ifis
Institut für Informationssysteme
Technische Universität Braunschweig

Relational Database Systems I

Wolf-Tilo Balke

Niklas Kiehne, Enrique Pinto Dominguez

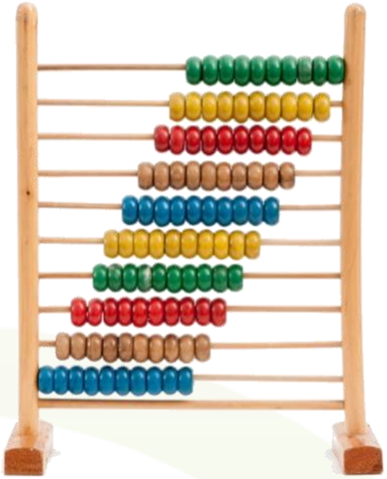
Institut für Informationssysteme

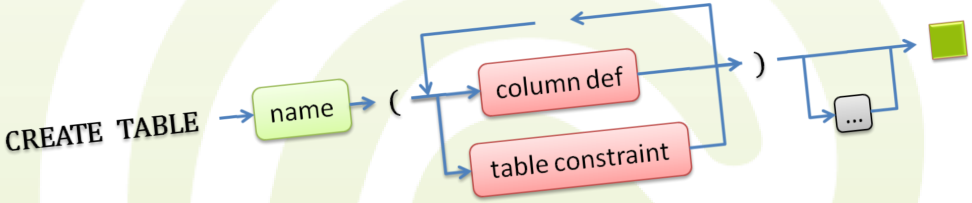
Technische Universität Braunschweig

http://www.ifis.cs.tu-bs.de

8.0 Overview of SQL

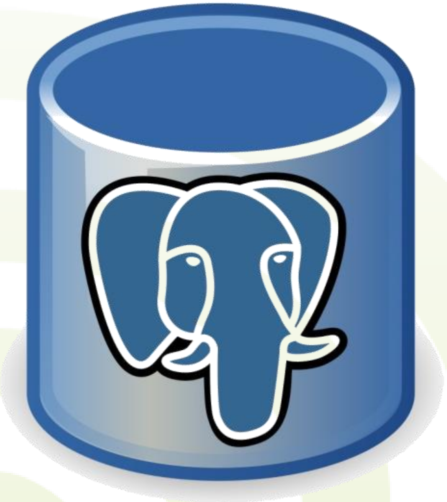
- SQL
  - Queries
    - `SELECT`
  - Data Manipulation Language (next lecture)
    - `INSERT`
    - `UPDATE`
    - `DELETE`
  - Data Definition Language (next lecture)
    - `CREATE TABLE`
    - `ALTER TABLE`
    - `DROP TABLE`





8.0 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

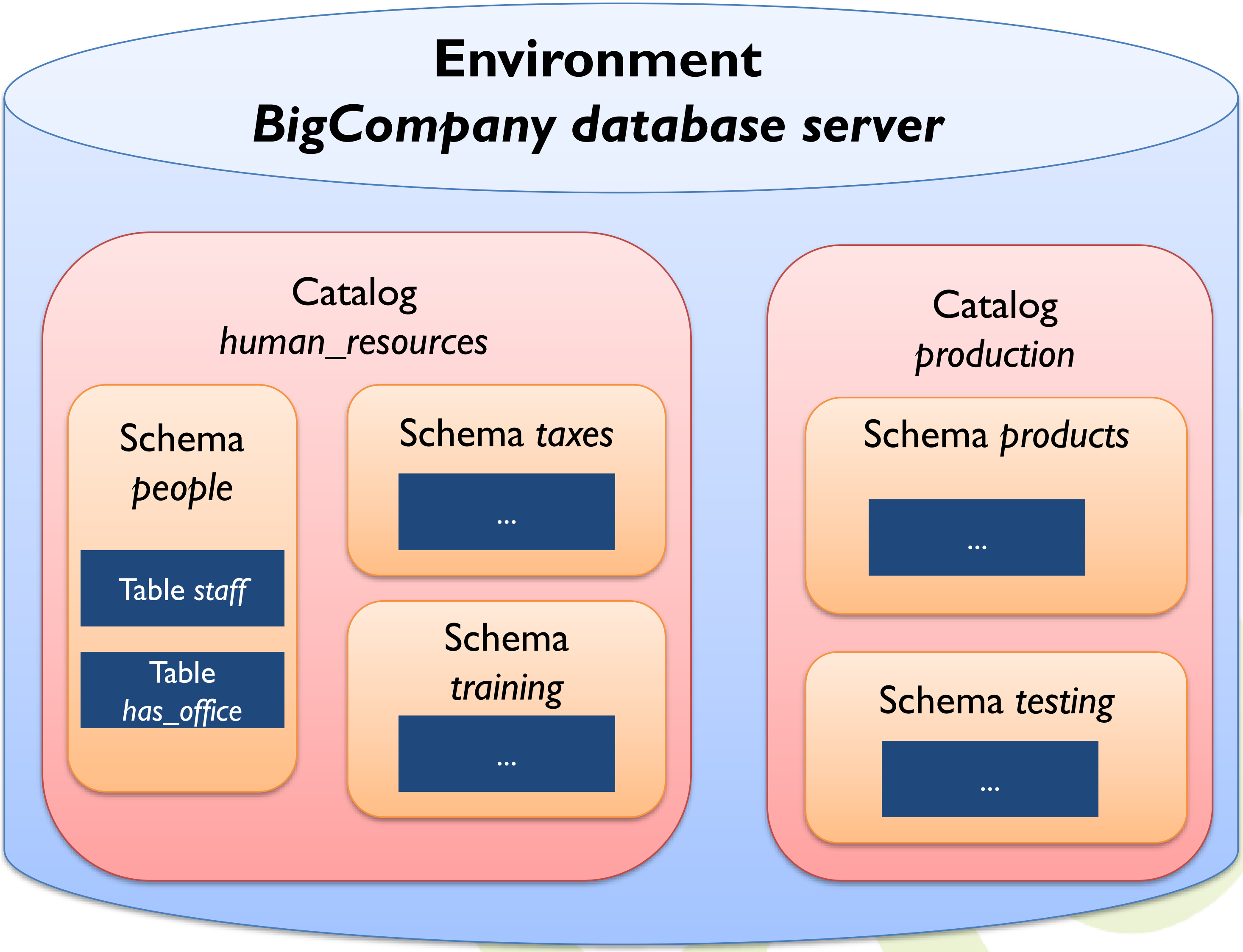


- **SQL covers all these classes**
- In this lecture, we will use **PostgreSQL 17**
  - mostly **similar** notation in other RDBMS

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



8.0 Overview of SQL

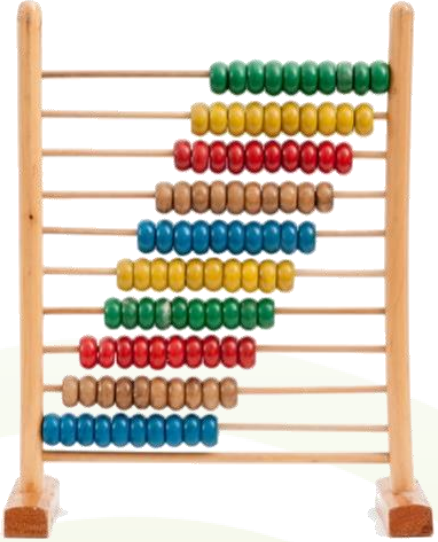
- When working with the environment, users connect to a **single catalog** and have access to all database objects in this catalog
  - however, accessing/combining data objects from different catalogs usually is not possible
  - thus, typically, catalogs are the **maximum scope** over which SQL queries 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

8.0 Overview of SQL

- 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`
  - to change the **default schema** in pgAdmin’s Query Tool:
    - for session: `SET search_path = <schema>, “$user”, public`
    - permanently for…
      - **role**: `ALTER` **`ROLE <role>`** `SET search_path = <schema>, “$<user>”, public`
      - **database**: `ALTER` **`DATABASE <database>`** `SET search_path = <schema>, “$<user>”, public`

8 SQL I

- **Basic SQL Queries**
  - **`SELECT`, `FROM`, `WHERE`**
- Advanced SQL Queries
  - **`Joins`**
  - Set operations
  - Aggregation and **`GROUP BY`**
  - **`ORDER BY`**
  - Subqueries
- Writing *Good* SQL Code



8.1 SQL Queries

- 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

```
SELECT id, lname
 FROM student
 WHERE id >= 100
```

  - TRC:

$$\{t.\text{id}, t.\text{lname} \mid \text{student}(t) \wedge t.\text{id} \geq 100\}$$

  - DRC:

$$\{\text{id}, \text{ln} \mid \exists \text{fn}(\text{student}(\text{id}, \text{fn}, \text{ln}) \wedge \text{id} \geq 100)\}$$

Rel. Algebra:

$$\pi_{\text{id},\text{lname}}\sigma_{\text{id}\geq 100}\text{student}$$

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** ρ 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

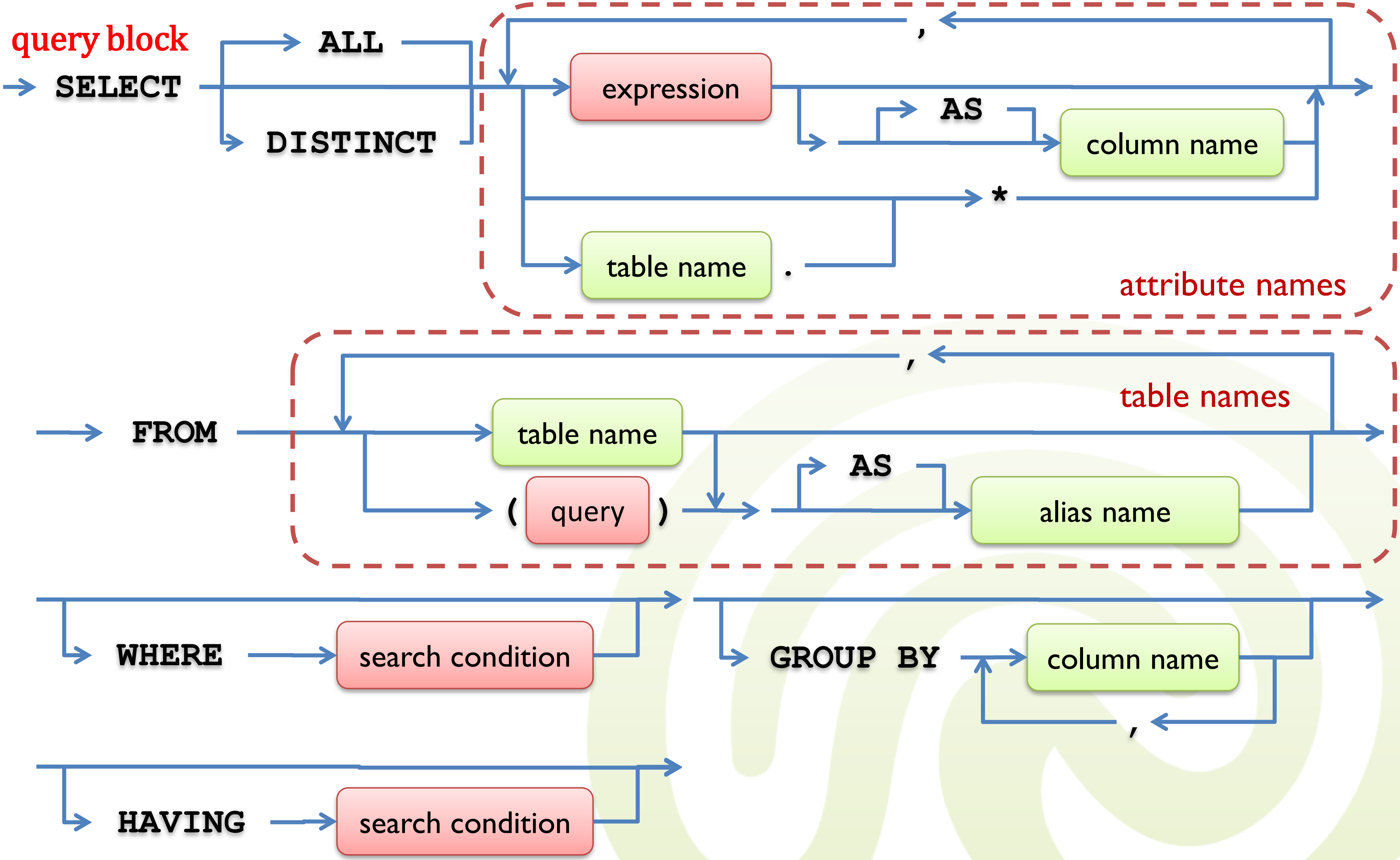
8.1 Table Names

- **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

```
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:
  $\{ m.title, g.genre \mid movie(m) \wedge genre(g) \wedge m.id = g.id \}$

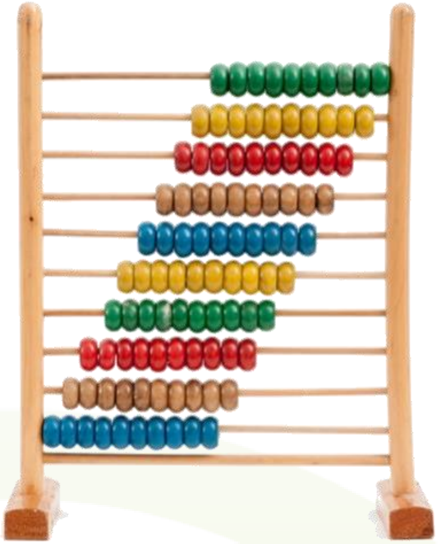
8.1 Basic Select



8.1 Expressions

- 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



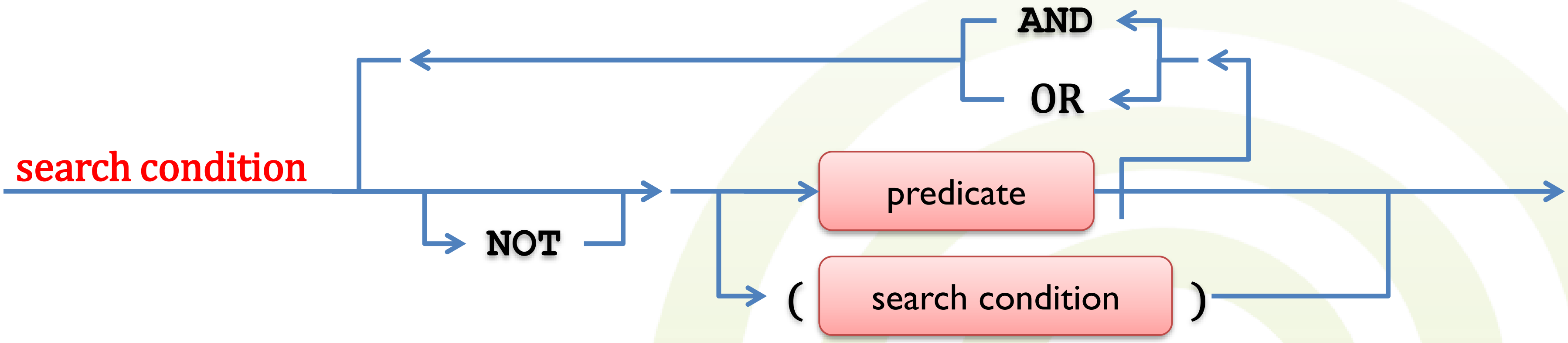
- Expressions can be combined using **expression operators**
  - **arithmetic operators:** +, -, *, and / with the usual semantics
    - `age + 2`
    - `price * quantity`
  - **string concatenation ||:** (also written as **`CONCAT`**) combines two strings into one
    - `first_name || ' ' || lastname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

8.1 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 missing*, …

8.1 Conditions

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



8.1 Conditions

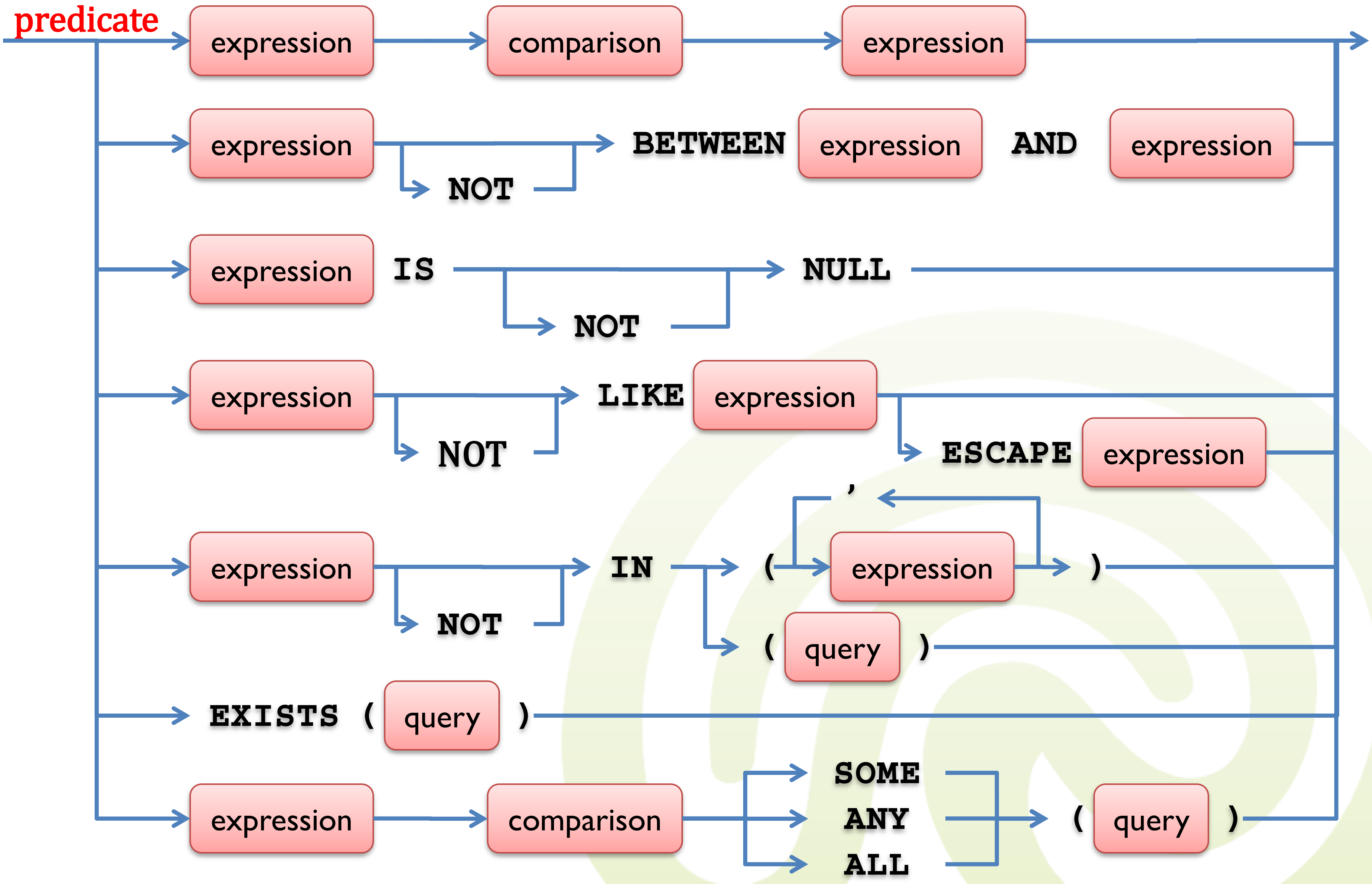
- Why **TRUE**, **FALSE**, and **NULL**?
  - SQL uses so-called ternary (three-valued) logic
  - when a predicate cannot be evaluated because it contains some **NULL** value, the result will be **NULL**
    - Example: `power_strength > 10` evalutes to **NULL** iff `power_strength` is **NULL**
    - **NULL = NULL** also evaluates to **NULL**
- Handling of **NULL** by the operators **AND** and **OR**:

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

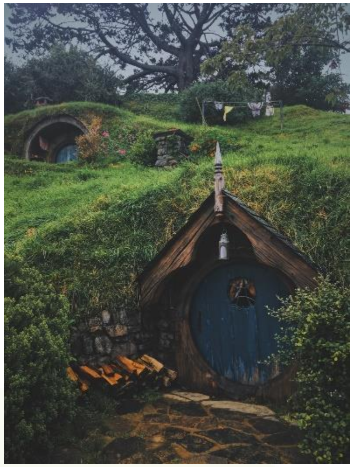
OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

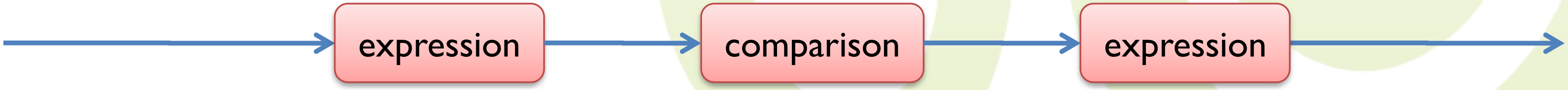
8.1 Conditions



8.1 Conditions



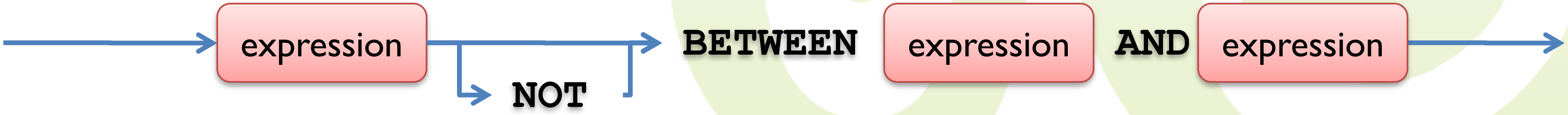
- **Simple comparisons**
  - valid comparison operators are
    - =, <, <=, >=, and >
    - <> (meaning: not equal)
  - data types of expressions need to be **compatible** (if not, CAST has to be used)
  - character values are usually compared lexicographically (while ignoring case)
  - examples
    - `patch area > 42`
    - `name = 'Tolkien'`
    - `'Martin' <= 'Tolkien'`



8.1 Conditions

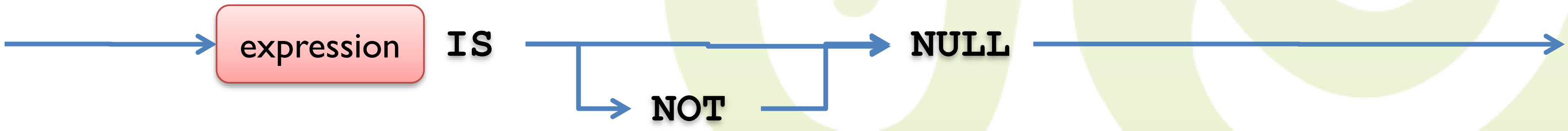
- **BETWEEN predicate:**
  - `X` **`BETWEEN`** `Y` **`AND`** `Z` is a **shortcut** for `Y <=` `X` **`AND`** `X <= Z`
  - note that you cannot reverse the order of `Z` and `Y`
    - `X` **`BETWEEN`** `Y` **`AND`** `Z` is different from `X` **`BETWEEN`** `Z` **`AND`** `Y`
    - the expression can never be true if Y > Z

  - examples
    - `year` **`BETWEEN`** `2000` **`AND`** `2008`
    - `score` **`BETWEEN`** `target_score-10` **`AND`** `target_score+10`



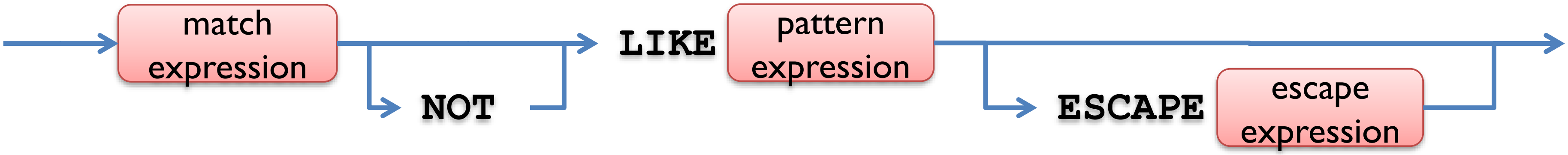
8.1 Conditions

- **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
    - `first_name` **`IS NOT NULL`**
    - `last_name` **`IS NULL`**
    - **`NULL IS NULL`**



expression **IS** **NOT** **NULL**

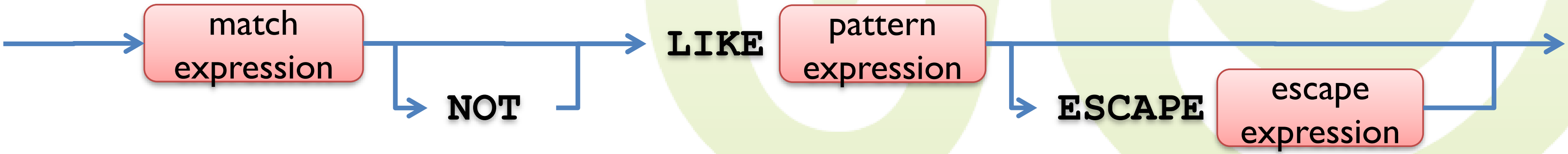
8.1 Conditions



- **`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 %
  - PostgreSQL also supports **regular expressions**
    - '~' operator

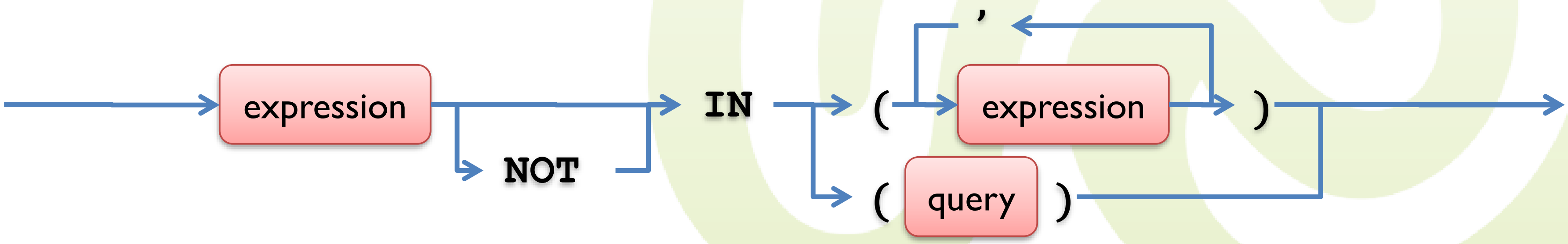
8.1 Conditions

- Examples
- address **LIKE** '%ein%'
  - 'Braunschweig' → **FALSE**
  - 'Meine' → **TRUE**
- name **LIKE** 'M%_t_'
  - 'Moritz' → **TRUE**
  - 'Matthew` → **FALSE**
  - 'Mtt' → **FALSE**
- status **LIKE** '_/_%' **ESCAPE** '/'
  - '1_sunFlower' → **TRUE**
  - '1sunFlower' → **FALSE**
  - '%_%' → **TRUE**
  - '_%%%' → **FALSE**



8.1 Conditions

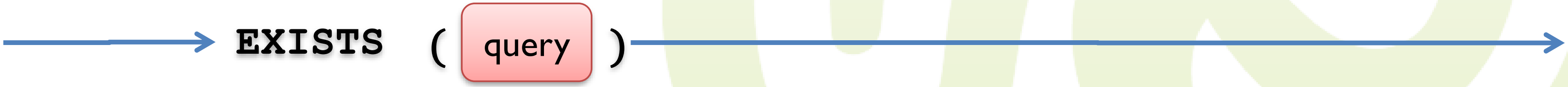
- **IN predicate**
  - evaluates to true if the value of the test expression is within a given **set of values**
  - particularly useful when used with a subquery (later)
  - examples
    - `name IN ('Mozart', 'Astley' , 'Franklin')`
    - `name IN (SELECT title FROM movie)`
      - *Those people having a film named after them…*



8.1 Conditions

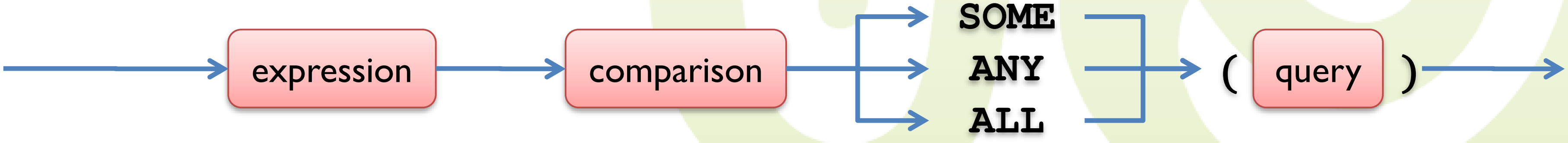
- **`EXISTS` predicate:**
  - evaluates to **`TRUE`** if a given subquery returns at least one result row
    - always returns either **`TRUE`** or **`FALSE`**

  - examples
    - **`EXISTS`** `(`**`SELECT`** `*` **`FROM`** `member)`
    - *Do we have any club member 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
    - `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



8 SQL I

- Simple SQL Queries
  - `SELECT`, `FROM`, `WHERE`
- **Advanced SQL Queries**
  - `Joins`
  - Set operations
  - Aggregation and `GROUP BY`
  - `ORDER BY`
  - Subqueries
- Writing *Good* SQL Code

8.2 Joins

- 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

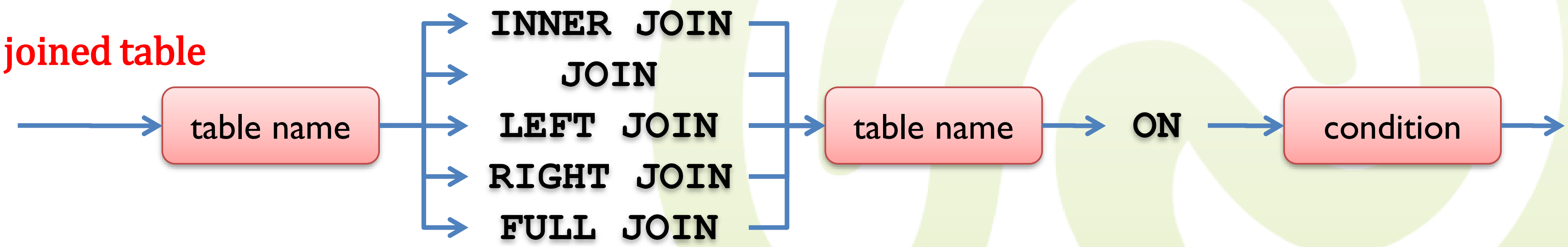
8.2 Joins

- Example
  - `SELECT * FROM member, rents_allot`
    - TRC: $\{m, ra \mid member(m) \land rents\_allot(ra)\}$
    - Rel. Algebra: $member \times rents\_allot$
  - `SELECT * FROM member m, rents_allot ra WHERE m.id = ra.mem_id`
    - TRC:
      - $\{ m, ra \mid member(m) \land rents\_allot(ra) \land m.id = ra.mem\_id \}$
    - Rel. Algebra (naïve):
      - $\sigma_{id=mem\_id} (member \times rents\_allot)$
    - Rel. Algebra (optimized):
      - $member \bowtie_{id=mem\_id} rents\_allot$
- Besides this common **implicit notation** of joins, SQL also supports **explicit joins**
  - Borrowed 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
    - explicit joins improve readability of your SQL code!



8.2 Joins

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

- $\pi_{\text{lastname, course, result}}$ (Student $\bowtie_{\text{mat\_no=student}}$ exam)
- `SELECT lastname, course, result FROM student AS s JOIN exam AS e ON s.mat_no=e.student`

Student

mat_no	firstname	lastname	sex
1005	John	McCarthy	m
2832	Grace	Hopper	f
4512	Edsger	Dijkstra	m
5119	Larry	Page	m

exam

student	course	result
9876	100	3.7
2832	102	2.0
1005	101	4.0
1005	100	1.3

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

lastname	course	result
McCarthy	100	1.3
McCarthy	101	4.0
Hopper	102	2.0

We lost Edsger Dijkstra and Larry Page 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

- $\pi_{\text{lastname, course, result}}(\text{Student} ⟕_{\text{mat\_no=student}} \text{exam})$
-
```
SELECT lastname, course, result FROM student AS s
LEFT JOIN exam AS e ON s.mat_no = e.student
```

Student

mat_no	firstname	lastname	sex
1005	John	McCarthy	m
2832	Grace	Hopper	f
4512	Edsger	Dijkstra	m
5119	Larry	Page	m

exam

student	course	result
9876	100	3.7
2832	102	2.0
1005	101	4.0
1005	100	1.3

$\pi_{\text{lastname, course, result}}(\text{Student} ⟕_{\text{mat\_nr=student}} \text{exam})$

lastname	course	result
McCarthy	100	1.3
McCarthy	101	4.0
Hopper	102	2.0
Dijkstra	NULL	NULL
Page	NULL	NULL

8.2 Joins

Right outer join:

- $\pi_{\text{lastname, course, result}}$ (Student $⟖_{\text{mat\_no=student}}$ exam)

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

Student

mat_no	firstname	lastname	sex
1005	John	McCarthy	m
2832	Grace	Hopper	f
4512	Edsger	Dijkstra	m
5119	Larry	Page	m

exam

student	course	result
9876	100	3.7
2832	102	2.0
1005	101	4.0
1005	100	1.3

$\pi_{\text{lastname, course, result}}$ (Student $⟖_{\text{mat\_no=student}}$ exam)

lastname	course	result
McCarthy	100	1.3
McCarthy	101	4.0
Hopper	102	2.0
NULL	100	3.7

8.2 Joins

Full outer join:

- $\pi_{\text{lastname, course, result}}\ (\text{Student} ⟗_{\text{mat\_no=student}} \text{exam})$
-
```
SELECT lastname, course, result FROM student s
FULL JOIN exam e ON s.mat_no = e.student
```

Student

matNr	firstname	lastname	sex
1005	John	McCarthy	m
2832	Grace	Hopper	f
4512	Edsger	Dijkstra	m
5119	Larry	Page	m

exam

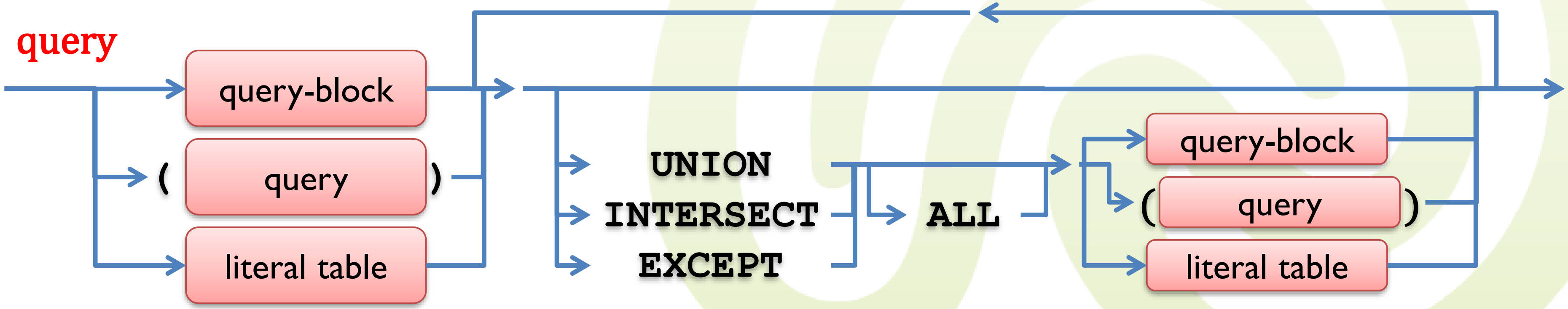
student	course	result
9876	100	3.7
2832	102	2.0
1005	101	4.0
1005	100	1.3

lastname	course	result
McCarthy	100	1.3
McCarthy	101	4.0
Hopper	102	2.0
Dijkstra	NULL	NULL
NULL	100	3.7
Page	NULL	NULL

$\pi_{\text{lastname, course, result}}\ (\text{Student} ⟗_{\text{mat\_no=student}} \text{exam})$

8.2 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)



8.2 Set Operators

- Example

  -
    ```
    ((SELECT course, result FROM exam
      WHERE course = 100)
    EXCEPT
       (SELECT course, result FROM exam
      WHERE result IS NULL))
    UNION
       VALUES (100, 2.3), (100, 1.7)
    ```

query block

query

literal table

exam

student	course	result
9876	100	3.7
2832	102	2.0
1005	101	4.0
1005	100	NULL
6676	102	4.3
3412	NULL	NULL

course	result
100	3.7
100	2.3
100	1.7

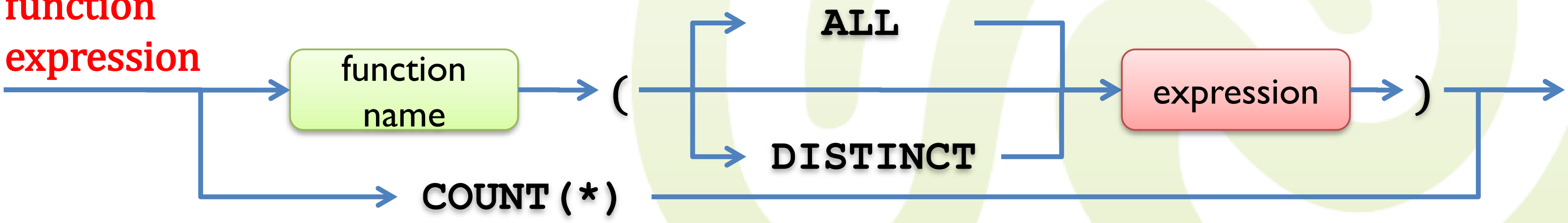
8.2 Column Function

- **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 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** (including NULL values)
  - typically, there are many more column functions available in your RDBMS (e.g. in DB2: CORRELATION, STDDEV, VARIANCE, ...)

column function expression



8.2 Column Function

- Examples

  -
    ```
    SELECT COUNT(*) FROM exam
    ```
    - *Returns the number of rows of the exam table.*

  -
    ```
    SELECT COUNT(result), COUNT(DISTINCT student)
    FROM exam
    ```
    - *Returns the number of rows in the exam table for which result is not null and the number of non-null unique students.*

  -
    ```
    SELECT MIN(result), MAX(result), AVG(result)
    FROM exam
    ```
    - *Returns the minimal, maximal, and average result in the exam 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!
    - this is a very common mistake, so make sure you really understand it!

8.2 Grouping

Examples

```
– SELECT course, AVG(result), COUNT(*), COUNT(result)
  FROM exam
  GROUP BY course
```

- *For each course, list the average result, the number of results, and the number of non-null results.*
- *Rel. Alg.:* ${}_{\text{course}}\mathfrak{F}_{\text{avg(result), count(*), count(result)}}\,\text{exam}$

exam

student	course	result
9876	100	3.7
2832	102	2.0
1005	101	4.0
1005	100	NULL
6676	102	4.3
3412	NULL	NULL

course	2	3	4
100	3,7	2	1
101	4.0	1	1
102	3.15	2	2
NULL	NULL	1	0

8.2 Grouping

Examples

```
– SELECT course, AVG(result), COUNT(*)
  FROM exam
  WHERE course IS NOT NULL
  GROUP BY course
```

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

exam

student	course	result
9876	100	3.7
2832	102	2.0
1005	101	4.0
1005	100	NULL
6676	102	4.3
3412	NULL	NULL

course	2	3
100	3.7	2
101	4.0	1
102	3.15	2

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

  -
    ```
    SELECT course, AVG(result), COUNT(*)
    FROM exam
    WHERE course <> 100
    GROUP BY course
    HAVING COUNT(*) > 1
    ```

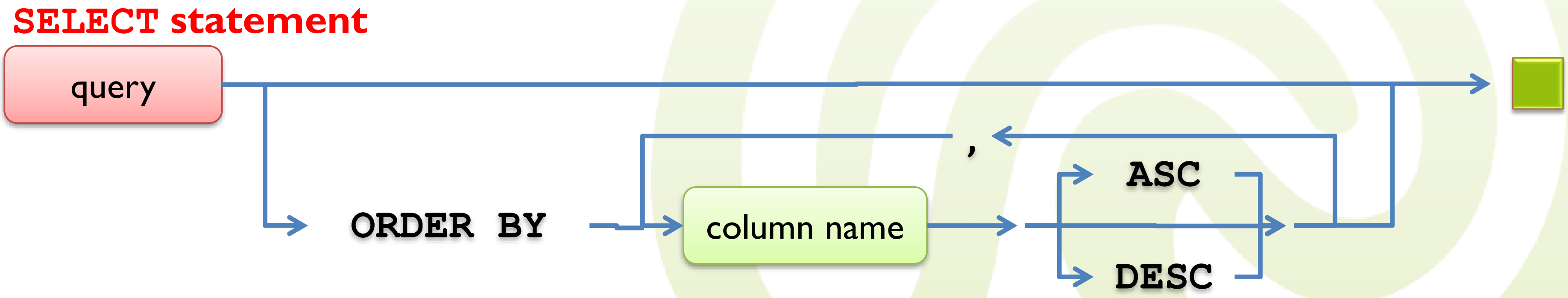
exam

student	course	result
9876	100	3.7
2832	102	2.0
1005	101	4.0
1005	100	NULL
6676	102	4.3
3412	NULL	NULL

course	1	2
102	3.15	2

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!



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
  - duplicates are not eliminated by default!

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 in PostgreSQL)
- Example

```
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>]

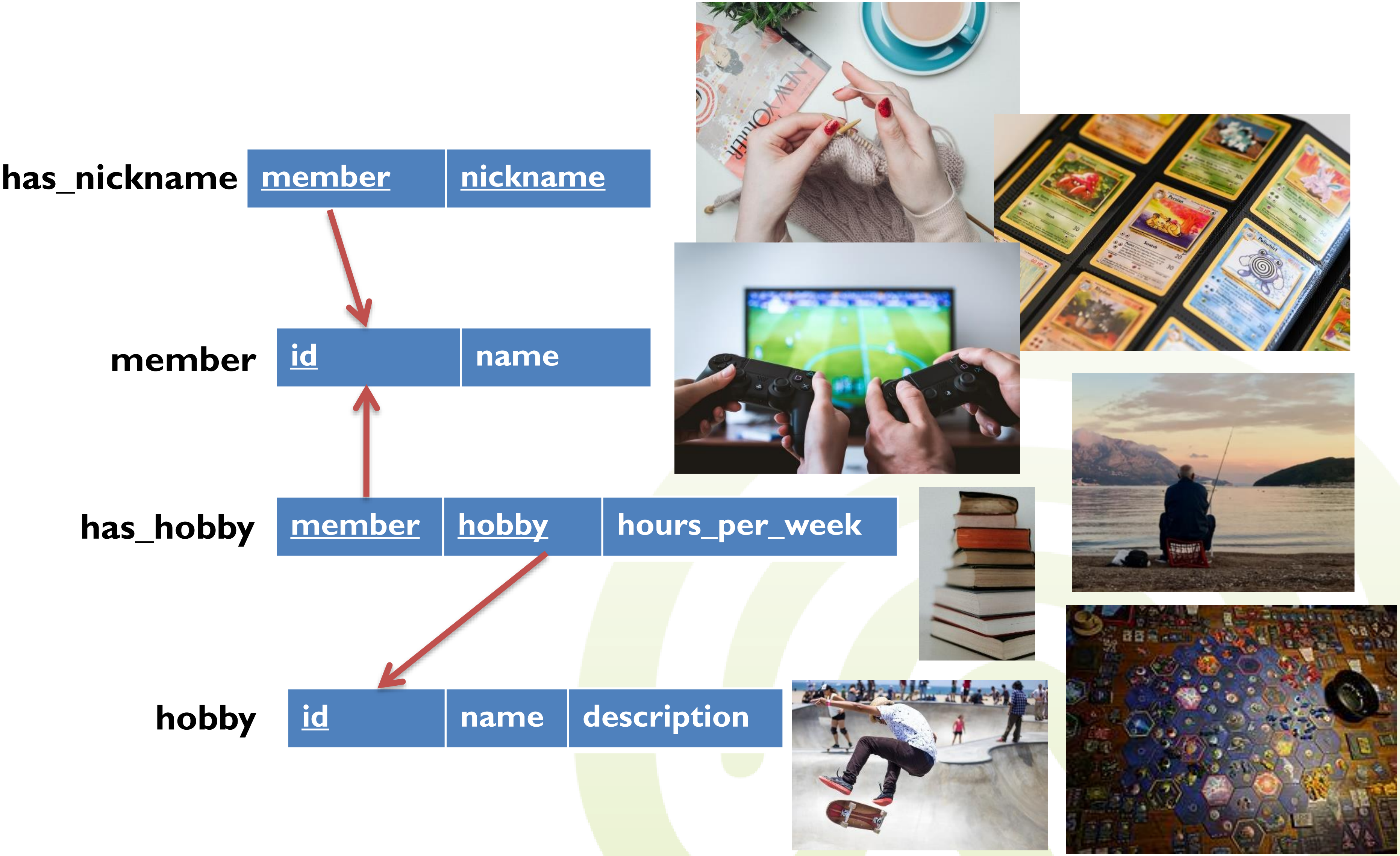
8.2 Subqueries

- In SQL, you may embed a query block within a query block (so called **subquery**, or nested query)
  - subqueries are written in parentheses
  - **scalar subqueries** can be used as **expressions**
    - if the subquery 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**

8.2 Subqueries

- 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_nickname: member, nickname

member: id, name

has_hobby: member, hobby, hours_per_week

hobby: id, name, description

8.2 Subqueries

- **Expressions**

  -
    ```
    SELECT member.* FROM member, has_hobby h
    WHERE member.id = h.member
      AND hours_per_week = (
        SELECT MAX(hours_per_week)
        FROM has_hobby
      )
    ```

    scalar subquery

    - *Select all those members pursuing hobbies with maximal time investment*
- **IN-condition**

  -
    ```
    SELECT * FROM member WHERE id IN (
      SELECT member FROM has_nickname
      WHERE nickname LIKE '%leaf'
    )
    ```
    - *Select all those members having a nickname that ends on "leaf".*

8.2 Subqueries

- **EXISTS-condition:**

  -
    ```
    SELECT * FROM member m
    WHERE EXISTS (
     SELECT * FROM has_nickname n WHERE m.id = n.member
    )
    ```
    - *Select members having at least one nickname.*
    - 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** (correlated subquery!)
- **Inline view**

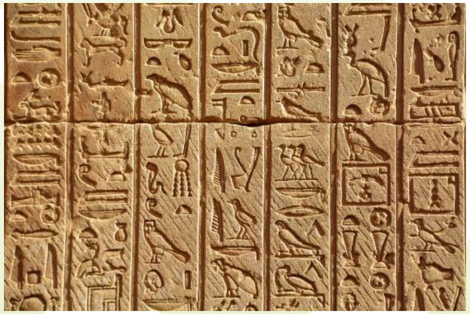
  -
    ```
    SELECT m.name, n.nickname
    FROM has_nickname n, (
      SELECT * FROM member WHERE name LIKE 'A%'
    ) m
    WHERE m.id < 100 AND n.member = m.id
    ```
    - *Get name-nickname pairs for all members with a name starting with A and an id smaller than 100.*

8.2 Subqueries

- **WITH-clause (temporary tables):**

```
WITH member_num_hobbies AS(
    SELECT member AS m_id, COUNT(*) AS num_hobbies
           FROM has_hobby GROUP BY member
)
SELECT * FROM member m
    JOIN member_num_hobbies AS mnh ON m.id = mnh.m_id
WHERE mnh.num_hobbies = (
    SELECT MAX(num_hobbies) FROM member_num_hobbies
)
```

  - *Select members having most hobbies*
  - Extremely useful if the expression in the **WITH**-clause is used multiple times
  - Also useful for readability



8 SQL I

- Simple SQL Queries
  - `SELECT`, `FROM`, `WHERE`
- Advanced SQL Queries
  - `Joins`
  - Set operations
  - Aggregation and `GROUP BY`
  - `ORDER BY`
  - Subqueries
- **Writing *Good* SQL Code**

8.3 Writing Good SQL Code

Detour

- 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

8.3 Writing Good SQL Code

Detour

1. 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

Detour

2. Use proper qualification!

BAD

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

GOOD

```
SET SCHEMA ‘imdbraw';
SELECT movie_title, movie_year
FROM movie
WHERE movie_year = 2009;
```

8.3 Writing Good SQL Code

Detour

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

BAD

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

GOOD

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

8.3 Writing Good SQL Code

Detour

4. Use joins to join!

BAD

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

GOOD

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

8.3 Writing Good SQL Code

Detour

5. Separate joins from conditions!

BAD

```
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

```
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%';
```

8.3 Writing Good SQL Code

Detour

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%';
```

8.3 Writing Good SQL Code

Detour

7. Extract uncorrelated subqueries!

BAD

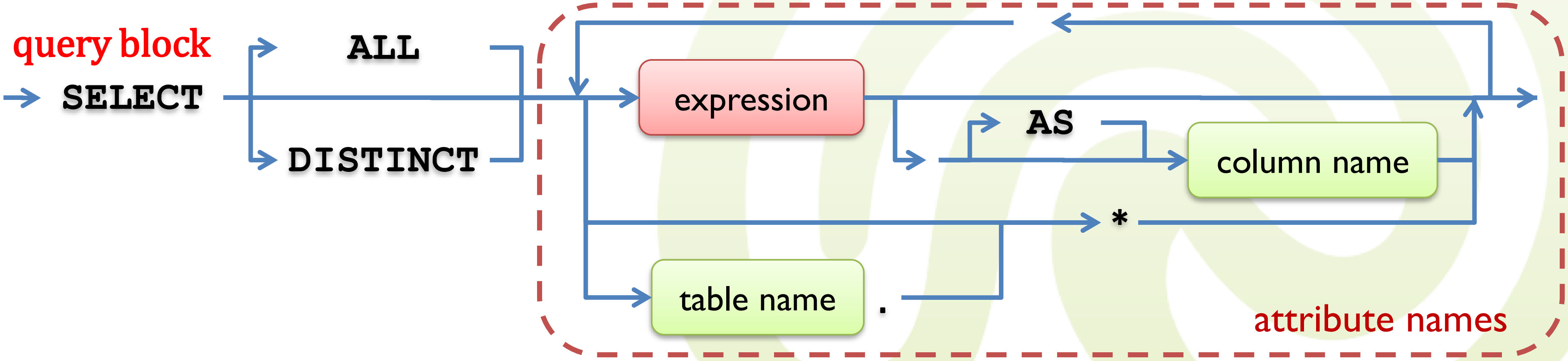
```
SELECT DISTINCT person_name name
FROM director d
WHERE d.person_id IN (
  SELECT DISTINCT person_id
  FROM actor a
    JOIN movie m ON a.movie_id = m.movie_id
  WHERE movie_year >= 2007
);
```

GOOD

```
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


query block
SELECT
ALL
DISTINCT
expression
AS
column name
*
table name
.
attribute names