b.guestNo = g.guestNo \land b.hotelNo = h.hotelNo \land b.dataFrom \leq TODAY \land b.dataTo \geq TODAY \land h.hotelName = 'Balke Inn') \}

Exercise 6.3
Exercise 6.3

DRC: \{ (g_1, g_2, g_3) \mid \text{Guest}(g_1, g_2, g_3) \land \exists h_1, h_2, b_1, b_2 \} \\
\forall (h_1, h_2, \text{Balke Inn', } h_3) \\
\land \text{Booking}(h_1, g_1, b_1, b_2) \land b_1 \leq \text{TODAY} \\
\land b_2 \geq \text{TODAY} \} 

RA: \Pi_{\text{roomNo}, \text{type}, \text{price}, \text{guestName}}(\text{Room}) \\
\forall (h_1, \text{hotelName} = \text{Balke Inn'}) \in (\text{Hotel}) \\
\forall (\text{dateFrom} \leq \text{TODAY} \land \text{dateTo} \geq \text{TODAY}) \in (\text{Booking}) \\
\forall \text{Guest}() 

Exercise 6.3

TRC:

BalkeRoom(r) := Room(r) \land \exists h (\text{Hotel}(h) \\
\land h.\text{hotelName} = \text{Balke Inn'} \land h.\text{hotelNo} = r.\text{hotelNo}) \\
CurrentBooking(g, r) := \text{Guest}(g) \land \exists b (\text{Booking}(b) \\
\land b.\text{hotelNo} = r.\text{hotelNo} \land b.\text{roomNo} = r.\text{roomNo} \\
\land b.\text{dateFrom} \leq \text{TODAY} \land b.\text{dateTo} \geq \text{TODAY} \\
\land b.\text{guestNo} = g.\text{guestNo}) \\
\{ (r.\text{roomNo}, r.\text{type}, r.\text{price}, g.\text{guestName} \mid \text{BR}(r) \land \\
(\text{CB}(g, r) \lor (\exists g'. \text{CB}(g', r) \land g = (\text{null, null, null, null}))) \}

7.1 From Theory to Practice

- In the early 1970s, the relational model became a “hot topic” database research
  - Based on set theory
  - A relation is a subset of the Cartesian product over a list of domains
- Early “query interfaces” for the relational model:
  - Relational algebra
  - Tuple relational calculus (SQUARE, SEQUEL)
  - Domain relational calculus (QBE)
- Question: How to build a working database management system using this theory?

7.1 From Theory to Practice

- System R was the first working prototype of a relational database system (starting 1973)
  - Most design decisions taken during the development of System R substantially influenced the design of subsequent systems
- Questions
  - How to store and represent data?
  - How to query for data?
  - How to manipulate data?
  - How do you do all this with good performance?
7.1 From Theory to Practice

• Design decisions:
  During the development of System R, two major and very controversial decisions had been made
  – Allow duplicate tuples
  – Allow NULL values
  Those decisions are still subject to discussions…

• Practical considerations
  – You want to query for name and birth year of all students of TU Braunschweig
  – The result returns roughly 13,000 tuples
  – Probably there are some duplicates
  – It’s 1973, and your computer has 16 kilobytes of main memory and a very slow external storage device…
  – To eliminate duplicates, you need to store the result, sort it, and scan for adjacent duplicate lines
    • System R engineers concluded that this effort is not worth the effort.
    • Duplicate elimination in result sets happens only on request.

• Decision: Don’t eliminate duplicates in results
  • What about the tables?
    – Again: Ensuring that no duplicates end up in the tables requires some work.
    – Engineers also concluded that there is actually no need in enforcing the no-duplicate policy.
      • If the user wants duplicates and is willing to deal with all the arising problems then that’s fine.
  • Decision: Allow duplicates in tables
    • As a result, the theory underlying relational databases shifted from set theory to multi-set theory
      – Straightforward, only notation is more complicated.

• Sometimes, an attribute value is not known or an attribute does not apply for an entity
  – Example: What value should the attribute universityDegree take for the entity Heinz Müller if Heinz Müller does not have any degree?
  – Example: You regularly observe the weather and store temperature, wind strength, and air pressure every hour and then your barometer breaks… What now?

• Possible solution:
  For each domain, define a value indicating that data is not available, not known, not applicable,…
  – For example, use none for Heinz Müller’s degree, use −1 for missing pressure data,…
  – Problem:
    • You need such a special value for each domain or use case.
    • You need special failure handling for queries, e.g., “compute average of all pressure values that are not −1.”
7.1 From Theory to Practice

• Again, system designers chose the simplest solution (regarding implementation): **NULL values**
  – NULL is a special value which is usable in any domain and represents that data is just there
    • There are many interpretations of what NULL actually means
  – Systems have some default rules how to deal with NULL values
    • Aggregation functions usually ignore rows with NULL values (which is good in most, but not all cases)
    • Three-valued logic
    • However, creates some strange anomalies

7.2 SQUARE & SEQUEL

• The design of relational query languages
  – Donald D. Chamberlin and Raymond F. Boyce worked on this task
  – Both of **IBM Research** in San Jose, California
  – Main concern: "**Querying** relational databases is **too difficult** with current paradigms"

• Chamberlin and Boyce’s first result was a query language called **SQUARE**
  – “**Specifying queries as relational expressions**”
  – Based directly on **tuple relational calculus**
  – Main observations:
    • Most database queries are rather simple
    • Complex queries are rarely needed
    • Quantification confuses people
    • Under the closed-world assumption, any TRC expression with quantifiers can be replaced by a join of quantifier-free expressions

7.1 From Theory to Practice

• Another tricky problem: How should users **query** the DB?
  • Classical answer:
    – Relational algebra and relational calculi
    – **Problem**: More and more **non-expert users**
  • More “natural” query interfaces:
    – **QBE** (query by example)
    – **SEQUEL** (structured English query language)
    – **SQL**: the current standard, derived from SEQUEL

7.2 SQUARE & SEQUEL

• “Current paradigms” at the time:
  – **Relational algebra**
    • Requires users to define how and in which order data should be retrieved
    • The specific choice of a sequence of operations has an enormous influence on the system’s performance
  – **Relational calculi (tuple, domain)**
    • Provide declarative access to data, which is good
    • Just state what you want and not how to get it
    • Relational calculi are quite complex:
      many variables and quantifiers

7.2 SQUARE & SEQUEL

• **SQUARE** is a notation for (or interface to) TRC
  – **No quantifiers**, implicit notation of variables
  – Adds **additional functionality** needed in practice (grouping, aggregating, among others)
  – Solves **safety problem** by introducing the closed world assumption
7.2 SQUARE & SEQUEL

- Retrieve the names of all female students
  - TRC: \{ t.name | students(t) \land t.sex = 'f' \}
  - SQUARE: namestudentssex ('f')

- Get all exam results better than 2.0
  - TRC: \{ t.result | exams(t) \land crsNr = 101 \land t.result < 2.0 \}
  - SQUARE: resultexamsresult (101, < 2.0)

- Get a list of all exam results better than 2.0 along with the according student name
  - TRC:
    \{ t1.name, t2.result | students(t1) \land exams(t2) \land t1.matNr = t2.matNr \land t2.result < 2.0 \}
  - SQUARE:
    name result studentsmatNr \land examsresult (< 2.0)

- Also, SQUARE is relationally complete
  - You do not need explicit quantifiers
  - Everything you need can be done using conditions and query combining

- However, SQUARE was not well received
  - Syntax was difficult to read and parse, especially when using text console devices:
    - name result matNr = matNr exams crsNr result (102, < 2.0)
  - SQUARE's syntax is too mathematical and artificial

- Fundamental keywords
  - SELECT: What attributes should be retrieved?
  - FROM: What relations are involved?
  - WHERE: What conditions should hold?

- Get all exam results better than 2.0 for course 101
  - SQUARE:
    resultexamsresult (101, < 2.0)
  - SEQUEL:
    SELECT result FROM exams WHERE crsNr = 101 \land result < 2.0

- In 1974, Chamberlin & Boyce proposed SEQUEL
  - Structured English Query Language
  - Based on SQUARE

- Guiding principle:
  - Use natural English keywords to structure queries
  - Supports "fluent" vocalization and notation

- Also, \cup, \cap, and \setminus can be used to combine SQUARE queries.
7.2 SQUARE & SEQUEL

- Get a list of all exam results better than 2.0, along with the according student names
  - SQUARE:
    ```sql
    SELECT name, result
    FROM students, exams
    WHERE students.matNr = exams.matNr
    AND result < 2.0
    ```

- SEQUEL:
  ```sql
  SELECT name, result
  FROM students, exams
  WHERE students.matNr = exams.matNr
  ```

7.2 SQUARE & SEQUEL

- Since then, SQL has been adopted by all (?) relational database management systems
- This created a need for standardization:
  - 1986: SQL-86 (ANSI standard, ISO standard)
  - The official pronunciation is "sequence el"
- However, most database vendors treat the standard as some kind of "recommendation"
  - More on this later (next lecture)

7.3 Overview of SQL

- According to the SQL standard, relations and other database objects exist in an environment
  - Think "environment = RDBMS"
- Each environment consists of catalogs
  - Think "catalog = database"
- Each catalog consists of a set of schemas
  - Think "schema = group of tables (and other stuff)"
- A schema is a collection of database objects (tables, domains, constraints, ...)
  - Each database object belongs to exactly one schema
  - Schemas are used to group related database objects

7.3 Overview of SQL

- IBM integrated SEQUEL into System R
- It proved to be a huge success
  - Unfortunately, the name SEQUEL already has been registered as a trademark by the Hawker Siddeley aircraft company
  - Name has been changed to SQL (spoken: Sequel)
  - Structured query language
  - Patented in 1985 by IBM

7.3 Overview of SQL

- There are three major classes of DB operations:
  - Defining relations, attributes, domains, constraints, ...
  - Data definition language (DDL)
  - Adding, deleting and modifying tuples
  - Data manipulation language (DML)
  - Asking queries
    - Often part of the DML
- SQL covers all these classes
- In this lecture, we will use IBM DB2's SQL dialect
  - DB2 is used in our SQL lab
  - Similar notation in other RDBMS (at least for the part of SQL we teach you in this lecture)
7.3 Overview of SQL

- When working with the environment, users connect to a single catalog and have access to all database objects in this catalog.
  - However, accessing/combining data objects from different catalogs usually is not possible.
  - Thus, typically, catalogs are the maximum scope over that SQL queries can be issued.
  - In fact, the SQL standard defines an additional layer in the hierarchy on top of catalogs:
    - Clusters are used to group related catalogs.
    - According to the standard, they provide the maximum scope.
    - However, hardly any vendor supports clusters.

- After connecting to a catalog, database objects can be referenced using their qualified name:
  - Qualified name = schemaname.objectname
- However, when working only with objects from a single schema, using unqualified names would be nice:
  - Unqualified name = objectname
- One schema always is defined to be the default schema:
  - SQL implicitly treats objectname as defaultschema.objectname
  - The default schema can be set to schemaname as follows:
    - SET SCHEMA schemaname
- Initially, after login: default schema = current user name
  - Remember to change the default schema accordingly!

7.4 SQL Queries

- SQL queries:
  - Simple SELECT
  - Joins
  - Set operations
  - Aggregation and grouping
  - Subqueries
  - Writing "good" SQL code

- Basic structure of SQL queries:
  - SELECT <attribute list>
    FROM <table list>
    WHERE <condition>
  - Attribute list: Attributes to be returned (projection)
  - Table list: All tables involved
  - Condition: A Boolean expression that intentionally defines the result set
    - If no condition is provided, it is implicitly replaced by TRUE

- SQL performs duplicate elimination of rows in result set:
  - May be expensive (due to sorting)
  - DISTINCT keyword is used

- Example:
  - SELECT DISTINCT name FROM staff
    - Returns all different names of staff members, without duplicates
7.4 Attribute Names

- The **SELECT keyword** is sometimes a bit confusing, since it actually denotes a **projection**.
- To return all attributes under consideration, the wildcard "*" may be used.
- Examples:
  - `SELECT * FROM ...` 
    Return all attributes of the tables in the FROM clause.
  - `SELECT movies.* FROM movies, persons WHERE ...` 
    Return all attributes of the movies table.

7.4 Table Names

- Attribute names are qualified or unqualified:
  - **Unqualified:** Just the attribute name.
  - **Qualified:** `tablename.attributename`
  - Only possible, if attribute name is unique among the tables given in the FROM clause.
  - Necessary if tables share identical attribute names.
  - If tables in the FROM clause share identical attribute names and also identical table names, we need even more qualification: `tablename.attributename`.

7.4 Attribute Names

- The **result set/table** consists of the attributes given in the SELECT clause.
- However, result attributes can be **renamed**:
  - Remember the renaming operator `AS` from relational algebra...
  - SQL uses the `AS` keyword for renaming.
  - The new names can also be used in the WHERE clause.
- Example:
  - `SELECT persons.personName AS name FROM persons, ... WHERE name = 'Smith' AND ...
    • PersonName is now called name in the result schema.
    • name = 'Smith' means persons.personName = 'Smith'.

7.4 Basic Select

- **Query block**:
  - `SELECT expression AS column FROM table name AS column` for renaming
  - `DISTINCT` filters out duplicate results.
  - `WHERE search condition` filters tuples matching a given search condition.
  - `GROUP BY column name` groups tuples by a column name.
  - `HAVING search condition` filters grouped results.

7.4 Conditions

- Usually, SQL queries return exactly those tuples matching a given **search condition**:
  - Indicated by the WHERE keyword.
  - The condition is a **logical expression** which can be applied to each row and may have one of three values: **TRUE, FALSE, and NULL**.
  - Again, NULL might mean “unknown,” “does not apply,” “is not matter,”...
7.4 Conditions

- Search conditions are conjunctions of predicates
  - Each predicate evaluates to TRUE, FALSE, or NULL

```
search condition
    AND
    OR
    NOT

predicate
    (search condition)
```

7.4 Conditions

- Why TRUE, FALSE, and NULL?
  - SQL uses so-called ternary (three-valued) logic
  - When a predicate cannot be evaluated because it
    contains some NULL value, the result will be NULL
  - Example: `powerStrength > 10` evaluates to NULL
    if `powerStrength` is NULL
  - `NULL = NULL` also evaluates to NULL

- Handling of NULL by the operators AND and OR:

```
AND   | TRUE   | FALSE | NULL
-----|--------|-------|------
TRUE  | TRUE   | FALSE | NULL
FALSE | TRUE   | FALSE | NULL
NULL  | TRUE   | FALSE | NULL
```

7.4 Conditions

- Predicates usually contain expressions
  - Column names or constants
  - Additionally, SQL provides some special expressions
    - Functions
    - CASE expressions
    - CAST expressions
    - Scalar subqueries

7.4 Conditions

- Expressions can be combined using expression operators
  - Arithmetic operators:
    +, -, *, / with the usual semantics
    • age + 2
    • price * quantity
  - String concatenation `||` (also written as CONCAT)
    Combines two strings into one
    • `firstName ' ' || lastname || ' (aka ' || alias || ')'`
    • ‘Hello’ CONCAT ‘World’ → ‘Hello World’
  - Parenthesis:
    Used to modify the evaluation order
    • `(price + 10) * 20`

7.4 Conditions

- Simple comparisons:
  - Valid comparison operators are
    • =, <, <=, >, >=
    • <> (meaning: not equal)
  - Data types of expressions need to be compatible (if not, CAST has to be used)
  - Character values are usually compared lexicographically (while ignoring case)
  - Examples:
    • `powerStrength > 10`
    • `name = 'Magneto'`
    • `Magneto < 'Professor X'`
7.4 Conditions

• BETWEEN predicate:
  - \( X \) BETWEEN \( Y \) AND \( Z \) is a shortcut for \( Y \leq X \) AND \( X \leq Z \)
  - Note that you cannot reverse the order of \( Z \) and \( Y \)
    • \( X \) BETWEEN \( Y \) AND \( Z \) is different from \( X \) BETWEEN \( Z \) AND \( Y \)
    • The expression can never be true if \( Y > Z \)
  - Examples:
    • year BETWEEN 2000 AND 2008
    • score BETWEEN targetScore - 10 AND targetScore + 10

• IS NULL predicate:
  - The only way to check if a value is NULL or not
  - Returns either TRUE or FALSE
  - Examples:
    • realName IS NOT NULL
    • powerStrength IS NULL
    • NULL IS NULL

• LIKE predicate:
  - The predicate is for matching character strings to patterns
  - Match expression must be a character string
  - Pattern expression is a (usually constant) string
    • May not contain column names
  - Escape expression is just a single character
    • During evaluation, the match expression is compared to the pattern expression with following additions
      • \% represents any number of arbitrary characters
      • \_ represents the special semantic of a single character
    • The escape character prevents the special semantics of \% and \_
    • Most modern database nowadays also support more powerful regular expressions (introduced in SQL-99)
  - Examples:
    • address LIKE '%City%'
      • 'Manhattan' → FALSE
      • 'Gothenburg City Prison' → TRUE
    • name LIKE 'M% t.'
      • 'Magneto' → TRUE
      • 'Matt' → TRUE
      • 'Mr.' → FALSE
    • status LIKE '/.\% ESCAPE/'
      • '1_inPrison' → TRUE
      • '1-inPrison' → FALSE
      • '99/99/99' → FALSE

• IN predicate:
  - Evaluates to true if the value of the test expression is within a given set of values
  - Particularly useful when used with a subquery (later)
  - Examples:
    • name IN ('Magneto', 'Batman', 'Dr. Doom')
    • name IN (SELECT title FROM movies)
      • Those people having a film named after them...

• EXISTS predicate:
  - Evaluates to TRUE if a given subquery returns at least one result row
  - Always returns either TRUE or FALSE
  - Examples:
    • EXISTS (SELECT * FROM heroes)
    • EXISTS may also be used to express semi-joins
7.4 Conditions

• SOME/ANY and ALL:
  – COMPARES AN expression TO each value PROVIDED BY a subquery
  – TRUE if
    • SOME/ANY: At least one comparison returns TRUE
    • ALL: All comparisons return TRUE
  – Examples:
    • result < ALL (SELECT result FROM results)
      – TRUE if the current result is the smallest one
    • result < SOME (SELECT result FROM results)
      – TRUE if the current result is not the largest one

7.5 Joins

• Also, SQL can do joins of multiple tables
• Traditionally, this is performed by simply stating multiple tables in the FROM clause
  – Result contains all possible combinations of all rows of all tables such that the search condition holds
  – If there is no WHERE clause, it’s a Cartesian product

4.0 102 100
s.matNr

68
70

List students and their exam results
– π lastName, crsNr, result
FROM students AS s JOIN exams AS e ON s.matNr = e.matNr

9876
100
3.7

We lost Lex Luther and Charles Xavier because they didn’t take any exams! Also information on student 9876 disappears...
7.5 Joins

Right outer join:
- π lastName, crsNr, result
  students ⎨ exams
- SELECT lastName, crsNr, result
  FROM students ⎩ exams

Full outer join:
- π lastName, crsNr, result
  students ⎨ exams
- SELECT lastName, crsNr, result
  FROM students ⎩ exams

7.6 Set Operators

- SQL also supports the common set operators
  - Set union ∪: UNION
  - Set intersection ∩: INTERSECT
  - Set difference \: EXCEPT
- By default, set operators eliminate duplicates unless the ALL modifier is used
- Sets need to be union-compatible to use set operators
  - Row definition must match (data types)

Example:
( SELECT course, result
  FROM exams
  WHERE course = 100
  UNION
  SELECT course, result
  FROM exams
  WHERE result IS NULL

7.7 Column Function

- Column functions are used to perform statistical computations
  - Similar to aggregate function in relational algebra
  - Column functions are expressions
  - They compute a scalar value for a set of values
- Examples:
  - Compute the average score over all exams
  - Count the number of exams each student
  - Retrieve the best student
  - ...
7.7 Column Function

• **Examples:**
  - `SELECT COUNT(*) FROM heroes`
    - Returns the number of rows of the heroes table
  - `SELECT COUNT(name), COUNT(DISTINCT name) FROM heroes`
    - Returns the number of rows in the heroes table for that name is not null and the number of non-null unique names
  - `SELECT MIN(strength), MAX(strength), AVG(strength) FROM powers`
    - Returns the minimal, maximal, and average power strength in the powers table

7.7 Grouping

• **Examples:**
  - `SELECT course, AVG(result), COUNT(*)
    FROM exams GROUP BY course`
    - For each course, list the average result, the number of results, and the number of non-null results
  - `SELECT COUNT(*) FROM heroes`
    - Returns the number of rows of the heroes table
  - `SELECT COUNT(name), COUNT(result)
    FROM exams GROUP BY course`
    - The where clause is evaluated before the groups are formed!

• **Examples:**
  - `SELECT course, AVG(result), COUNT(*)
    FROM exams GROUP BY course
    HAVING count(*) > 1`
    - Select course, average result, and count(*) where count(*) is greater than 1
  - `SELECT course, AVG(result) AS avgResult
    FROM exams GROUP BY course
    HAVING avgResult > 1`
    - Select course and average result where average result is greater than 1

• **Examples:**
  - `SELECT course, AVG(result), COUNT(*)
    FROM exams GROUP BY course
    HAVING COUNT(result) > 1`
    - Select course, average result, and count(*) where count(*) is greater than 1
  - `SELECT course, AVG(result) AS avgResult
    FROM exams GROUP BY course
    HAVING avgResult > 1`
    - Select course and average result where average result is greater than 1

7.7 Grouping

• **Similar to aggregation in relation algebra, SQL supports grouping**
  - `GROUP BY <column names>`
    - Creates a group for each combination of distinct values within the provided columns
  - `A query containing GROUP BY can access non-group-by-attributes only by column functions`
7.8 Ordering

- As last step in the processing pipeline, (unordered) result sets may be converted into lists
  - Impose an order on the rows
  - This concludes the SELECT statement
  - ORDER BY keyword
- Please note: Ordering completely breaks with set calculus/algebra
  - Result after ordering is a list, not a (multi-)set!

7.8 Evaluation Order of SQL

- SQL queries are evaluated in this sequence:
  1. FROM <table list>
  2. [WHERE <condition>]
  3. [GROUP BY <attribute list>]
  4. [HAVING <condition>]
  5. SELECT <attribute list>
  6. [UNION/INTERSECT/EXCEPT <query>]
  7. [ORDER BY <attribute list>]

7.9 Subqueries

- In SQL, you may embed a query block within a query block (so called subquery, or nested query)
  - Subqueries are written in parenthesis
  - Literal subqueries can be used as expressions
    - Return just one single value
  - Subqueries may create the test set for the IN-condition
  - Subqueries may create the test set for the EXISTS-condition
  - Each subquery within the table list creates a temporary source table
    - Called inline view
- Subqueries may either be correlated or uncorrelated
  - If the WHERE clause of the inner query uses an attributes within a table declared in the outer query, the two queries are correlated
    - The inner query needs to be re-evaluated for every tuple in the outer query
    - This is rather inefficient, so avoid correlated subqueries whenever possible!
  - Otherwise, the queries are uncorrelated
    - The inner queries needs to be evaluated just once

7.8 Ordering

- ORDER BY may order ascending or descending
  - Default: ascending
  - Ordering on multiple columns possible
  - Columns used for ordering are references by their name
- Example
  - SELECT * FROM results
    ORDER BY student, csnNo DESC
  - Returns all results ordered by student id (ascending)
  - If student ids are identical, we sort in descending order by course number
7.9 Subqueries

- **Expressions:**
  - `SELECT hero.*` FROM `hero`, `hasPower` `p`
    `WHERE hero.id = p.hero_id`
    `AND powerStrength =`
    `(SELECT MAX(powerStrength) FROM hasPower)`
  - Select all those heroes having powers with maximal strength
  - Uncorrelated! Subquery is evaluated only once

- **IN-condition:**
  - `SELECT * FROM hero WHERE id IN (SELECT hero_id FROM hasAlias WHERE aliasName LIKE 'Super%')`
  - Select all those heroes having an alias starting with “Super”
  - Uncorrelated

7.10 Writing Good SQL Code

1. Write SQL keywords in uppercase, names in lowercase!

   **BAD**
   ```sql```
   SELECT MOVIE_TITLE
   FROM MOVIES
   WHERE MOVIE_YEAR = 2009
   ```

   **GOOD**
   ```sql```
   SELECT movie_title
   FROM movies
   WHERE movie_year = 2009
   ```

2. Use proper qualification!

   **BAD**
   ```sql```
   SET SCHEMA imdbraw
   SELECT imdbraw.movies.movie_title, imdbraw.movies.movie_year
   FROM imdbraw.movies
   WHERE imdbraw.movies.movie_year = 2009
   ```

   **GOOD**
   ```sql```
   SELECT movie_title, movie_year
   FROM movies
   WHERE movie_year = 2009
   ```
7.10 Writing Good SQL Code

3. Use aliases to keep your code short and the result clear!

**BAD**

```sql
SELECT movie_title, movie_year
FROM movies, genres
WHERE movies.movie_id = genres.movie_id
AND g.genre = 'Action'
```

**GOOD**

```sql
SELECT movie_title, movie_year
FROM movies m, genres g
WHERE m.movie_id = g.movie_id
AND g.genre = 'Action'
```

5. Use proper indentation!

**BAD**

```sql
SELECT movie_title, movie_year
FROM movies m, genres g, actors a
WHERE m.movie_id = a.movie_id
AND g.genre = 'Action'
AND a.person_name LIKE '%Schwarzenegger'
```

**GOOD**

```sql
SELECT movie_title, movie_year
FROM movies m
JOIN genres g ON m.movie_id = g.movie_id
JOIN actors a ON a.movie_id = m.movie_id
WHERE g.genre = 'Action'
AND a.person_name LIKE '%Schwarzenegger'
```

7. Avoid subqueries (use joins if possible)!

**BAD**

```sql
WITH recent_actors (person_id) AS
(SELECT DISTINCT person_id
FROM directors d
WHERE d.person_id IN (SELECT * FROM recent_actors))
```

**GOOD**

```sql
SELECT DISTINCT person_name
FROM directors d
WHERE d.person_id IN (SELECT * FROM recent_actors)
```

4. Separate joins from conditions!

**BAD**

```sql
SELECT movie_title, movie_year
FROM movies m, genres g, actors a
WHERE m.movie_id = a.movie_id
AND g.genre = 'Action'
AND a.person_name LIKE '%Schwarzenegger'
```

**GOOD**

```sql
SELECT movie_title, movie_year
FROM movies m
JOIN genres g ON m.movie_id = g.movie_id
JOIN actors a ON a.movie_id = m.movie_id
WHERE g.genre = 'Action'
AND a.person_name LIKE '%Schwarzenegger'
```

6. Extract uncorrelated subqueries!

**BAD**

```sql
SELECT DISTINCT person_id
FROM directors d
WHERE d.person_id IN (SELECT * FROM recent_actors)
```

**GOOD**

```sql
SELECT DISTINCT person_name
FROM directors d
WHERE d.person_id IN (SELECT * FROM recent_actors)
```