Relational Database Systems I

Christoph Lofi
Simon Barthel

Institut für Informationssysteme
Technische Universität Braunschweig
www.ifis.cs.tu-bs.de
8.0 Overview of SQL

• SQL
  – Queries
    • SELECT
  – Data Manipulation Language (next lecture)
    • INSERT
    • UPDATE
    • DELETE
  – Data Definition Language (next lecture)
    • CREATE TABLE
    • ALTER TABLE
    • DROP TABLE
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 taught in this lecture)
8.0 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**
8.0 Overview of SQL

Environment

*BigCompany database server*

Catalog

*human_resources*

- Schema *people*
  - Table *staff*
  - Table *has_office*

- Schema *taxes*

- Schema *training*

Catalog

*production*

- Schema *products*
  - ...

- Schema *testing*
  - ...

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**
- e.g. schemaname.objectname

However, when working only with objects from a single schema, using **unqualified names** would be nice
- e.g. objectname

One schema always is defined to be the **default schema**
- SQL implicitly treats objectname as
defaultschema.objectname
- the default schema can be set with **SET SCHEMA**:
  - SET SCHEMA schemaname
- initially, after login the default schema corresponds to the current user name
  - remember to change the default schema accordingly!
• **Basic SQL Queries**
  - `SELECT`, `FROM`, `WHERE`

• **Advanced SQL Queries**
  - Joins
  - Set operations
  - Aggregation and `GROUP BY`
  - `ORDER BY`
  - Subqueries

• **Writing Good SQL Code**
SQL queries are statements that **retrieve information** from a DBMS

- simplest case: from a single table, return all rows matching some given condition
- SQL queries may return **multi-sets (bags)** of rows
  - duplicates are allowed by default (but can be eliminated on request)
  - even just a single value is a result set (one row with one column)
  - This is different in Relational Algebra, TRC, DRC, etc.
  - however, often it’s just called a result set…
8.1 SQL Queries

- 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 is evaluated on every tuple
  - if no condition is provided, it is implicitly replaced by **TRUE**
8.1 Attribute Names

• The `SELECT` keyword is often confused with `selection` from relational algebra
  – actually `SELECT` corresponds to `projection`

• Example:
  – Table `student` with attributes `id`, `fname`, `lname`
    ```sql
    SELECT id, lname
    FROM student
    WHERE id >= 100
    ```
  – TRC:
    ```plaintext
    \{t.id, t.lname | student(t) \land t.id \geq 100\}
    ```
  – DRC:
    ```plaintext
    \{id, ln | \exists fn(student(id, fn, ln) \land id \geq 100)\}
    ```
  Rel. Algebra:
  ```plaintext
  \pi_{id,lname} \sigma_{id \geq 100}\text{student}
  ```
8.1 Attribute Names

• To return all attributes under consideration, the wildcard * may be used

• Examples
  – SELECT * FROM <list of tables>
    Return all attributes of the tables in the FROM clause.
  – SELECT movie.* FROM movie, person WHERE...
    Return all attributes of the movies table.
8.1 Enforcing Sets in SQL

• SQL can perform 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
8.1 Attribute Names

- Attribute names are qualified or unqualified
  - **unqualified**: just the attribute name
    - only possible, if attribute name is *unique* among the tables given in the **FROM** clause
  - **qualified**: `tablename.attributename`
    - 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:
      `schemaname.tablename.attributename`
8.1 Attribute Names

• The attributes in the result set are defined in a SELECT clause

• However, result attributes can be renamed
  – remember the renaming operator \( \rho \) from relational algebra…
  – SQL uses the AS keyword for renaming
  – the new names can also be used in the WHERE clause

• Example
  – SELECT person.person_name AS name
    FROM person WHERE name = 'Smith'
  • person_name is now called name in the result table
8.1 Table Names

- **Table names** can be referenced in the same way as attribute names (qualified or unqualified)
- However, **renaming** works slightly different
  - the result table of an SQL query has no name
  - but tables can be given **alias names** to simplify queries (also called **tuple variables** or just **aliases**)
  - indicated by the **AS** keyword

- Example
  ```sql
  SELECT title, genre
  FROM movie AS m, genre AS g
  WHERE m.id = g.id
  ```

- The **AS** keyword is optional: `FROM movie m, genre g`

- Compare to TRC:
  ```
  \{ m.title, g.genre | movie(m) \land genre(g) \land m.id = g.id \}
  ```
8.1 Basic Select

query block

SELECT

ALL
DISTINCT

expression

FROM

WHERE
HAVING

GROUP BY

search condition

column name

attribute names

query block

SELECT

expression

FROM

WHERE
HAVING

GROUP BY

table name

alias name

attribute names
• One of the basic building blocks of SQL queries is the **expression**
  
  – Expressions represent a literal, i.e. a number, a string, or a date
  
  – **column names** or **constants**
  
  – additionally, SQL provides some **special expressions**
    
    • functions
    • **CASE** expressions
    • **CAST** expressions
    • scalar subqueries
8.1 Expressions

• Expressions can be combined using **expression operators**
  
  – **arithmetic operators:** 
    
    $+, -, *, \text{ and } /$ with the usual semantics
    
    • `age + 2`
    
    • `price * quantity`
  
  – **string concatenation** `||`: (also written as `CONCAT`) combines two strings into one
    
    • `first_name || ' ' || lastname || ' (aka ' || alias || ')'
    
    • `'Hello' CONCAT ' World'` $\rightarrow$ `'Hello World'`
  
  – **parenthesis:**
    
    used to modify the evaluation order
    
    • `(price + 10) * 20`
• 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 missing,* ...
8.1 Conditions

• Search conditions are conjunctions of predicates
  — each predicate evaluates to TRUE, FALSE, or NULL
8.1 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: `power_strength > 10` evaluates to `NULL` iff `power_strength` is `NULL`
  - `NULL = NULL` also evaluates to `NULL`

- **Handling of `NULL` by the operators `AND` and `OR`:**

<table>
<thead>
<tr>
<th>AND</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>TRUE</th>
<th>FALSE</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th>TRUE</th>
<th>FALSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
8.1 Conditions

- **Predicate**: expression → comparison → expression

- **NOT**: expression

- **BETWEEN**: expression → AND → expression

- **IS**: expression → NOT → NULL

- **LIKE**: expression → NOT → ESCAPE → expression

- **IN**: expression → NOT → (expression)

- **EXISTS**: (query)

- **NOT IN**: expression

- **NOT BETWEEN**: expression

- **NOT IS**: expression

- **NOT LIKE**: expression

- **NOT IN**: expression

- **SOME**: (query)

- **ANY**: (query)

- **ALL**: (query)
• **Simple comparisons**
  - valid comparison operators are
    • =, <, <=, >=, and >
    • <> (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'`
8.1 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** target_score-10 **AND** target_score+10
8.1 Conditions

• **IS NULL predicate**
  – the only way to check if a value is **NULL** or not
    • recall: NULL = NULL returns NULL
  – returns either **TRUE** or **FALSE**
  – examples
    • `real_name IS NOT NULL`
    • `power_strength IS NULL`
    • `NULL IS NULL`
8.1 Conditions

- **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
    - _ in the pattern expression represents any single character
    - % represents any number of arbitrary characters
    - the escape character prevents the special semantics of _ and %
  - most modern database nowadays also support more powerful **regular expressions** (introduced in SQL-99)
8.1 Conditions

• **Examples**
  
  - **address** `LIKE '%City%'`
    - 'Manhattan' → **FALSE**
    - 'Gotham City Prison' → **TRUE**
  
  - **name** `LIKE 'M%_t_'`
    - 'Magneto' → **TRUE**
    - 'Matt' → **TRUE**
    - 'Mtt' → **FALSE**
  
  - **status** `LIKE '_/_%' ESCAPE '/'`
    - '1_inPrison' → **TRUE**
    - '1inPrison' → **FALSE**
    - '%_%' → **TRUE**
    - '%%%%' → **FALSE**
8.1 Conditions

- **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 movie)`
      - Those people having a film named after them…
8.1 Conditions

• **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 hero)
    • *Do we have any hero stored in our database?*
  - **EXISTS** may also be used to express semi-joins
8.1 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**
      - **SOME** and **ANY** are synonyms
    - **ALL**: All comparisons return **TRUE**
  - **examples**
    - \( \text{result} \leq \text{ALL}(\text{SELECT result FROM results}) \)  
      - **TRUE** if the current result is the smallest one
    - \( \text{result} < \text{SOME}(\text{SELECT result FROM results}) \)  
      - **TRUE** if the current result is not the largest one
• Simple SQL Queries
  – `SELECT, FROM, WHERE`

• Advanced SQL Queries
  – Joins
  – Set operations
  – Aggregation and `GROUP BY`
  – `ORDER BY`
  – Subqueries

• Writing Good SQL Code
• Also, SQL can do **joins of multiple tables**

• Traditionally, this is performed by simply stating **multiple tables** in the **FROM** clause
  
  – This directly stems from the tuple calculus
  
  – 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
8.2 Joins

• Example
  – `SELECT * FROM hero, has_alias`
    • TRC: \{h, ha | hero(h) \land has_alias(ha)\}
    • Rel. Algebra: hero \times has_alias
  
  – `SELECT * FROM hero h, has_alias ha`
    WHERE h.id = ha.hero_id`
    • TRC:
      – \{h, ha | hero(h) \land has_alias(ha) \land h.id = ha.hero_id\}
    • Rel. Algebra (naïve):
      – \(\sigma_{id=hero_id}\) hero \times has_alias
    • Rel. Algebra (optimized):
      – hero \bowtie_{id=hero_id} has_alias

• Besides this common implicit notation of joins, SQL also supports explicit joins
  – Burrowed from Relational Algebra
8.2 Joins

- Explicit joins are specified in the **FROM** clause
  
  ```
  SELECT * FROM table1
  JOIN table2 ON <join condition>
  WHERE <some other condition>
  ```

- Often, attributes in joined tables have the same names, so qualified attributes are needed.

- **INNER JOIN** and **JOIN** are equivalent

  - `INNER JOIN`
  - `JOIN` (equivalent to `INNER JOIN`)
  - `LEFT JOIN`
  - `RIGHT JOIN`
  - `FULL JOIN`

- **Join operators**

  - **INNER JOIN**
  - **LEFT JOIN**
  - **RIGHT JOIN**
  - **FULL JOIN**

- *explicit joins improve readability of your SQL code!*
8.2 Joins

**Inner join:** List students and their exam results.

- $\pi_{\text{lastname, course, result}} (\text{Student } \bowtie_{\text{mat_no}=\text{student}} \text{ exam})$

- `SELECT lastname, course, result FROM student AS s JOIN exam AS e ON s.mat_no = e.student`

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

**Left outer join:** List students and their exam results

- $\pi_{\text{lastname, course, result}} (\text{Student} \bowtie_{\text{mat_no}=\text{student}} \text{exam})$

- SELECT 
  lastname, course, result FROM student AS s
  LEFT JOIN exam AS e ON s.mat_no = e.student

### Student

<table>
<thead>
<tr>
<th>mat_no</th>
<th>firstname</th>
<th>lastname</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
<td>Kent</td>
<td>m</td>
</tr>
<tr>
<td>2832</td>
<td>Louise</td>
<td>Lane</td>
<td>f</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
<td>Luther</td>
<td>m</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
<td>Xavier</td>
<td>m</td>
</tr>
</tbody>
</table>

### Exam

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>2.0</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
</tr>
<tr>
<td>1005</td>
<td>100</td>
<td>1.3</td>
</tr>
</tbody>
</table>

$\pi_{\text{lastname, course, result}} (\text{Student} \bowtie_{\text{mat_nr}=\text{student}} \text{exam})$
### 8.2 Joins

#### Right outer join:

- \[ \pi \text{lastname, course, result} (\text{Student } \bowtie_{\text{mat_no}=\text{student}} \text{ exam}) \]

- \[ \text{SELECT lastname, course, result FROM student s RIGHT JOIN exam e ON s.mat_no = e.student} \]

<table>
<thead>
<tr>
<th>mat_no</th>
<th>firstname</th>
<th>lastname</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
<td>Kent</td>
<td>m</td>
</tr>
<tr>
<td>2832</td>
<td>Louise</td>
<td>Lane</td>
<td>f</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
<td>Luther</td>
<td>m</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
<td>Xavier</td>
<td>m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>2.0</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
</tr>
</tbody>
</table>

\[ \pi \text{lastname, course, result} (\text{Student } \bowtie_{\text{mat_no}=\text{student}} \text{ exam}) \]

<table>
<thead>
<tr>
<th>lastname</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kent</td>
<td>100</td>
<td>1.3</td>
</tr>
<tr>
<td>Kent</td>
<td>101</td>
<td>4.0</td>
</tr>
<tr>
<td>Lane</td>
<td>102</td>
<td>2.0</td>
</tr>
<tr>
<td><strong>NULL</strong></td>
<td>100</td>
<td>3.7</td>
</tr>
</tbody>
</table>
8.2 Joins

Full outer join:

- \( \pi \) \text{lastname}, \text{course}, \text{result} (\text{Student} \bowtie_{\text{mat_no} = \text{student}} \text{exam})

- \text{SELECT} \ \text{lastname}, \ \text{course}, \ \text{result} \ \text{FROM} \ \text{student} \ s \ \text{FULL JOIN} \ \text{exam} \ e \ \text{ON} \ s.\text{mat_no} = e.\text{student}

Student

<table>
<thead>
<tr>
<th>matNr</th>
<th>firstname</th>
<th>lastname</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1005</td>
<td>Clark</td>
<td>Kent</td>
<td>m</td>
</tr>
<tr>
<td>2832</td>
<td>Louise</td>
<td>Lane</td>
<td>f</td>
</tr>
<tr>
<td>4512</td>
<td>Lex</td>
<td>Luther</td>
<td>m</td>
</tr>
<tr>
<td>5119</td>
<td>Charles</td>
<td>Xavier</td>
<td>m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lastname</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kent</td>
<td>100</td>
<td>1.3</td>
</tr>
<tr>
<td>Kent</td>
<td>101</td>
<td>4.0</td>
</tr>
<tr>
<td>Lane</td>
<td>102</td>
<td>2.0</td>
</tr>
<tr>
<td>Luther</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>Xavier</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

\( \pi \) \text{lastname}, \text{course}, \text{result} (\text{Student} \bowtie_{\text{mat_no} = \text{student}} \text{exam})
8.2 Set Operators

• SQL also supports the common set operators
  – set union \( \cup \): \textsc{union}
  – set intersection \( \cap \): \textsc{intersect}
  – set difference \( \setminus \): \textsc{except}

• By default, set operators \textbf{eliminate duplicates} unless the \textsc{all} modifier is used

• Sets need to be \textbf{union-compatible} to use set operators
  – row definition must match (data types)
8.2 Set Operators

• Example

- \((\text{SELECT course, result FROM exam}
  \text{ WHERE course = 100})
  \text{ EXCEPT}
  (\text{SELECT course, result FROM exam}
  \text{ WHERE result IS NULL})\)

\text{UNION VALUES (100, 2.3), (100, 1.7)}

exam

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>2.0</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
</tr>
<tr>
<td>1005</td>
<td>100</td>
<td>NULL</td>
</tr>
<tr>
<td>1005</td>
<td>100</td>
<td>NULL</td>
</tr>
<tr>
<td>6676</td>
<td>102</td>
<td>4.3</td>
</tr>
<tr>
<td>3412</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>100</td>
<td>2.3</td>
</tr>
<tr>
<td>100</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Relational Database Systems 1 – Christoph Lofi – Institut für Informationssysteme – TU Braunschweig
• **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 has taken
  – retrieve the *best* student
  – …
8.2 Column Function

- **Column functions in the SQL standard**
  - **MIN, MAX, AVG, COUNT, SUM:**
    - each of these are applied to some other expression
      - NULL values are ignored
      - function columns in result set just get their column number as name
  - if **DISTINCT** is specified, duplicates are eliminated in advance
    - by default, duplicates are not eliminated (**ALL**)
  - **COUNT** may also be applied to *****
    - simply counts the number of rows
  - typically, there are many more column functions available in your RDBMS (e.g. in DB2: **CORRELATION, STDDEV, VARIANCE, ...**)
8.2 Column Function

• Examples

  – SELECT COUNT(*) FROM hero
    • Returns the number of rows of the heroes table.

  – SELECT COUNT(name), COUNT(DISTINCT name) FROM hero
    • Returns the number of rows in the hero table for that name is not null and the number of non-null unique names.

  – SELECT MIN(strength), MAX(strength), AVG(strength) FROM power
    • Returns the minimal, maximal, and average power strength in the power table.
8.2 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
8.2 Grouping

Examples

- \( \text{SELECT} \text{ course, AVG(result), COUNT(*), COUNT(result)} \)
  \( \text{FROM} \) exam
  \( \text{GROUP BY course} \)

  - For each course, list the average result, the number of results, and the number of non-null results.
  
  - Rel. Alg.: \( \text{course} \bowtie \text{avg(result), count(result)} \) exam
    - \( \text{count(*)} \) is not defined in Relational Algebra

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>2.0</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
</tr>
<tr>
<td>1005</td>
<td>100</td>
<td>NULL</td>
</tr>
<tr>
<td>6676</td>
<td>102</td>
<td>4.3</td>
</tr>
<tr>
<td>3412</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>course</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3.7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>4.0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>3.15</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
8.2 Grouping

Examples

- \textbf{SELECT} course, \textbf{AVG(result)}, \textbf{COUNT(*)}
  \textbf{FROM} exam
  \textbf{WHERE} course IS NOT NULL
  \textbf{GROUP BY} course

  • the where clause is evaluated before the groups are formed!

<table>
<thead>
<tr>
<th>student</th>
<th>course</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9876</td>
<td>100</td>
<td>3.7</td>
</tr>
<tr>
<td>2832</td>
<td>102</td>
<td>2.0</td>
</tr>
<tr>
<td>1005</td>
<td>101</td>
<td>4.0</td>
</tr>
<tr>
<td>1005</td>
<td>100</td>
<td>NULL</td>
</tr>
<tr>
<td>6676</td>
<td>102</td>
<td>4.3</td>
</tr>
<tr>
<td>3412</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

\begin{tabular}{|c|c|c|}
\hline
\textbf{course} & 2 & 3 \\
\hline
100 & 3.7 & 2 \\
101 & 4.0 & 1 \\
102 & 3.15 & 2 \\
\hline
\end{tabular}
8.2 Grouping

• Additionally, there may be restrictions on the groups themselves
  
  – **HAVING** <condition>
  
  – the condition may involve group properties and **column functions**
  
  – only those groups are created that fulfill the **HAVING** condition
  
  – a query may have a **WHERE** and a **HAVING** clause
  
  – also, it is possible to have **HAVING** without **GROUP BY**
    
    • then, the whole table is treated as a single group – which is rarely useful
8.2 Grouping

• Examples

- \[\text{SELECT}\ \text{course, AVG(result), COUNT(*)}\]
  \[\text{FROM exam}\]
  \[\text{WHERE course <> 100}\]
  \[\text{GROUP BY course}\]
  \[\text{HAVING COUNT(*) > 1}\]
8.2 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!

SELECT statement

query

ORDER BY

column name

ASC

DESC
8.2 Ordering

• ORDER BY may order ascending or descending
  – default: ascending (ASC)
• Ordering on multiple columns possible
• Columns used for ordering are referenced by their name
• Example
  – SELECT * FROM exam
    ORDER BY student, course DESC
  – returns all exam results ordered by student id (ascending)
  – if student ids are identical, we sort in descending order by course number
8.2 Ordering

• When working with result lists, often only the first $k$ rows are of interest

• How can we limit the number of result rows?
  – since SQL:2008, the SQL standard offers the `FETCH FIRST` clause (supported e.g. in DB2)

• Example

  ```sql
  SELECT name, salary
  FROM salaries
  ORDER BY salary
  FETCH FIRST 10 ROWS ONLY
  ```

  – `FETCH FIRST` can also be used without `ORDER BY`

  • get a quick impression of the result set
8.2 Evaluation Order of SQL

- SQL queries are evaluated in this order

  5. `SELECT <attribute list>`
  1. `FROM <table list>`
  2. `[WHERE <condition>]`
  3. `[GROUP BY <attribute list>]`
  4. `[HAVING <condition>]`
  6. `[UNION/INTERSECT/EXCEPT <query>]`
  7. `[ORDER BY <attribute list>]`
8.2 Subqueries

• In SQL, you may embed a query block within a query block (so called subquery, or nested query)
  – subqueries are written in parenthesis
  – scalar subqueries can be used as expressions
    • if query returns only one row with one column
  – subqueries may be used for IN or EXISTS conditions
  – 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 attribute 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 query needs to be evaluated just once
8.2 Subqueries

- **has_alias**
  - **hero**
  - **alias_name**

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

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

- **power**
  - **id**
  - **name**
  - **description**
8.2 Subqueries

• Expressions

  - \textbf{SELECT} hero.* \textbf{FROM} hero, has_power p \\
    \textbf{WHERE} hero.id = p.hero \\
    \textbf{AND} power_strength = ( \\
    \hspace{1em} \textbf{SELECT} \text{MAX}(\text{power_strength}) \\
    \hspace{1em} \textbf{FROM} has_power \\
    )

  • \textit{Select all those heroes having powers with maximal strength.}

• IN-condition

  - \textbf{SELECT} * \textbf{FROM} hero \textbf{WHERE} id \textbf{IN} ( \\
    \hspace{1em} \textbf{SELECT} hero \textbf{FROM} has_alias \\
    \hspace{1em} \textbf{WHERE} alias_name \textbf{LIKE} 'Super%' \\
    )

  • \textit{Select all those heroes having an alias starting with Super.}
8.2 Subqueries

• **EXISTS-condition:**
  - \( \text{SELECT * FROM hero h} \)
  - \( \text{WHERE EXISTS (} \)
  - \( \text{SELECT * FROM has_alias a WHERE h.id = a.hero} \)
  - \( \text{)} \)
  - Select heroes having at least one alias.
  - this pattern is normally used to express a *semi join*
  - if the DBMS would not optimize this into a semi join, the subquery has to be evaluated *for each tuple* (uncorrelated subquery!)

• **Inline view**
  - \( \text{SELECT h.real_name, a.alias_name} \)
  - \( \text{FROM has_alias a, (} \)
  - \( \text{SELECT * FROM hero WHERE real_name LIKE 'A%'} \)
  - \( \text{)} \)
  - \( \text{WHERE h.id < 100 AND a.hero = h.id} \)
  - Get real_name-alias pairs for all heroes with a real name staring with A and an id smaller than 100.
• **WITH-clause (temporary tables):**

```sql
WITH hero_num_powers AS (
    SELECT hero AS h_id, COUNT(*) AS num_pow
)

SELECT * FROM hero h
JOIN hero_num_powers hnp ON h.id = hnp.h_id
WHERE hnp.num_pow = (SELECT MAX(num_pow) FROM hero_num_powers)
```

- **Select heroes having most powers**
- Extremely useful if the expression in the **WITH**-clause is used multiple times
- Also useful for readability
• Simple SQL Queries
  – SELECT, FROM, WHERE

• Advanced SQL Queries
  – Joins
  – Set operations
  – Aggregation and GROUP BY
  – ORDER BY
  – Subqueries

• Writing Good SQL Code
• What is good SQL code?
  – easy to read
  – easy to write
  – easy to understand!

• There is no official SQL style guide, but here are some general hints
I. Write SQL keywords in uppercase, names in lowercase!

**BAD**

```
SELECT MOVIE_TITLE
FROM MOVIE
WHERE MOVIE_YEAR = 2009;
```

**GOOD**

```
SELECT movie_title
FROM movie
WHERE movie_year = 2009;
```
8.3 Writing Good SQL Code

2. Use proper qualification!

**BAD**

```sql
SELECT imdbraw.movie.movie_title,
imdbraw.movie.movie.movie_year
FROM imdbraw.movie
WHERE imdbraw.movie.movie.movie_year = 2009;
```

**GOOD**

```sql
SET SCHEMA imdbraw;
SELECT movie_title, movie_year
FROM movie
WHERE movie_year = 2009;
```
3. Use aliases to keep your code short and the result clear!

BAD

```sql
SELECT movie_title, movie_year
FROM movie, genre
WHERE movie.movie_id = genre.movie_id
    AND genre.genre = 'Action';
```

GOOD

```sql
SELECT movie_title, movie_year
FROM movie m, genre g
WHERE m.movie_id = g.movie_id
    AND g.genre = 'Action';
```
4. Use joins to join!

**GOOD**

```
SELECT movie_title title, genre g
FROM movie m
    JOIN genre g ON m.movie_id = g.movie_id
WHERE g.genre='Action'
```

**BAD**

```
SELECT movie_title title, genre g
FROM movie m
    JOIN genre g ON g.genre='Action'
WHERE m.movie_id = g.movie_id
```
5. Separate joins from conditions!

BAD

```sql
SELECT movie_title title, movie_year year
FROM movie m, genre g, actor a
WHERE m.movie_id = g.movie_id
    AND g.genre = 'Action'
    AND m.movie_id = a.movie_id
    AND a.person_name LIKE '%Schwarzenegger%';
```

GOOD

```sql
SELECT movie_title title, movie_year year
FROM movie m
    JOIN genre g ON m.movie_id = g.movie_id
    JOIN actor a ON g.movie_id = a.movie_id
WHERE g.genre = 'Action'
    AND a.person_name LIKE '%Schwarzenegger%';
```
6. Use proper indentation!

**BAD**

```
SELECT movie_title title, movie_year year
FROM movie m
JOIN genre g ON m.movie_id = g.movie_id
JOIN actor a ON g.movie_id = a.movie_id
WHERE g.genre = 'Action' AND a.person_name LIKE '%Schwarzenegger%';
```

**GOOD**

```
SELECT movie_title title, movie_year year
FROM movie m
JOIN genre g ON m.movie_id = g.movie_id
JOIN actor a ON g.movie_id = a.movie_id
WHERE g.genre = 'Action'
    AND a.person_name LIKE '%Schwarzenegger%';
```
7. Extract uncorrelated subqueries!

**BAD**

```sql
SELECT DISTINCT person_name name
FROM director d
WHERE d.person_id IN (SELECT DISTINCT person_id
FROM actor a
JOIN movie m ON a.movie_id = m.movie_id
WHERE movie_year >= 2007);
```

**GOOD**

```sql
WITH recent_actor AS (SELECT DISTINCT person_id AS pid
FROM actor a
JOIN movie m ON a.movie_id = m.movie_id
WHERE movie_year >= 2007)
SELECT DISTINCT person_name name
FROM director d
WHERE d.person_id IN (SELECT * FROM recent_actor);
```
8 Next Lecture

- SQL data definition language
- SQL data manipulation language (apart from querying)
- SQL ≠ SQL
- Some advanced SQL concepts