ifis

Institut für Informationssysteme
Technische Universität Braunschweig

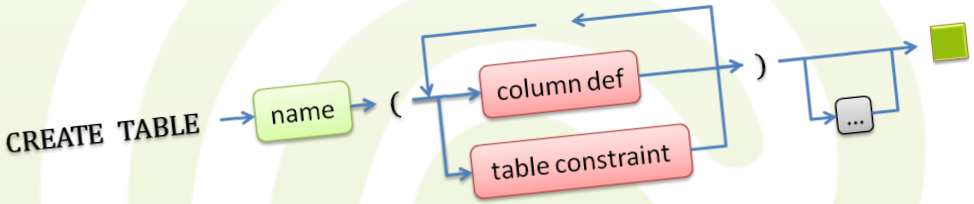
Relational Database Systems I

Wolf-Tilo Balke
Simon Barthel

Institut für Informationssysteme
Technische Universität Braunschweig
www.ifis.cs.tu-bs.de

Overview

- SQL
  - Queries
    - `SELECT`
  - Data manipulation language (next lecture)
    - `INSERT`
    - `UPDATE`
    - `DELETE`
  - Data definition language (next lecture)
    - `CREATE TABLE`
    - `ALTER TABLE`
    - `DROP TABLE`



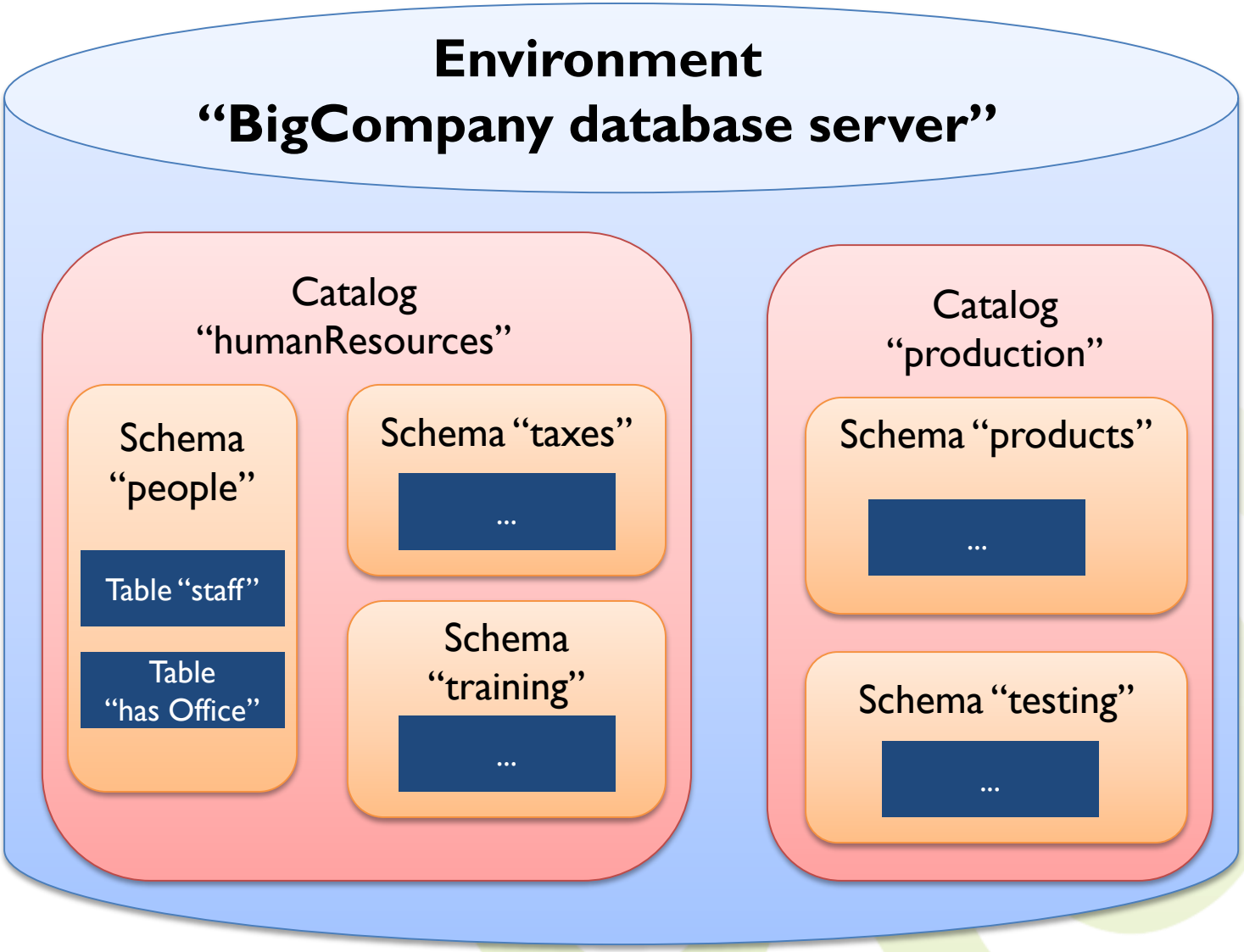
8.1 Overview of SQL

- 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 taught in this lecture)

8.1 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**

8.1 Overview of SQL



8.1 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

8.1 Overview of SQL

- After connecting to a catalog, database objects can be referenced using their **qualified name**
  - Example: `schemaname.objectname`
- However, when working only with objects from a single schema, using **unqualified names** would be nice
  - Example: `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!

8.2 SQL Queries

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

8.2 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)
    - However, often it's just called a result set…

8.2 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**

8.2 Enforcing Sets in SQL

- **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

8.2 Attribute Names

- The **SELECT** keyword is often confused with **selection** from relational algebra
  - Actually **SELECT** corresponds to **projection**
- 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

8.2 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
```

8.2 Attribute Names

- The attributes in the **result set** are defined in **`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 schema
    ```

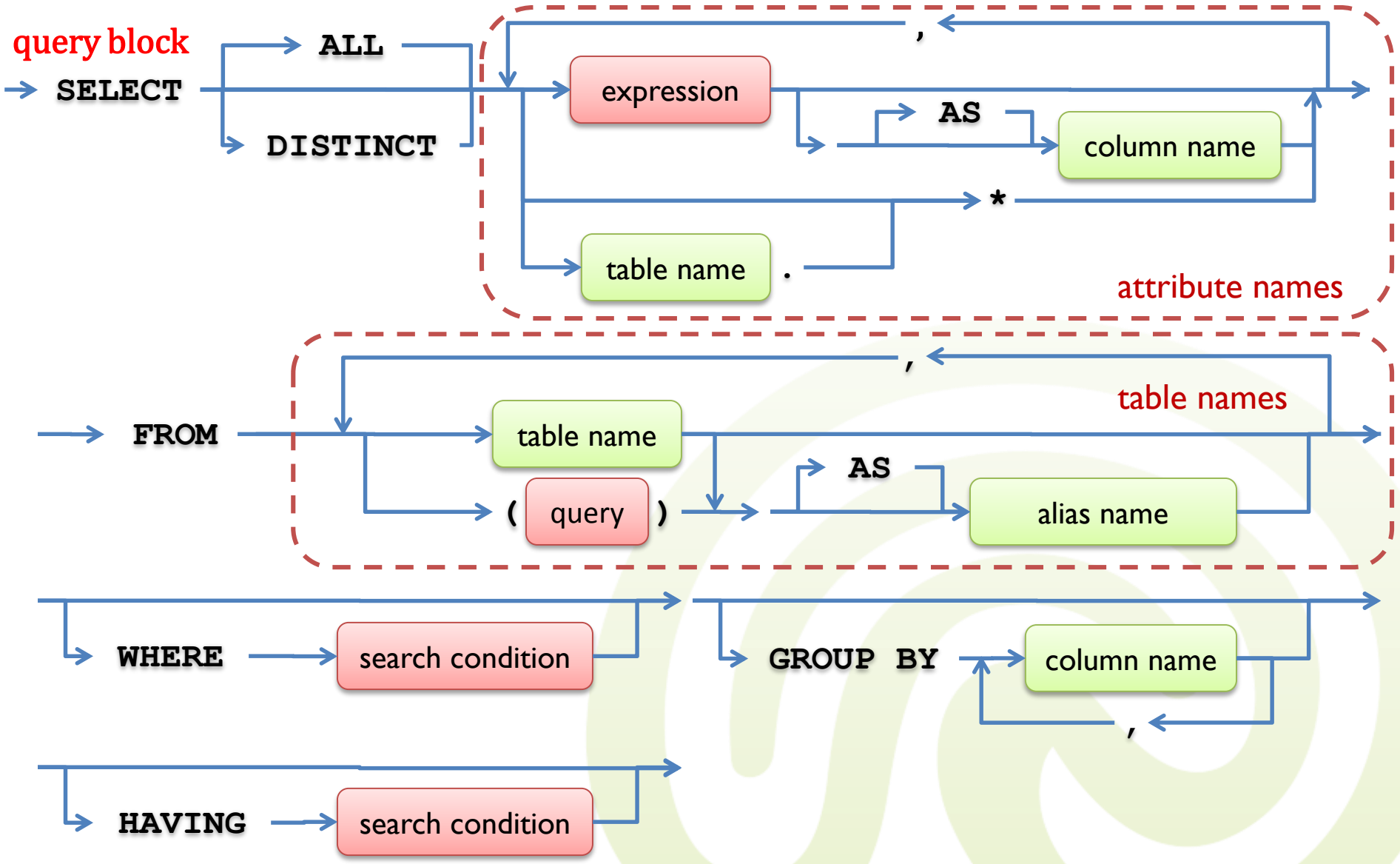
8.2 Table Names

- **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`

8.2 Basic Select

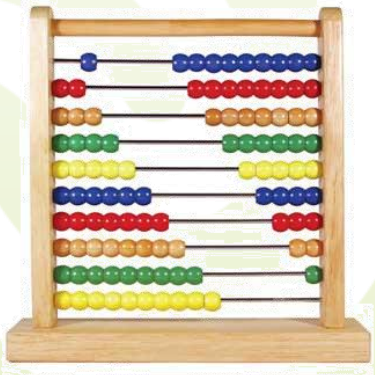


8.2 Expressions

- One of the basic building blocks of SQL queries is the **expression**.
  - **Column names** or **constants**
  - Additionally, SQL provides some **special expressions**
    - Functions
    - `CASE` expressions
    - `CAST` expressions
    - Scalar subqueries

8.2 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 || ')'`
    - `'Hello' CONCAT ' World'` → `'Hello World'`
  - **Parenthesis:**
    Used to modify the evaluation order
    - `(price + 10) * 20`

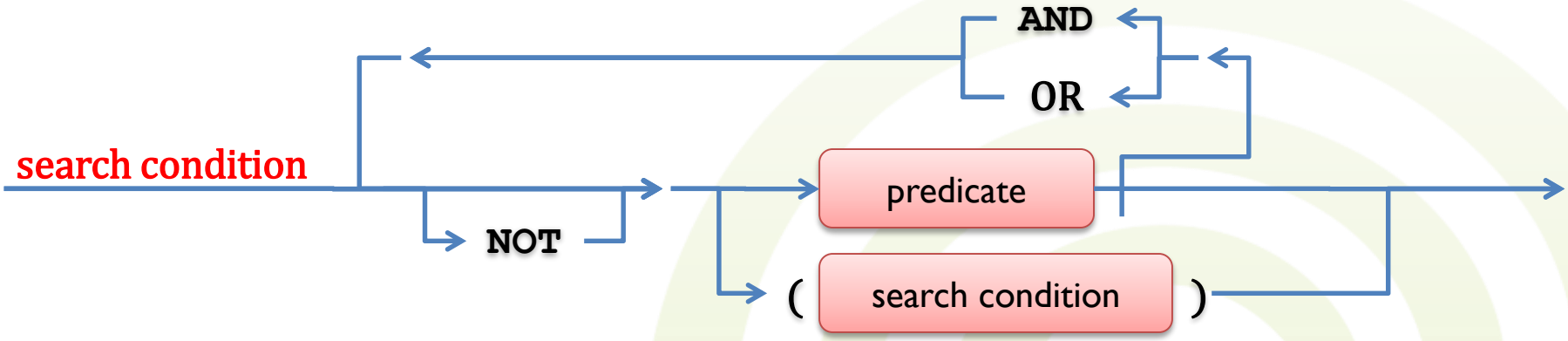


8.2 Conditions

- 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," "does not matter," ...

8.2 Conditions

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



8.2 Conditions

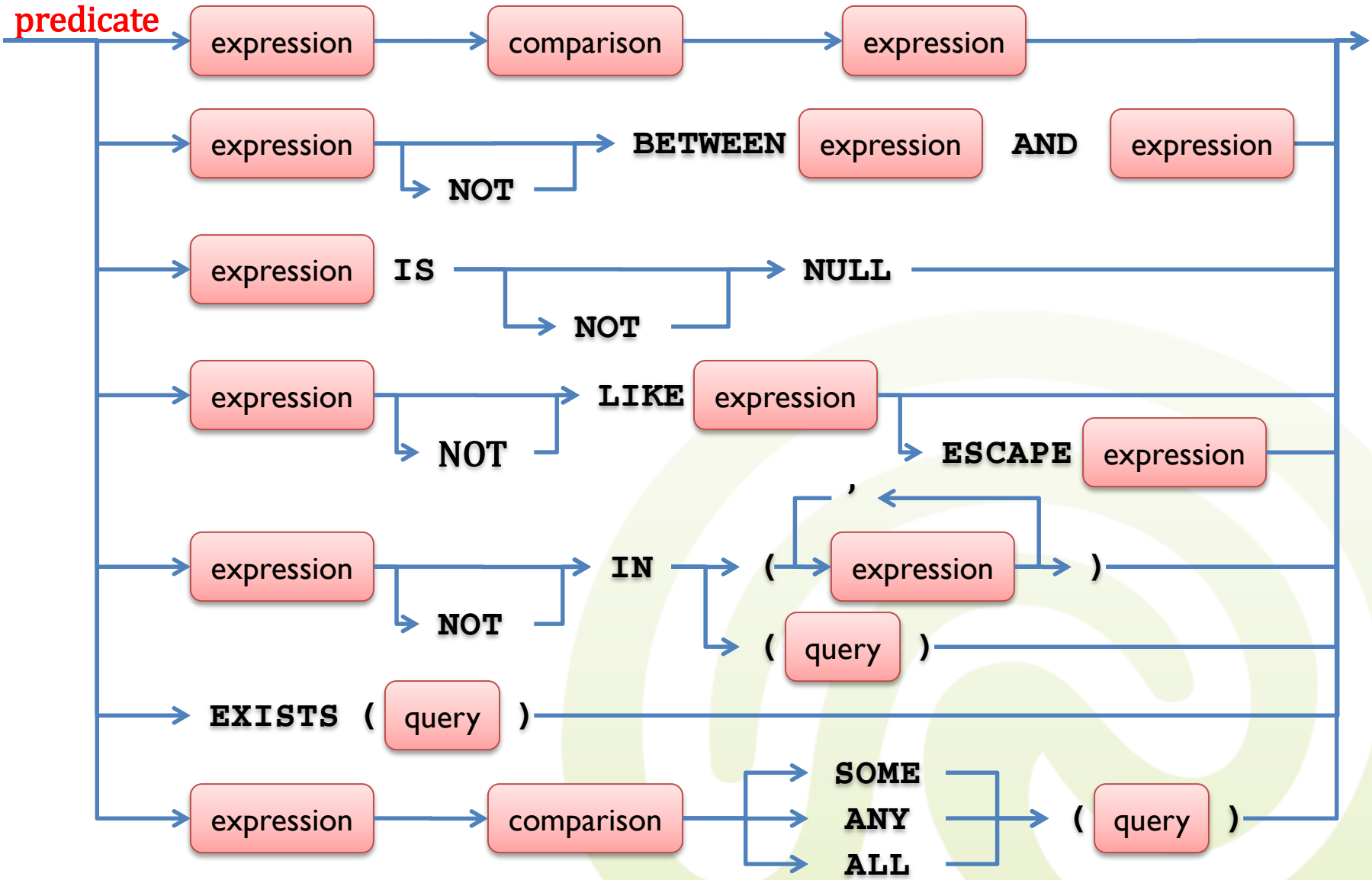
- Why **TRUE**, **FALSE**, and **NULL**?
  - SQL uses so-called ternary (three-valued) logic
  - When a predicate cannot be evaluated because it contains some **NULL** value, the result will be **NULL**
    - Example: `power_strength > 10` evalutes to **NULL** iff `power_strength` is **NULL**
    - **NULL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

8.2 Conditions



8.2 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:**
    - `powerStrength > 10`
    - `name = 'Magneto'`
    - `'Magneto' <= 'Professor X'`

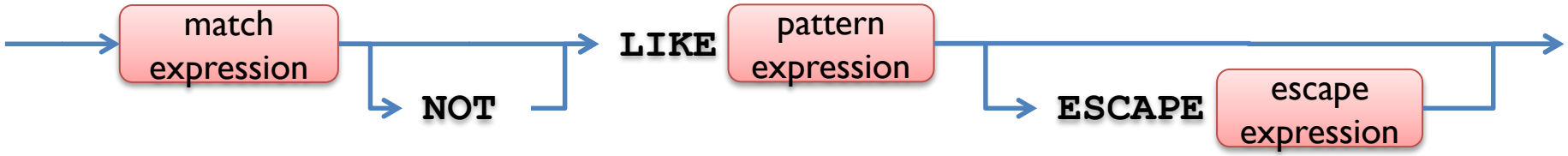
8.2 Conditions

- **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`** `targetScore-10` **`AND`** `targetScore+10`

8.2 Conditions

- **`IS NULL` predicate:**
  - The only way to check if a value is **`NULL`** or not
    - Remember: NULL = NULL returns NULL
  - Returns either **`TRUE`** or **`FALSE`**
  - Examples:
    - `real_name` **`IS NOT NULL`**
    - `power_strength` **`IS NULL`**
    - **`NULL IS NULL`**

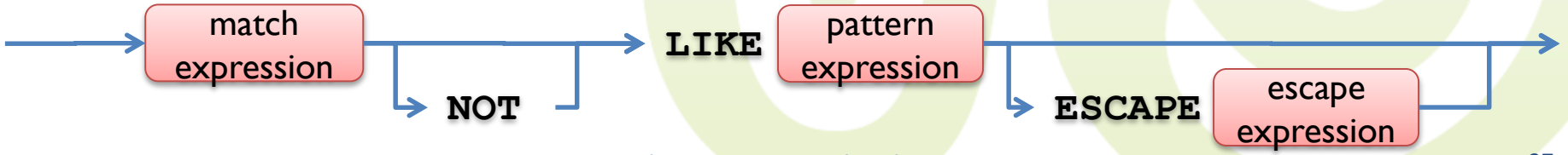
8.2 Conditions



- **`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)

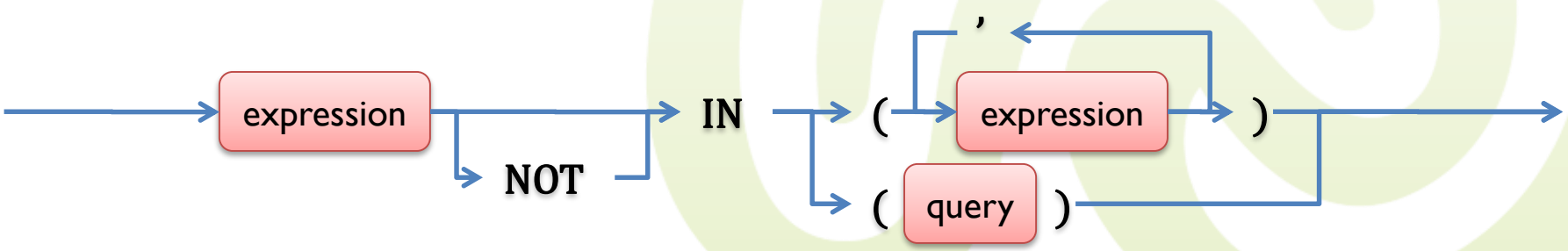
8.2 Conditions

- **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**



8.2 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…

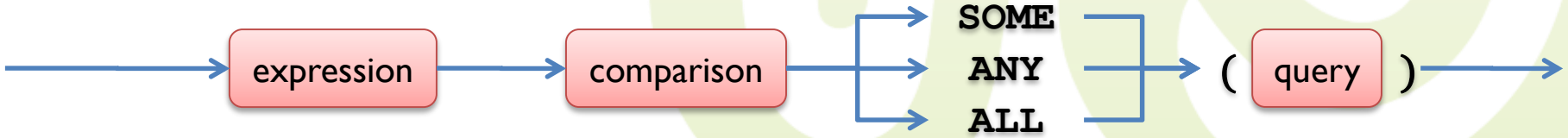


8.2 Conditions

- **`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)
  - **`EXISTS`** may also be used to express semi-joins

8.2 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



8.3 Joins

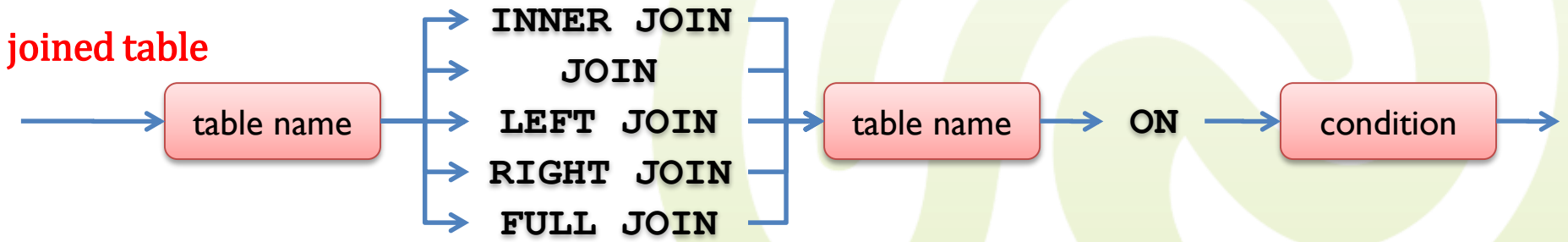
- Also, SQL can do **joins of multiple tables**
- Traditionally, this is performed by simply stating **multiple tables** in the **`FROM`** clause
  - Result contains **all possible combinations** of all rows of all tables such that the search condition holds
  - If there is no **`WHERE`** clause, it's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

8.3 Joins

- **Example:**
  - `SELECT * FROM hero, hasAlias`
    - $\text{hero} \times \text{hasAlias}$
  - `SELECT * FROM hero, hasAlias WHERE hero.id = hasAlias.hero_id`
    - $\text{hero} \bowtie_{\text{id} = \text{hero\_id}} \text{hasAlias}$
- Besides this common **implicit notation** of joins, SQL also supports **explicit joins**
  - Inner joins
  - Left/right/full outer joins

8.3 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!



8.3 Joins

Inner join: List students and their exam results

- $\pi_{\text{lastName, crsNr, result}}$ Student $\bowtie_{\text{matNr=student}}$ exam
- `SELECT lastName, crsNr, result FROM student AS s JOIN exam AS e ON s.matNr = e.student`

Student

matNr	firstName	lastName	sex
1005	Clark	Kent	m
2832	Louise	Lane	f
4512	Lex	Luther	m
5119	Charles	Xavier	m

exam

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

$\pi_{\text{lastName, crsNr, result}}$ Student $\bowtie_{\text{matNr=course}}$ exam

lastName	crsNr	result
Kent	100	1.3
Kent	101	4.0
Lane	102	2.0

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

8.3 Joins

Left outer join: List students and their exam results

- $\pi_{\text{lastName, crsNr, result}}$ Student $\bowtie_{\text{matNr=student}}$ exam
- `SELECT lastName, crsNr, result FROM student AS s LEFT JOIN exam AS e ON s.matNr = e.student`

Student

matNr	firstName	lastName	sex
1005	Clark	Kent	m
2832	Louise	Lane	f
4512	Lex	Luther	m
5119	Charles	Xavier	m

exam

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

$\pi_{\text{lastName, crsNr, result}}$ Student $\bowtie_{\text{matNr=student}}$ exam

lastName	crsNr	result
Kent	100	1.3
Kent	101	4.0
Lane	102	2.0
Luther	NULL	NULL
Xavier	NULL	NULL

8.3 Joins

Right outer join:

- $\pi_{\text{lastName, crsNr, result}}$ Student $\bowtie_{\text{matNr=student}}$ exam
- `SELECT lastName, crsNr, result FROM student s RIGHT JOIN exam e ON s.matNr = e.student`

Student

matNr	firstName	lastName	sex
1005	Clark	Kent	m
2832	Louise	Lane	f
4512	Lex	Luther	m
5119	Charles	Xavier	m

exam

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

$\pi_{\text{lastName, crsNr, result}}$ Student $\bowtie_{\text{matNr=student}}$ exam

lastName	crsNr	result
Kent	100	1.3
Kent	101	4.0
Lane	102	2.0
NULL	100	3.7

8.3 Joins

Full outer join:

- $\pi_{\text{lastName, crsNr, result}}$ Student $\ltimes_{\text{matNr=student}}$ exam
- `SELECT lastName, crsNr, result FROM student s FULL JOIN exam e ON s.matNr = e.student`

Student

matNr	firstName	lastName	sex
1005	Clark	Kent	m
2832	Louise	Lane	f
4512	Lex	Luther	m
5119	Charles	Xavier	m

exam

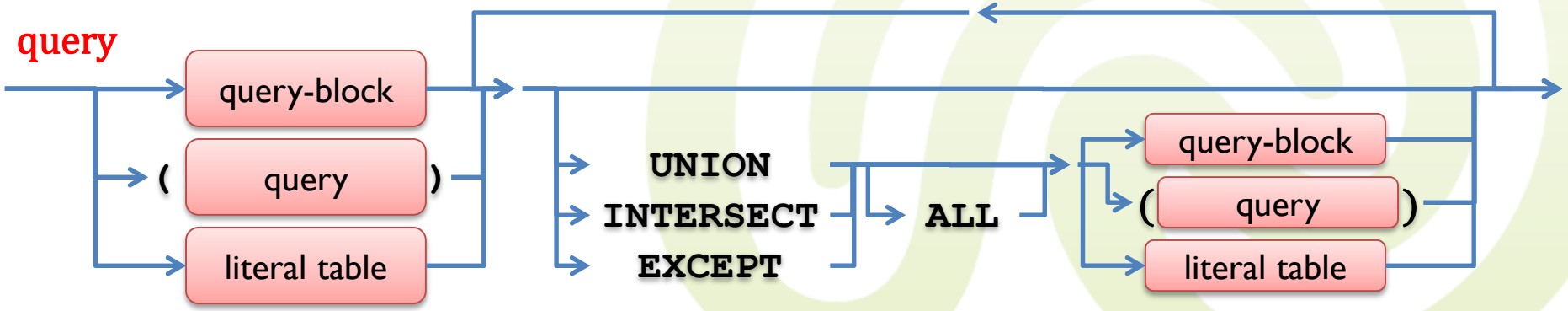
student	course	result
9876	100	3.7
2832	102	2.0
1005	101	4.0
1005	100	1.3

lastName	crsNr	result
Kent	100	1.3
Kent	101	4.0
Lane	102	2.0
Luther	NULL	NULL
NULL	100	3.7
Xavier	NULL	NULL

$\pi_{\text{lastName, crsNr, result}}$ Student $\ltimes_{\text{matNr=student}}$ exam

8.4 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)



8.4 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

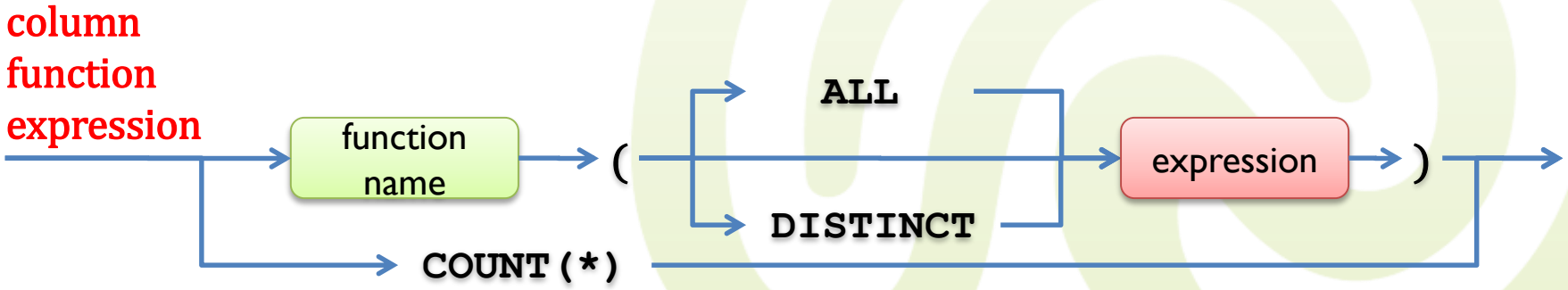
course	result
100	3.7
100	2.3
100	1.7

8.5 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
  - ...

8.5 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, ...)



8.5 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

8.5 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

8.5 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

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

8.5 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
  ```

8.5 Grouping

- Additionally, there may be restrictions on the groups themselves
  - **`HAVING`** <condition>
  - The condition may involve group properties
  - 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

8.5 Grouping

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

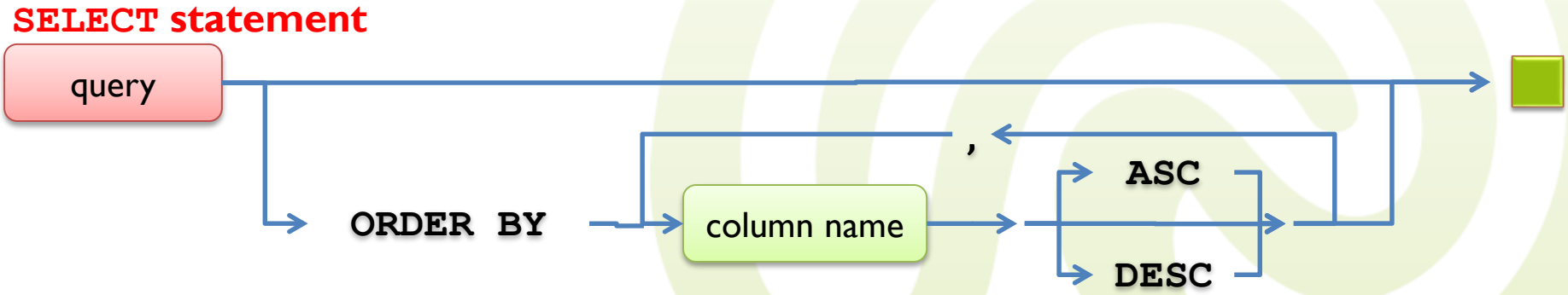
exam

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

8.6 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!



8.6 Ordering

- `ORDER BY` may order **ascending** or **descending**
  - Default: ascending
- 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


    ```

8.6 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:**

```
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

8.6 Evaluation Order of SQL

- SQL queries are evaluated in this **sequence:**

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>]

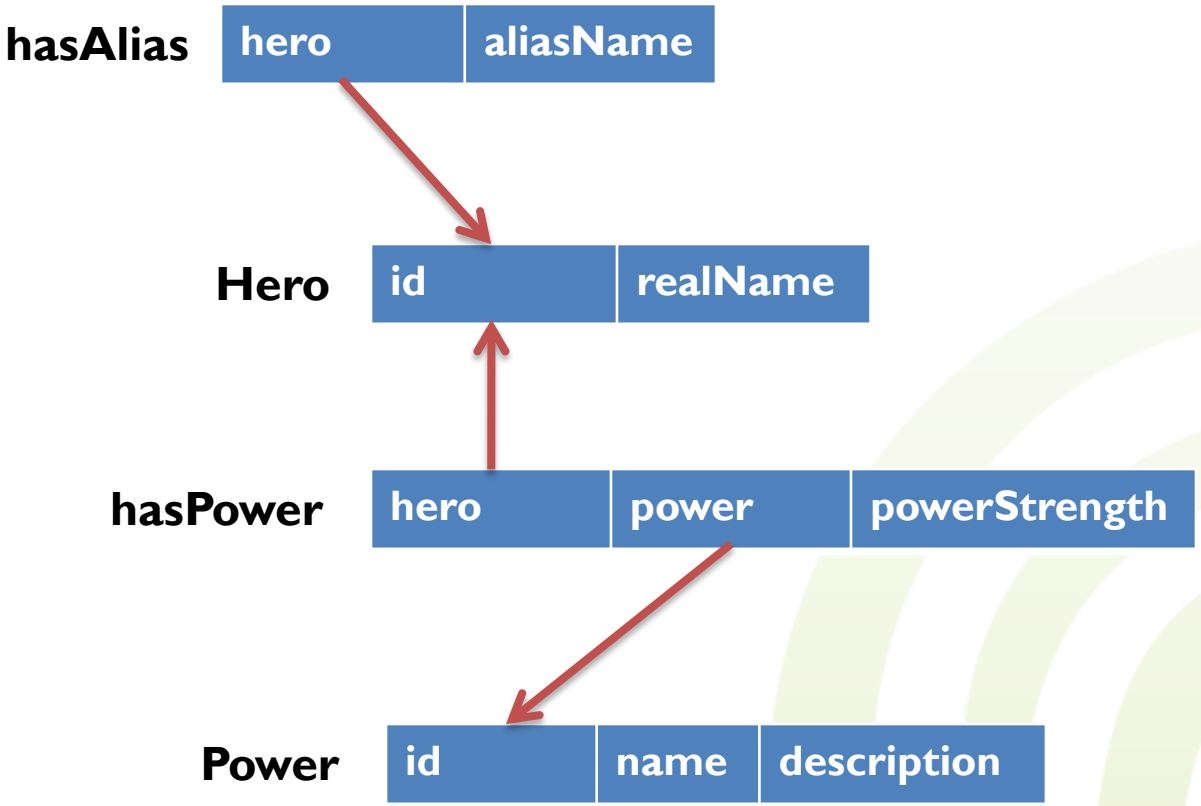
8.7 Subqueries

- 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**

8.7 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**

8.7 Subqueries



8.7 Subqueries

- **Expressions:**
  -
```
SELECT hero.* FROM hero, hasPower p
WHERE hero.id = p.hero
  AND powerStrenth = (
    SELECT MAX(powerStrength) FROM hasPower
  )
```
    - Select all those heroes having powers with maximal strength

scalar subquery

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

8.7 Subqueries

- **EXISTS-condition:**
  - ```
    SELECT * FROM hero h
    WHERE EXISTS (
      SELECT * FROM hasAlias a WHERE h.id = a.hero
    )
    ```
    - 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**
  - ```
    SELECT h.realName, a.aliasName
    FROM hasAlias a, (
      SELECT * FROM hero WHERE real_name LIKE 'A%'
    ) h
    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

8.8 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

8.8 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
```

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
```

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 title, movie_year year
FROM movie m, genre g
WHERE m.movie_id = g.movie_id
AND g.genre = 'Action'
```

Detour

4. 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%'
```

5. Use proper indentation!

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%'
```

Detour

6. 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 (person_id) AS (
  SELECT DISTINCT person_id
  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)
```

Detour

7. Avoid subqueries (use joins if possible)!

BAD

```
WITH recent_actor(person_id) AS (
  SELECT DISTINCT person_id
  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)
```

GOOD

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

Next Lecture

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