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

9 SQL 2

- **SQL data definition language**
- SQL data manipulation language (apart from **`SELECT`**)
- SQL ≠ SQL
- Some advanced SQL concepts

9.1 Recap

Summary

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

SELECT <attribute, function, scalar subquery>
FROM <table, table subquery>
[**WHERE** <condition>]
[**GROUP BY** <attribute list>]
[**HAVING** <condition>]
[**UNION/INTERSECT/EXCEPT** <query>]
[**ORDER BY** <attribute list>]

9.1 SQL DDL

- What's missing?
  - how to **create** schemas, tables, ...
  - how to **drop** schemas, tables, ...
  - how to **alter** schemas, tables, ...

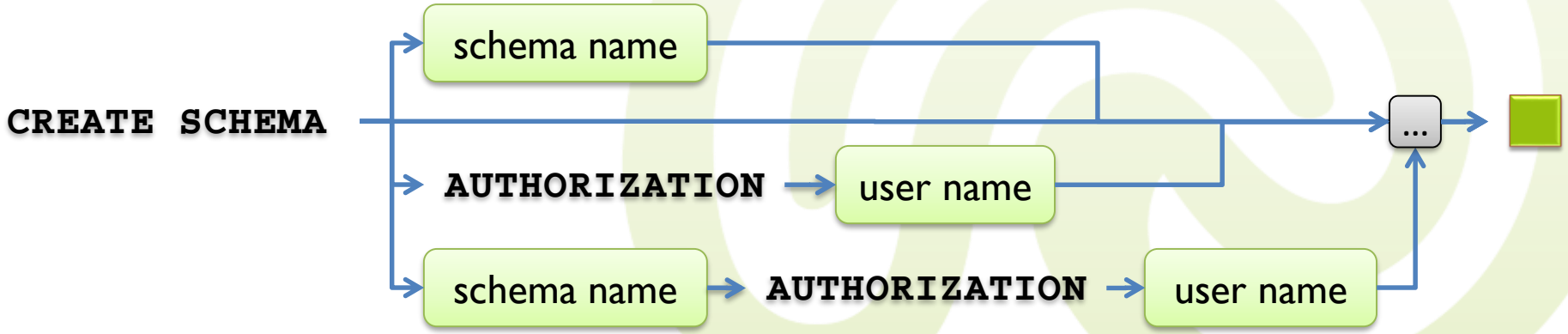
  DDL

  - how to **insert** new tuples into existing tables?
  - how to **delete** tuples from existing tables?
  - how to **update** tuples in existing tables?

  DML

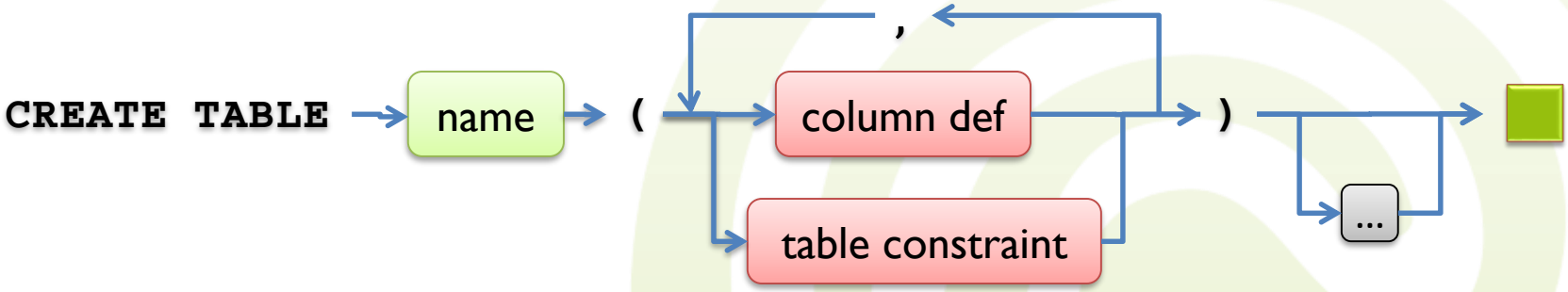
9.1 SQL DDL: Schemas

- **`CREATE SCHEMA`** creates a **new schema** with a given name for a given **owner**
  - if no schema name is provided, the **current username** is used
  - if no explicit owner is provided, also the **current user** is used
- Example
  - `CREATE SCHEMA allot_club AUTHORIZATION karl`



9.1 SQL DDL: Tables

- **`CREATE TABLE`** creates a **new table** with a given name
  - contains column definition for **each column**
  - contains additional table-specific **structural constraints**

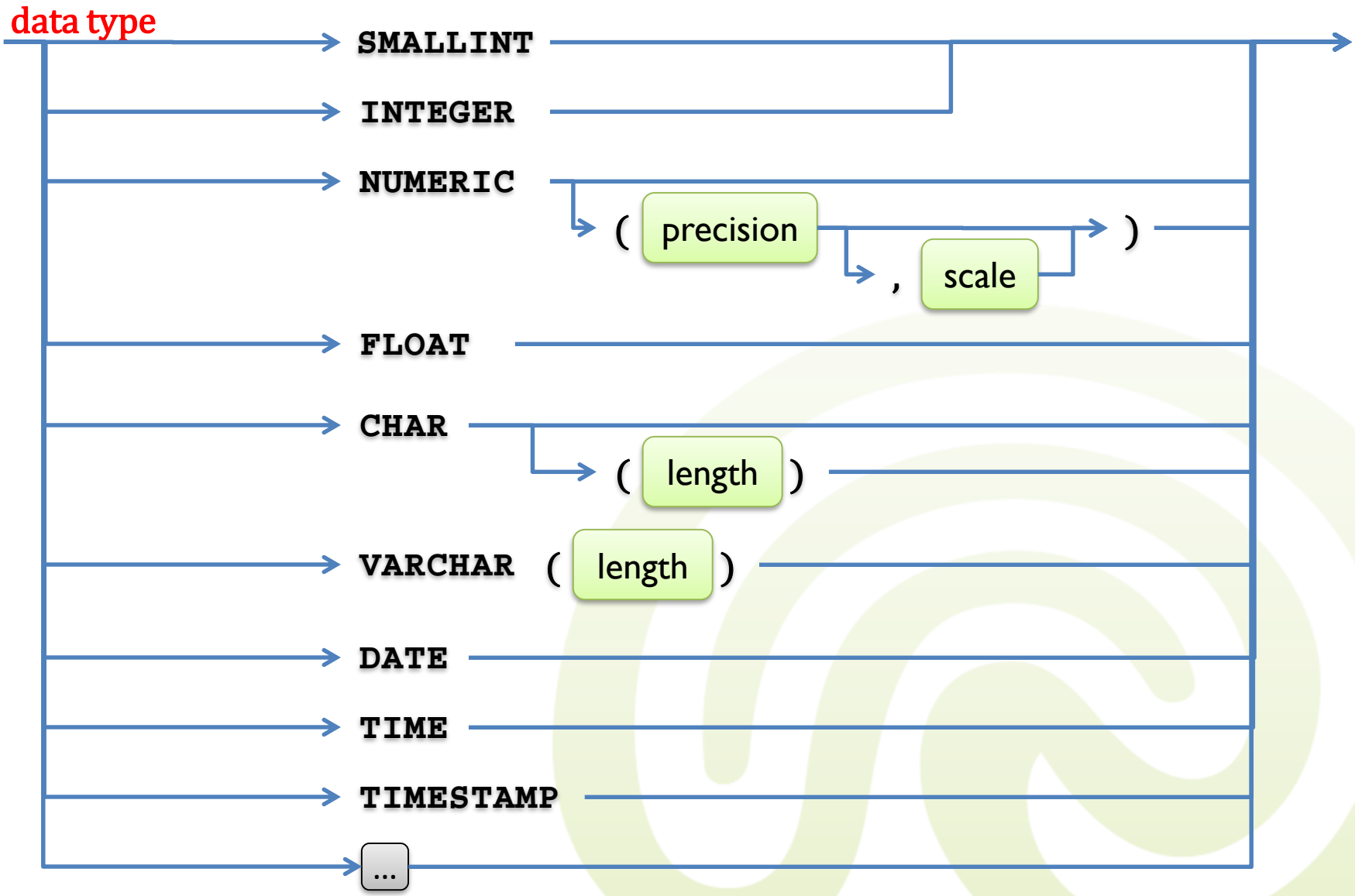


9.1 SQL DDL: Tables

- each column has a **name** and a **data type**
- each column may have multiple column **options**
- example

```
CREATE TABLE member (
        name VARCHAR(200),
        age INTEGER
)
```

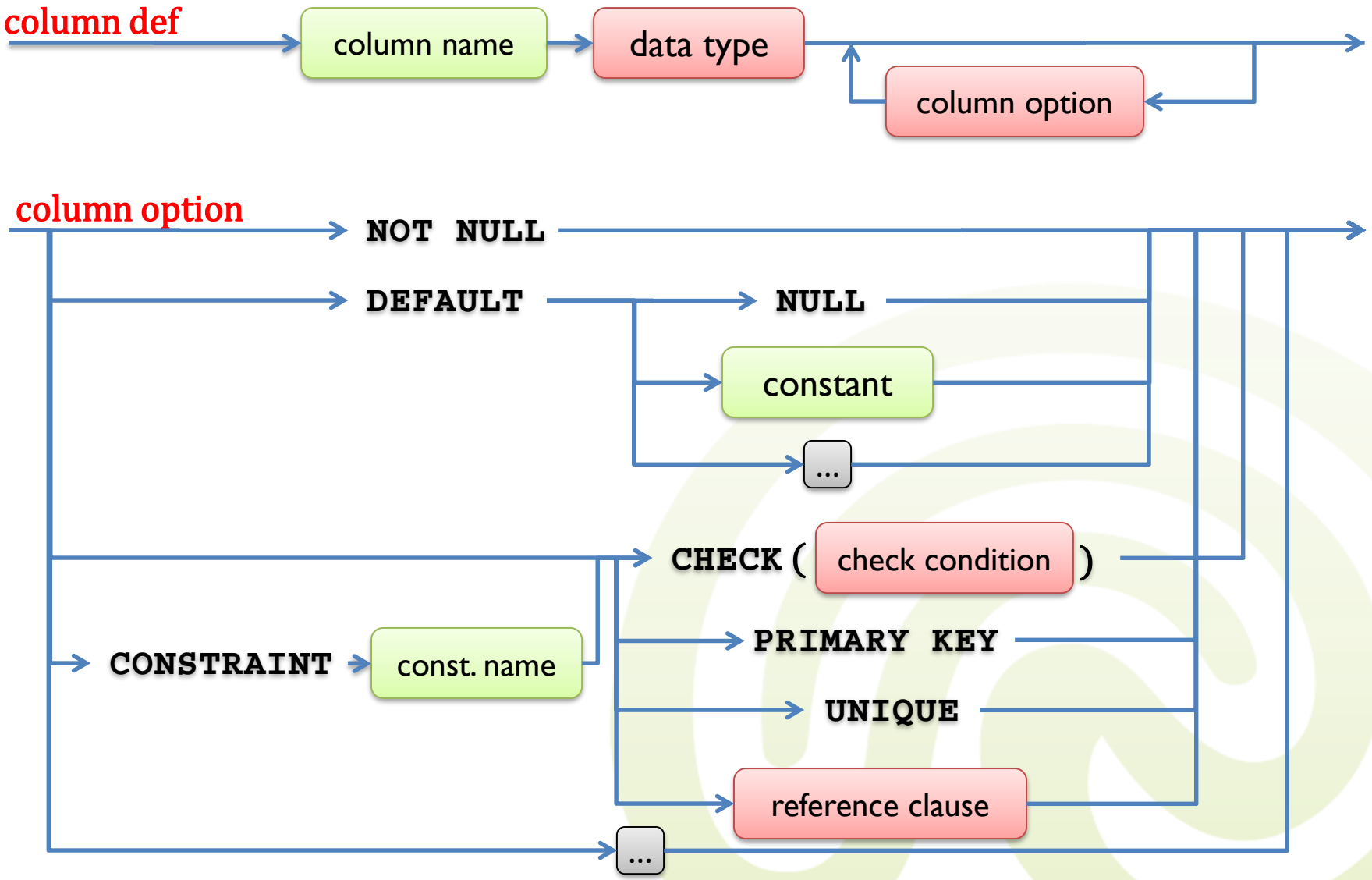
9.1 SQL DDL: Tables



9.1 SQL DDL: Tables

Name	Syntax	description
Integer	`INTEGER`	Signed four-byte integer
Float	`FLOAT/REAL/ DOUBLE PRECISION`	Floating point number of approximate precision (the supported precision is implementation-dependent)
Numeric	`NUMERIC(p,s)`	An **exact** decimal number with *p* digits: (p-s) before the decimal point, and *s* digits after the decimal point
Character	`CHAR(x)`	A textual string of fixed length *x*
Character varying	`VARCHAR(x)`	A textual string of length at most *x*
Date	`DATE`	Year, month, and day
Time	`TIME`	Hours, minutes, and seconds
Timestamp	`TIMESTAMP`	A date and a time

9.1 SQL DDL: Tables



9.1 SQL DDL: Tables

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

– example

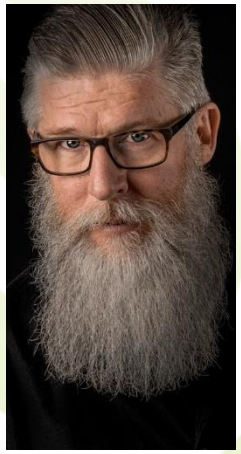
```
CREATE TABLE member (
        name VARCHAR(200) NOT NULL,
        age INTEGER NOT NULL
)
```

– **DEFAULT**:
defines the default value if a value is not explicitly set

- usually a constant or **NULL**
- if omitted, **NULL** is the default

9.1 SQL DDL: Tables

- **Column constraints**
  - restrict possible values for the current column
  - may have a **unique name** indicated by **`CONSTRAINT`** <name>
    - if name is omitted, system creates a default name
  - **`CHECK`**: user-defined constraint.
    To be valid, values have to satisfy the condition.
  - example
    - ```
      CREATE TABLE person (
            name VARCHAR(200),
            age INTEGER CONSTRAINT adult
                CHECK (age >= 18)
      )
      ```


      ```

9.1 SQL DDL: Tables

- **UNIQUE**: no duplicate values are allowed within this attribute
  - For multiple attributes, you need a different option (later)
  - NULL values are not considered equal
- example
  - ```
    CREATE TABLE person (
          name VARCHAR(200) NOT NULL UNIQUE,
          age INTEGER NOT NULL
    )
    ```

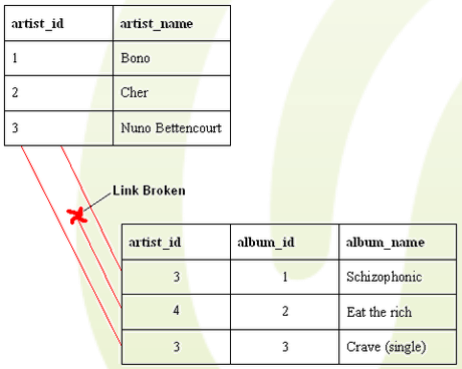

    ```

9.1 SQL DDL: Tables

- **`PRIMARY KEY`**: each table may have a **primary key** (optionally, but recommended) made up of at least one column
  - this option can only be used if the primary key consists of only one column
  - for multi-column primary keys, you need a different option (later)
  - implies **`NOT NULL`** and **`UNIQUE`**
- additionally, a **referential clause** may be specified (see next slides)

9.1 SQL DDL: Referential Integrity

- Rows in tables may **refer** to rows in other tables to capture **relationships**
- Of course, you should not be allowed to refer to a non-existing row
  - **referential integrity** between **primary keys** and **foreign keys** ensures that references are correct



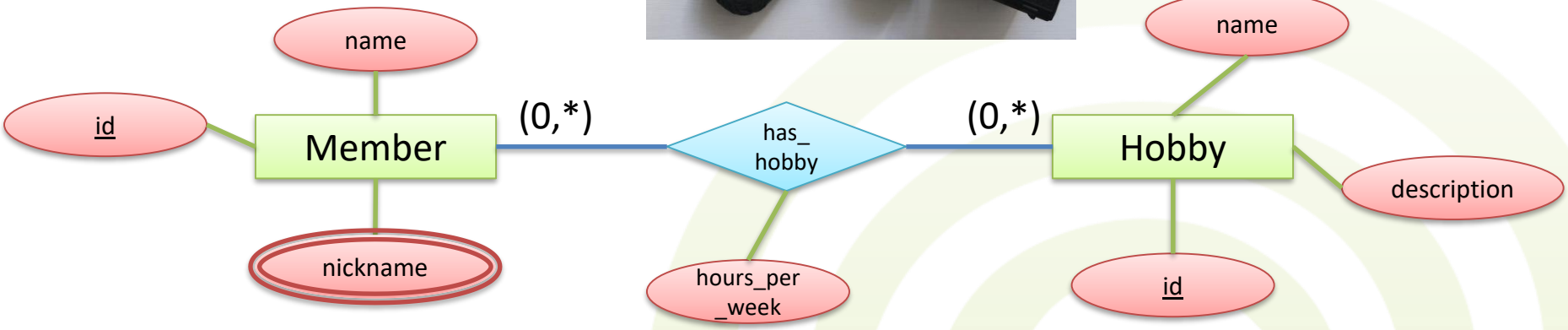
artist_id	artist_name
1	Bono
2	Cher
3	Nuno Bettencourt

artist_id	album_id	album_name
3	1	Schizophonic
4	2	Eat the rich
3	3	Crave (single)

9.1 SQL DDL: Referential Integrity

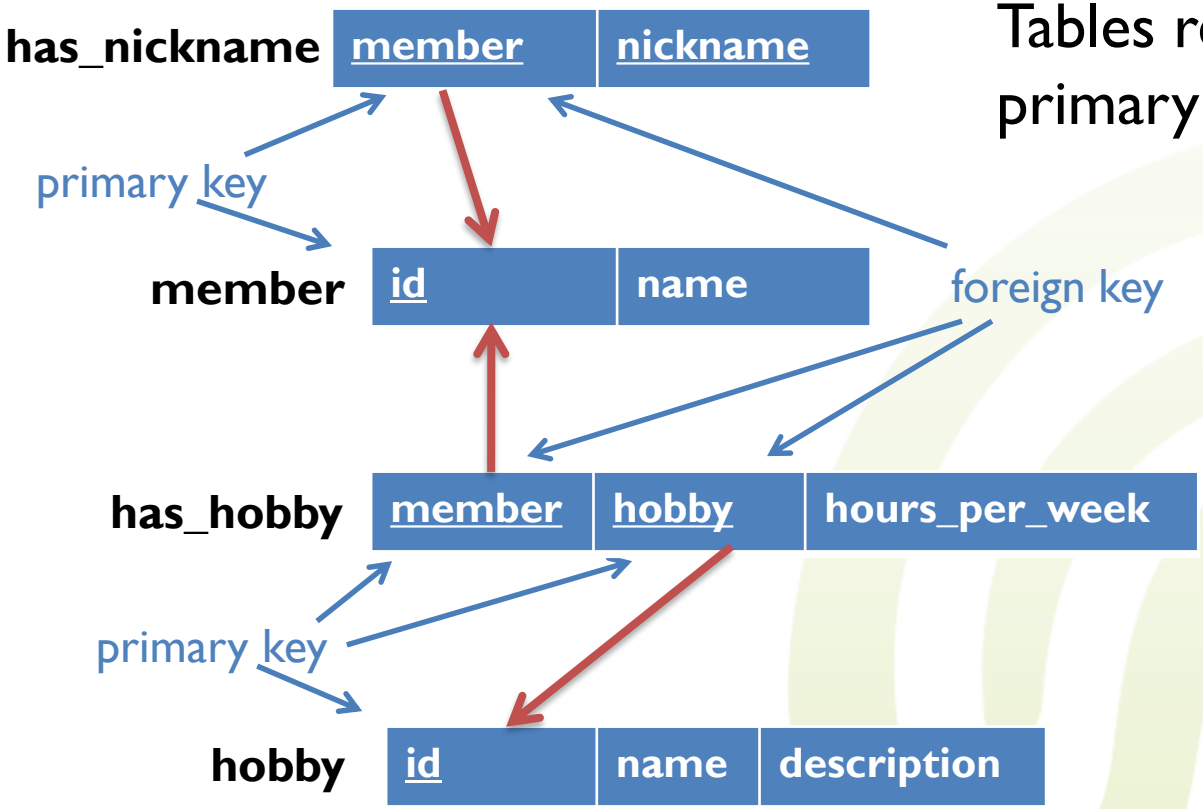
Example

Conceptual ER schema



9.1 SQL DDL: Referential Integrity

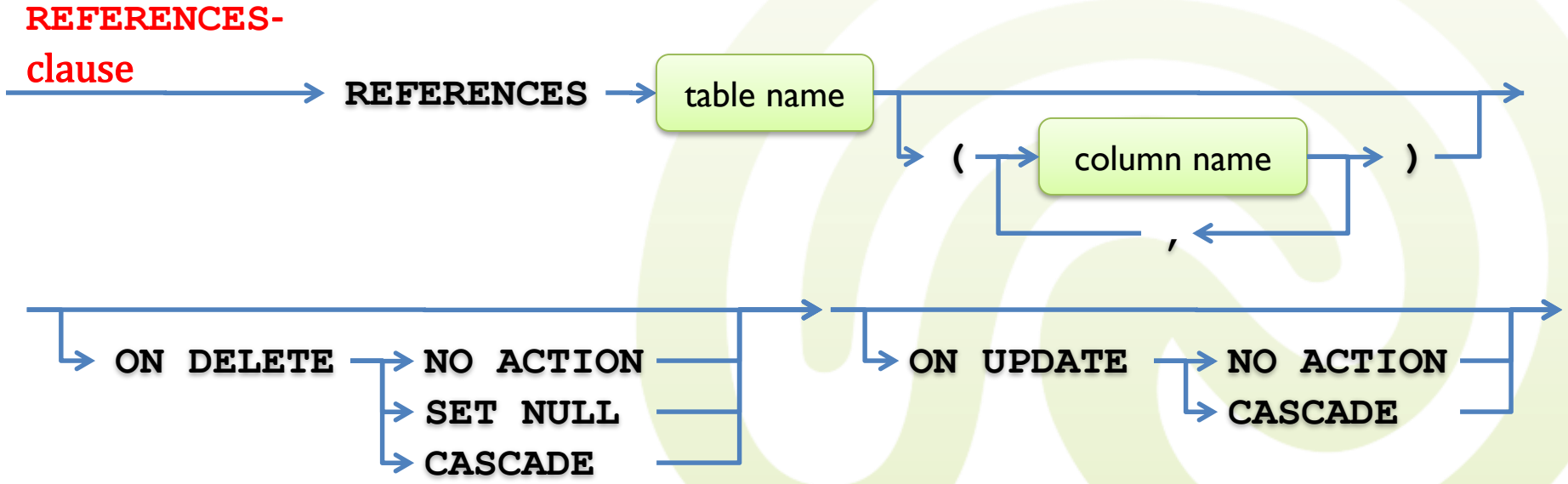
Resulting tables



Tables refer to others by primary keys and foreign keys

9.1 SQL DDL: Referential Integrity

- Referential integrity can be defined using the **REFERENCES clause**
  - either used by constraints in **column options** or within **table constraints**
  - if no attribute is selected, it refers to the primary key

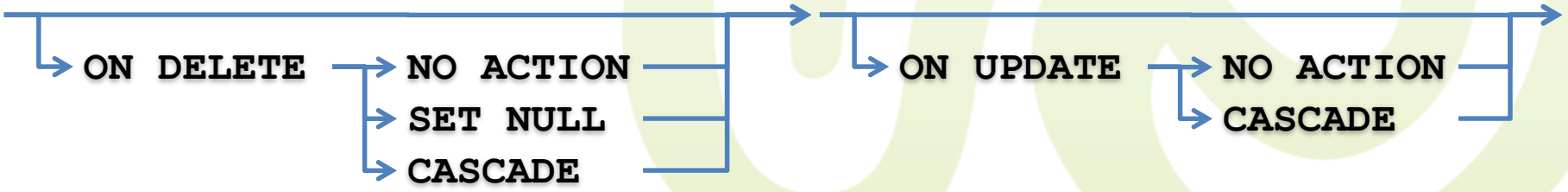


9.1 SQL DDL: Referential Integrity

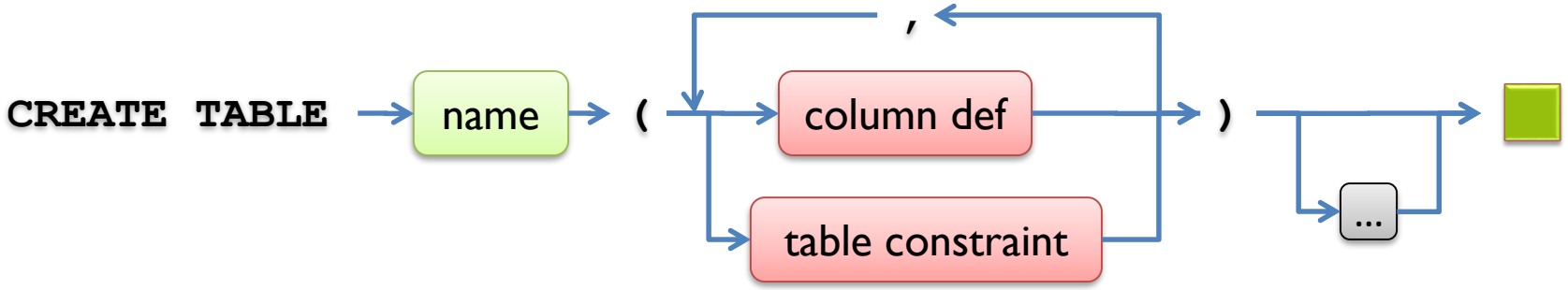
- Example
  -
```
CREATE TABLE employee(
        id INTEGER NOT NULL PRIMARY KEY,
        name VARCHAR(100) NOT NULL
)
```
  -
```
CREATE TABLE managed_by(
        employee INTEGER NOT NULL
            REFERENCES employee,
        manager INTEGER NOT NULL
            REFERENCES employee
)
```

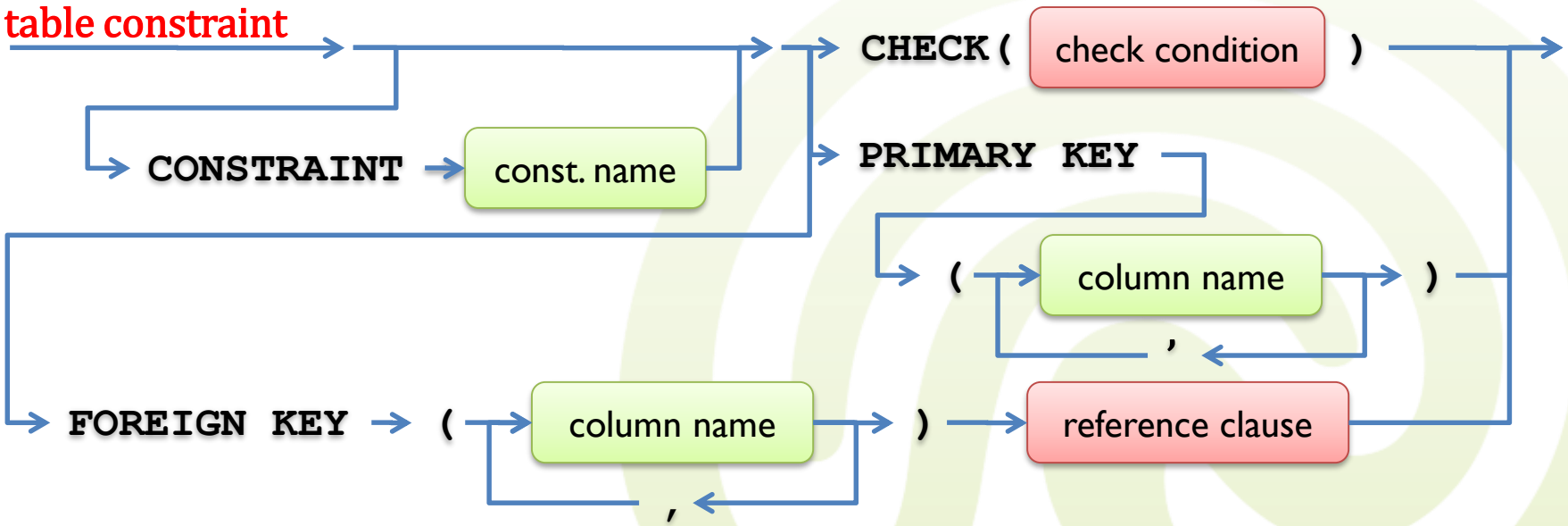
9.1 SQL DDL: Referential Integrity

- Optionally, you may specify what happens if a row that is referenced will be deleted or modified
  - **`ON DELETE`**: if a referenced row is deleted, ...
    - **`NO ACTION`**: ...reject the deletion (that is, it cannot be performed)
    - **`SET NULL`**: ...delete it and set all referencing foreign keys to **`NULL`**
    - **`CASCADE`**: ...delete it along with all rows referring to it
  - **`ON UPDATE`**: if the primary key of a referenced row is modified, ...
    - **`NO ACTION`**: ...reject the modification (that is, it cannot be performed)
    - **`CASCADE`**: ...change all values of referencing foreign keys
  - default
    - **`ON DELETE NO ACTION ON UPDATE NO ACTION`**



9.1 SQL DDL: Table Constraints

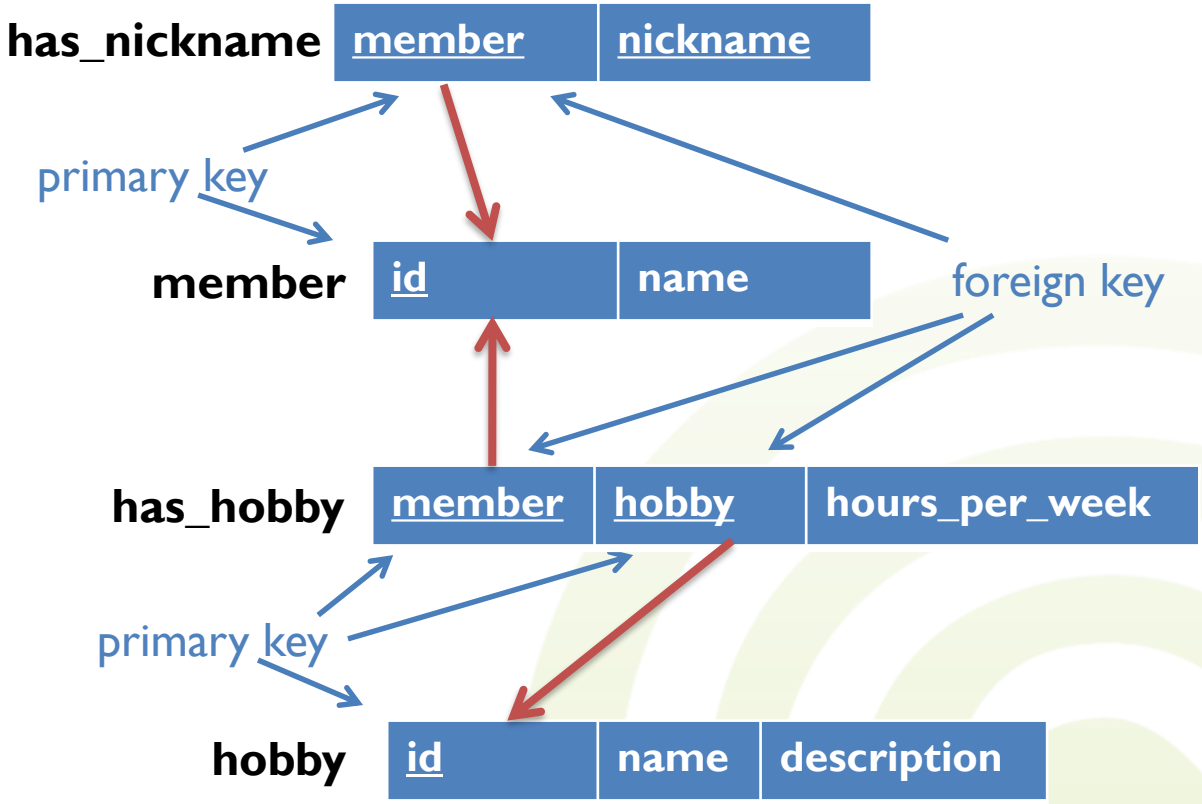




9.1 SQL DDL: Table Constraints

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

9.1 SQL DDL: Table Example



9.1 SQL DDL: Table Example

```
CREATE TABLE member(
    id INTEGER NOT NULL PRIMARY KEY,
    name VARCHAR(100)
);
CREATE TABLE hobby(
    id INTEGER NOT NULL PRIMARY KEY,
    name VARCHAR(100),
    description VARCHAR(255)
);
CREATE TABLE has_nickname (
    member INTEGER REFERENCES member
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    nickname VARCHAR(100) NOT NULL,
    PRIMARY KEY (member, nickname)
);
```

link has_nickname to member

delete nickname if member is deleted

update nickname if member is updated

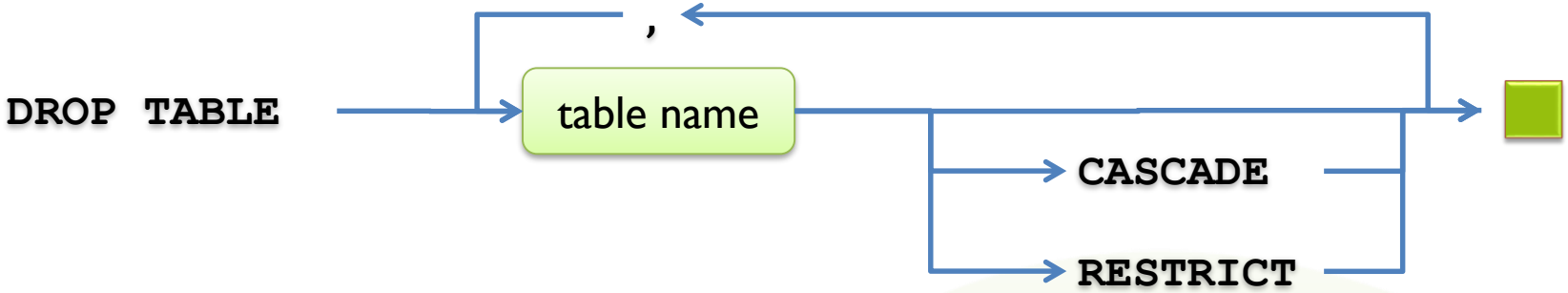
composed primary key

9.1 SQL DDL: Table Example

```
CREATE TABLE has_hobby(
        member INTEGER NOT NULL,
        hobby INTEGER NOT NULL,
        hours_per_week INTEGER NOT NULL,
        PRIMARY KEY (member, hobby),
        FOREIGN KEY (member) REFERENCES member
            ON DELETE CASCADE
            ON UPDATE CASCADE,
        FOREIGN KEY (hobby) REFERENCES hobby
            ON DELETE CASCADE
            ON UPDATE CASCADE
)
```

9.1 SQL DDL: Drop Tables

- For **deleting** tables, there is the **`DROP TABLE`** command



  - if **`RESTRICT`** is used, you may only drop empty tables that are not referenced by any other table
  - if **`CASCADE`** is used, all referencing tables are also deleted (including all stored rows)
  - if neither is used, the table does not have to be empty, but must not be referenced by another one
  - example
    - `DROP TABLE member CASCADE, hobby CASCADE`

9.1 SQL DDL: Alter Tables

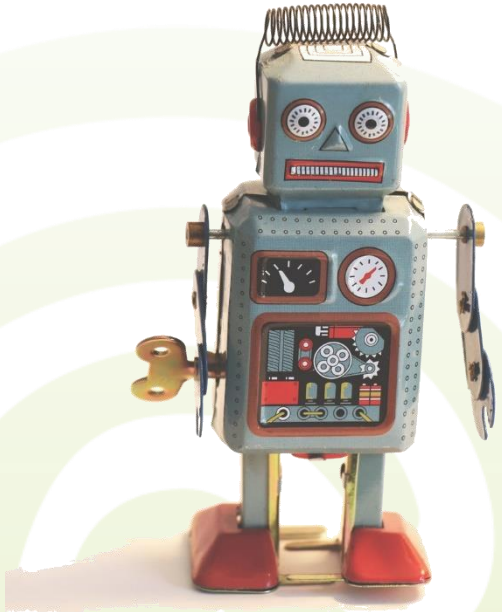
- After a table has been created, you may **alter** it by adding/removing **columns** or **constraints**

9.1 SQL DDL: Alter Tables

- if you add a **new column** with a **`NOT NULL`** constraint, you also need to provide a **default value**
- when **dropping** a **column,** you must either choose
  - **`CASCADE`** to also delete any views, indexes, and constraints dependent on that column
  - **`RESTRICT`** to allow the drop only if there is no referring column (default)
- if the **name** of a constraint is **auto-generated**, you need to look it up in the **system catalog**
- example
  - ```
    ALTER TABLE has_hobby DROP hours_per_week
    ALTER TABLE has_hobby
          ADD COLUMN since DATE
    ```
    ```

9 SQL 2

- SQL data definition language
- **SQL data manipulation language (apart from `SELECT`)**
- SQL ≠ SQL
- Some advanced SQL concepts

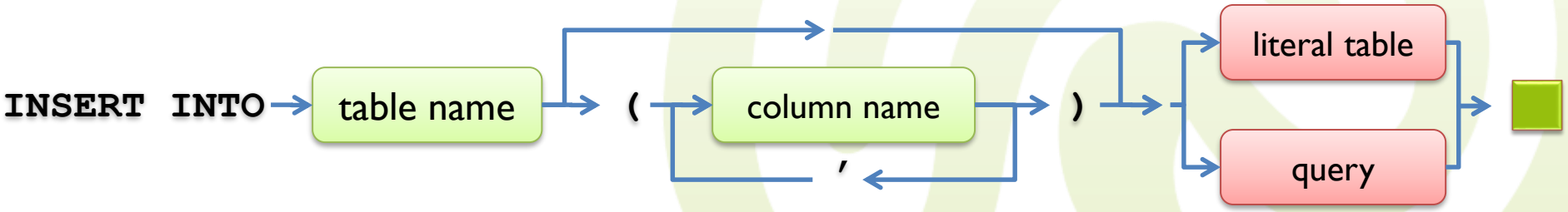


9.2 SQL DML

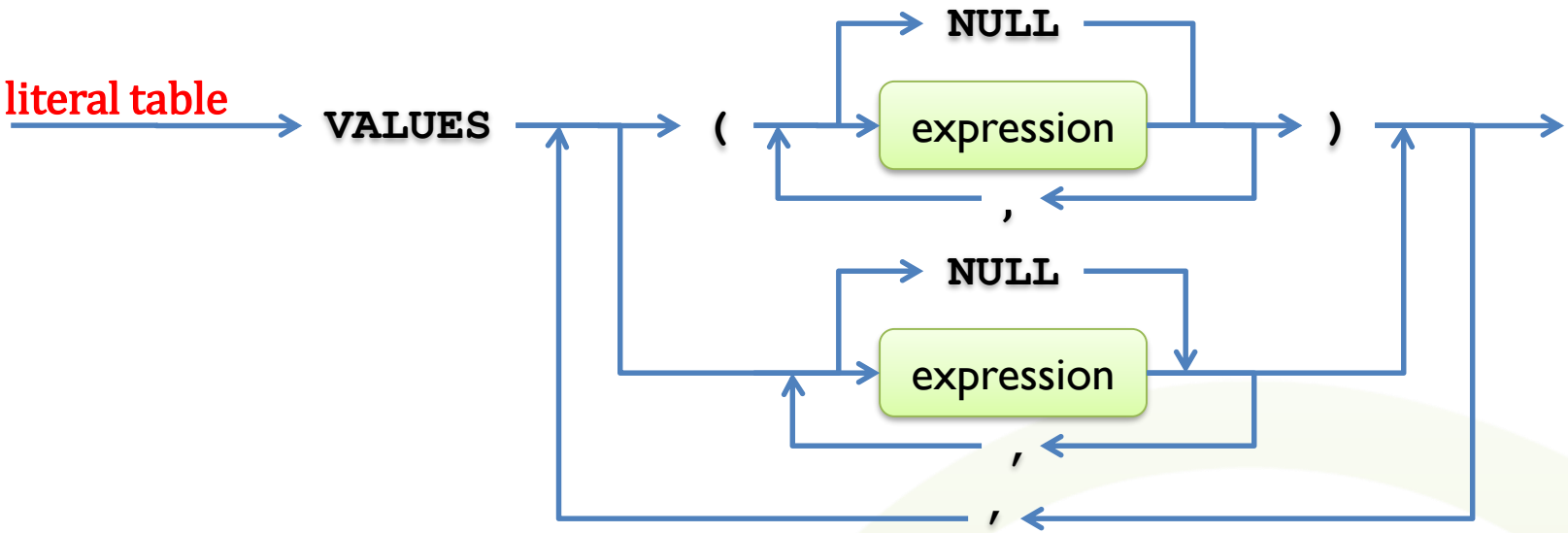
- **Data definition language** (DDL)
  - creating, changing, altering schemas, tables, ...
    - `CREATE SCHEMA`
    - `CREATE TABLE`
    - `ALTER TABLE`
    - `DROP TABLE`
- **Data manipulation language** (DML)
  - querying
    - `SELECT`
  - adding and updating data
    - `INSERT INTO`
    - `UPDATE`
    - `DELETE`

9.2 SQL DML: Insert

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



9.2 SQL DML: Insert



- A **literal table** is defined extensionally:

```
VALUES ('Artemis', 'Towel')

VALUES ('Karl', 'Sparks'), ('Hannah', 'Antenna')

VALUES 'Sokrates', ('Professor Y'), 'Asteriks'
```

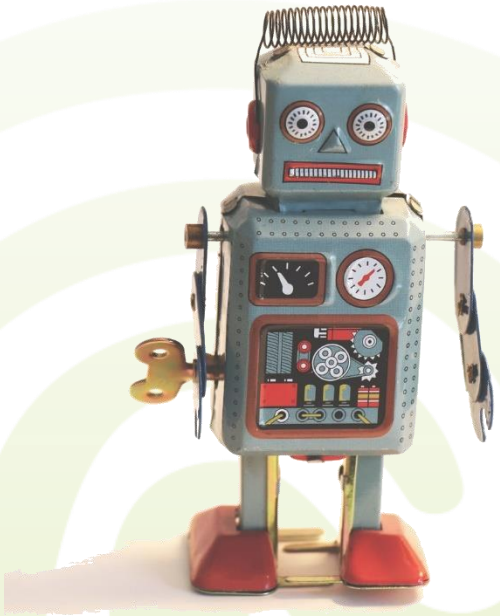
Artemis	Towel

Karl	Sparks
Hannah	Antenna

Sokrates
Professor Y
Asteriks

9.2 SQL DML: Insert

```
INSERT INTO member(id, real_name) VALUES
     (1, 'Sammy Riegel'),
     (2, 'Morris Hinz')
INSERT INTO has_nickname VALUES
     (1, 'FCG'),
     (1, 'Letters'),
     (2, 'Moritz'),
     (2, 'Boris')
```



9.2 SQL DML: Insert

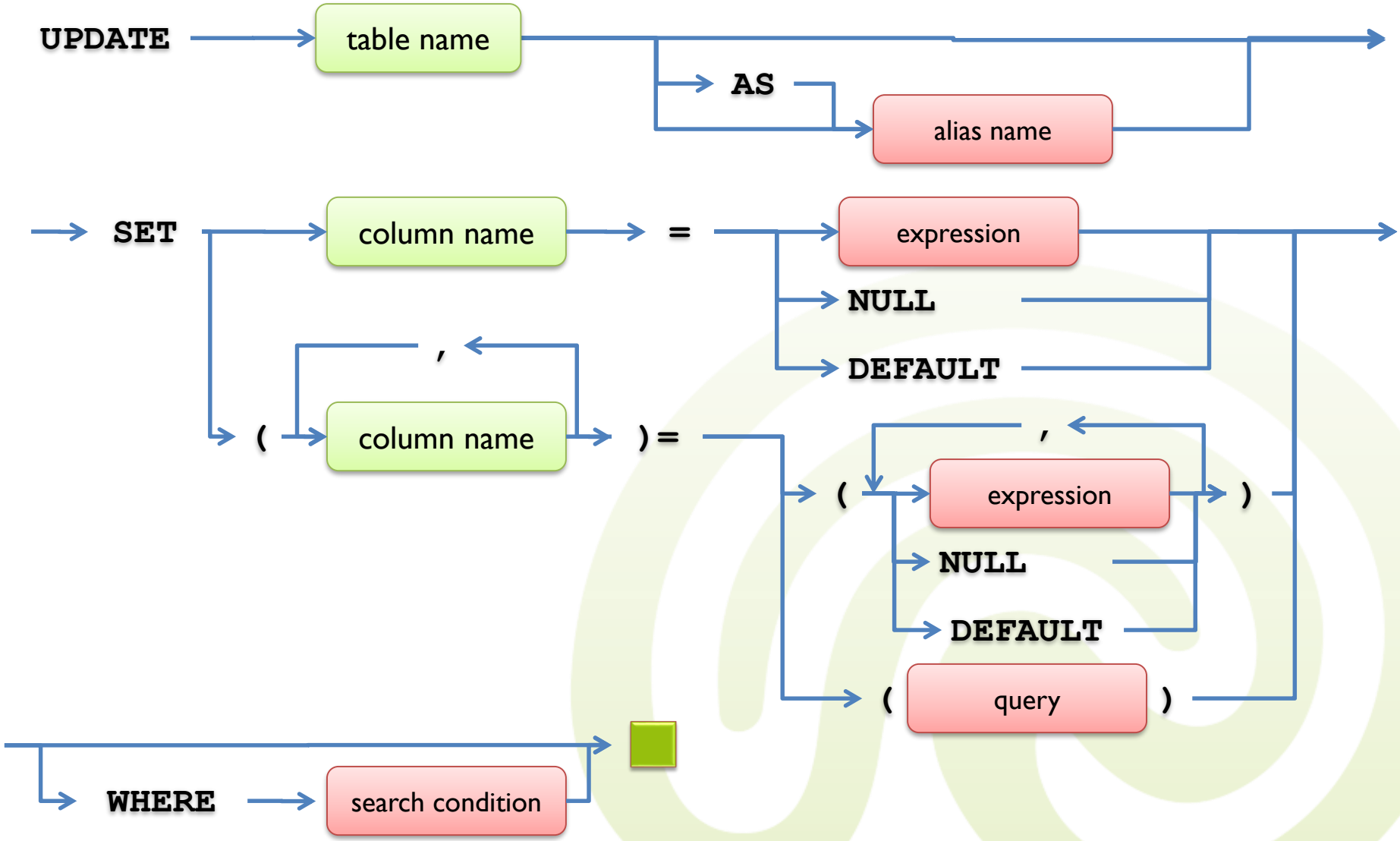
- Of course, subqueries may also be used in **INSERT** statements
  - ```
    INSERT INTO members_starting_with_a(
        SELECT * FROM member
        WHERE name LIKE 'A%'
    )
    ```


    ```

9.2 SQL DML: Update

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

9.2 SQL DML: Update

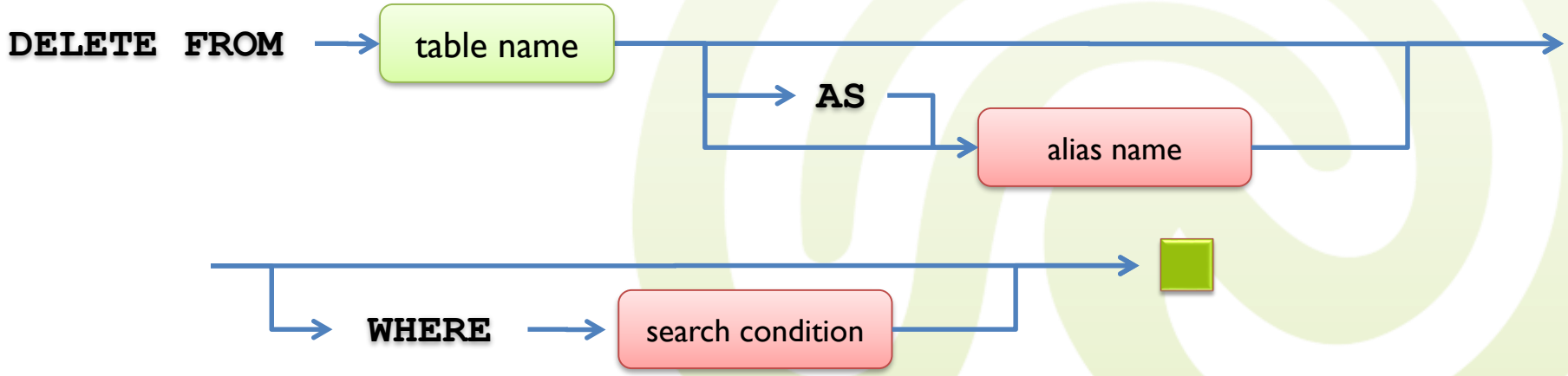


9.2 SQL DML: Update

- Replace the name of each member with NULL
  - `UPDATE member SET name = NULL`
- Multiply all hours_per_week values by 10
  - ```
    UPDATE has_hobby
        SET hours_per_week = hours_per_week * 10
    ```
- Change the name of the member with id 1
  - ```
    UPDATE member
        SET name= 'John of Warringham'
    WHERE id = 1
    ```
- Change name and id of Rick Chances
  - ```
    UPDATE member
        SET (id, name) = ('3', 'Pickle Rick')
    WHERE name = 'Rick Chances'
    ```
  - Change of id is propagated to other tables when `ON UPDATE CASCADE` is used in table definition
- Again, subqueries can be used in the **`WHERE`** clause
    ```

9.2 SQL DML: Delete

- The **`DELETE` statement** is used to delete rows from a table
  - deletes all rows satisfying a certain search condition
  - example
    - delete *Pickle Rick*
      - `DELETE FROM member WHERE name = 'Pickle Rick'`
    - delete all members
      - `DELETE FROM member`



9.2 SQL DML: Delete

- Again, subqueries can be used here
  - ```
    DELETE FROM member m
    WHERE NOT EXISTS(
          SELECT * FROM has_hobby h
          WHERE h.member = m.id
    )
    ```


    ```

9 SQL 2

- SQL data definition language
- SQL data manipulation language (apart from `SELECT`)
- **SQL ≠ SQL**
- Some advanced SQL concepts

9.3 SQL ≠ SQL

Detour

- First, the **good news:**
  - SQL has been standardized by the ISO in 1987
  - the standard is well-maintained and under active development
    - SQL-86, SQL-89, **SQL-92, SQL:1999, SQL:2003,** SQL:2006, SQL:2008, SQL:2011, SQL:2016, SQL:2019, SQL:2023
  - many *big* database vendors participate in the standardization process
    - IBM, Postgres, Microsoft, Oracle, Sybase, ...

9.3 SQL ≠ SQL

Detour

A timeline of SQL standardization:

- 1986
  - ANSI SQL
  - relations, attributes, views
  - **`SELECT` ... `FROM` ... `WHERE` ...**
- 1987
  - SQL-86 (ISO/IEC 9075:1986)
- 1989
  - SQL-89 (SQL1)
    - ≈ SQL-86 + restricted referential integrity

9.3 SQL ≠ SQL

Detour

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

9.3 SQL ≠ SQL

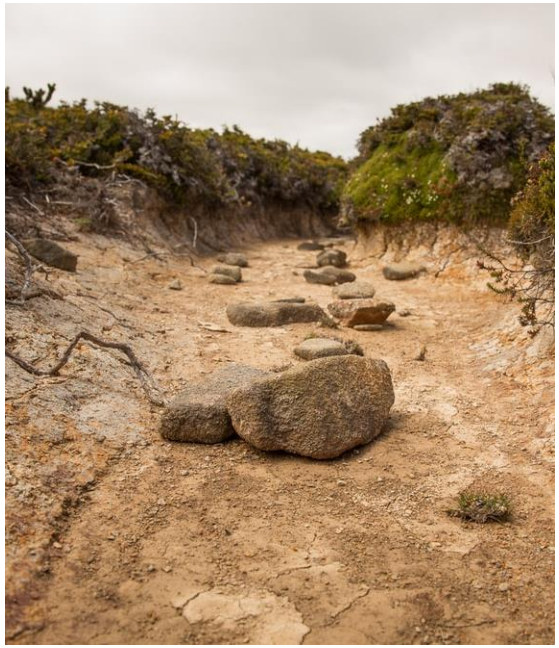
Detour

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

9.3 SQL ≠ SQL

Detour

- Recursive queries (SQL:1999):
  - *How to find all rivers that flow into the North Sea?*
  - Multiple joins?



flows_into	
river	mouth
Oker	Aller
Aller	Weser
Weser	North Sea
Elbe	North Sea
Edder	Wietze
Flöth	Wietze
Wietze	Aller
Isar	Danube
Inn	Danube

9.3 SQL ≠ SQL

Detour

- Solution

```
WITH RECURSIVE flows_into_ns(river) AS (
  SELECT river
  FROM flows_into
  WHERE mouth = 'North Sea'

  UNION

  SELECT fi.river
  FROM flows_into AS fi
    JOIN flows_into_ns AS fins
      ON fi.mouth = fins.river
)
SELECT river FROM flows_into_ns
```

9.3 SQL ≠ SQL

Detour

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

9.3 SQL ≠ SQL

Detour

- 2006
  - SQL:2006
  - successor of SQL:2003
  - a new extension for XML handling
    - importing, storing, querying, and manipulating XML data
    - support for XQuery
    - concurrently access (object-)relational data and XML data
- 2008
  - SQL:2008
  - successor of SQL:2006
  - *maintenance release* (some new but minor features)

9.3 SQL ≠ SQL

Detour

- 2011
  - SQL:2011
  - successor of SQL:2008
  - adds support for **time periods**

```
CREATE TABLE mutation(
        name VARCHAR(255) PRIMARY KEY NOT NULL,
        person INTEGER REFERENCES person(id),
        from DATE,
        until DATE,
        PERIOD FOR period_of_validity (from, until));
SELECT name, person FROM mutation
WHERE period_of_validity CONTAINS '2014-12-19';
```

  - temporal predicates, temporal primary keys, temporal referential integrity
    - This might be very useful for work planning or scheduling applications

9.3 SQL ≠ SQL

Detour

- Well, here are the **bad news**
  - there are still **too many variants** of SQL (both **syntactic and semantic differences**)
    - true application portability remains a challenge
  - the standard has been used to introduce **two kinds of features**
    1. features that are well-understood and widely implemented
    2. new and largely untried technologies, hoping that vendors follow the lead and deliver new functionalities
  - **vendors don't care** too much about the standard

9.3 SQL ≠ SQL

Detour

- A **common myth** among software developers

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

- If you don't believe me, here are some examples ...

9.3 SQL ≠ SQL

Detour

- ```
  CREATE TABLE name (
      first VARCHAR(100),
      middle VARCHAR(100),
      last VARCHAR(100)
  )

  INSERT INTO name VALUES ('George', 'Walker', 'Bush')
  INSERT INTO name VALUES ('Horst', '', 'Kr')
  INSERT INTO name VALUES ('Angela', NULL, 'Merkel')
  ```
  - `''` (empty string) means that we know that there is no middle name
  - **NULL** means that we don't know whether there is a middle name
- Sounds like a good design? What do you think?
  - according to the SQL standard, this approach is fine ...
  - ... unless your RDBMS is Oracle (`''` is the same as **NULL**)
  ```

9.3 SQL ≠ SQL

Detour

- What about terminology?
  - the SQL standard defines the following notions
    - Environment
    - Cluster
    - Catalog
    - Schema
  - the reality
    - Database server
    - (unsupported)
    - Database
    - Schema
  - but attention
    - in MySQL, there are no catalogs, *schema* and *database* are synonyms
    - in Oracle, there is exactly one schema per user.
      `CREATE/ALTER SCHEMA` x <command> executes <command> on all objects located in schema x

9.3 SQL ≠ SQL

Detour

- The **statement terminator** ;
  - according to the SQL standard, (almost) every SQL statement has to be terminated by a semicolon
  - *What's happening in practice?*
    - many RDBMS treat the terminator as being optional (which is fine, but may cause some problems)
      - PostgreSQL does not require semicolons for single commands, but multiple commands must be separated by semicolons
    - some RDBMS either strictly require a terminator or complain if it is present
    - in some RDBMS, this behavior can be configured ...
  - summary:
    No matter what you do, it causes problems!

9.3 SQL ≠ SQL

Detour

- The **BOOLEAN data type**
- ```
  CREATE TABLE customers (
      id INTEGER PRIMARY KEY,
      name VARCHAR(100),
      is_vip BOOLEAN,
      is_blacklisted BOOLEAN
  )

  SELECT id, name FROM customers
  WHERE is_vip AND NOT is_blacklisted
  ```
  - practice?
    - not supported by DB2, and MS SQL Server
      - official workarounds: use **CHAR** or **INTEGER** …
    - supported by Oracle, MySQL and PostgreSQL
      - where in MySQL **BOOLEAN** is just a short hand for **TINYINT(1)** …


  ```

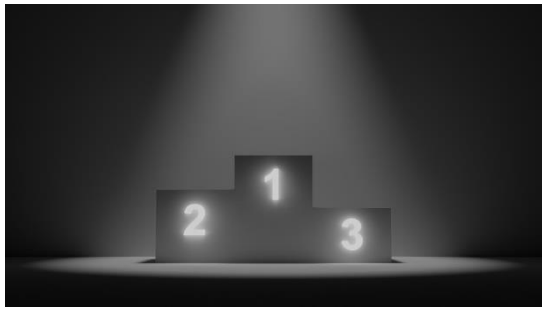
9.3 SQL ≠ SQL

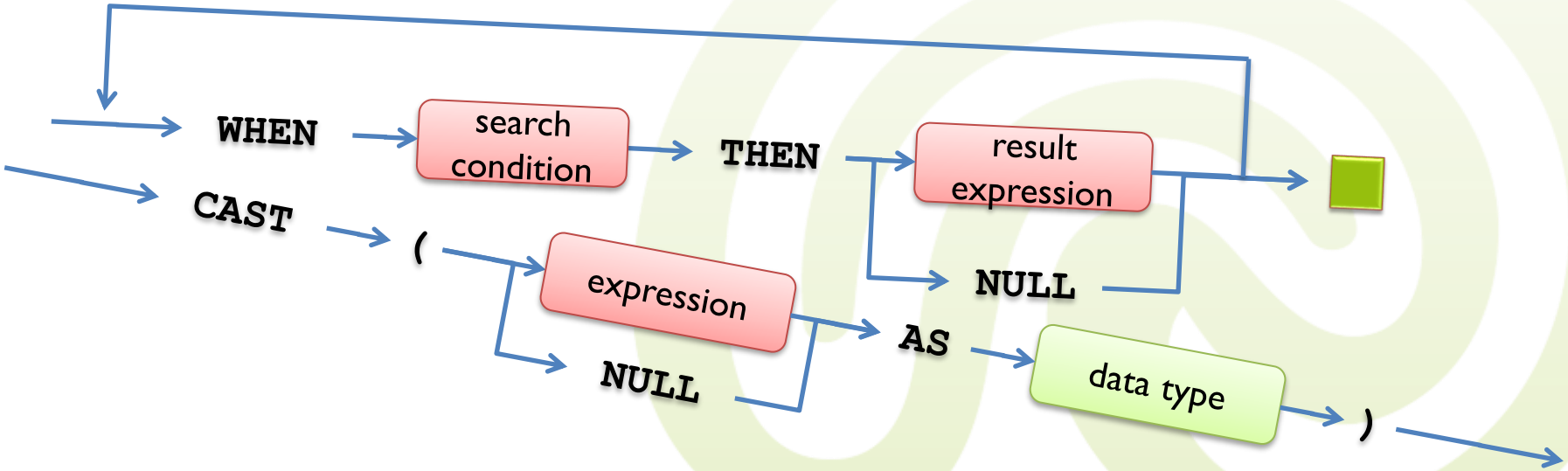
Detour

- **Summary**
  - **SQL is not SQL**
  - in some cases, even **identical SQL** statements **work differently** on different RDBMS
- Current trends?
  - open-source RDBMS (PostgreSQL, MySQL, Derby, ...) typically try to adhere to the standard
    - however, many advanced features are not supported yet
    - new features in PostgreSQL 14:
      - parallel query execution on remote databases
      - query pipelining: send multiple queries without waiting for responses
      - …

9 SQL 2

- SQL data definition language
- SQL data manipulation language (apart from **`SELECT`**)
- SQL ≠ SQL
- **Some advanced SQL concepts**

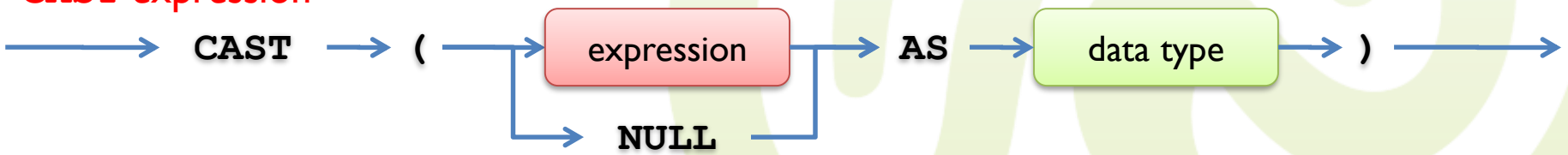




9.4 Type Casting

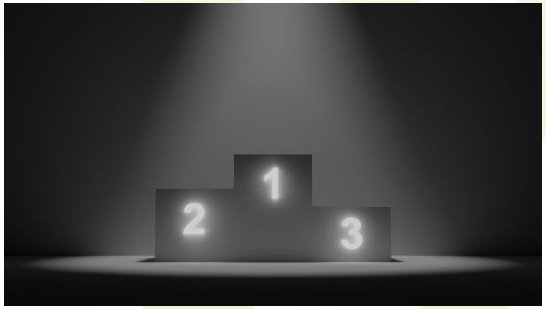
- SQL is a **strongly typed** language
  - Basically, this means that e.g. **`INTEGER`** is different from **`VARCHAR(100)`**
- If data types are incompatible, **type casting** may be used to make them compatible
  - **`CAST`** expression
  - during casting, precision may be lost (e.g. **`FLOAT`** → **`INTEGER`**)
  - example
    - ```
      CAST (hours_per_week AS NUMERIC(3, 2))
      CAST (nickname || name AS CHAR(255))
      ```
  - List possible type castings in **psql** with `\dCS <type>`

`CAST` expression


      ```

9.4 Ranking Functions

- Since SQL:2003, there are special functions for working with **result lists**
- Examples
  - output only every other row of the list
  - create a ranking with explicit ranks (1, 2, 3, ...)
  - on what rank position is some given row?

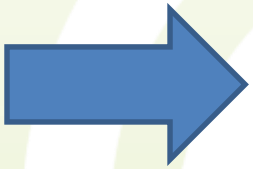


9.4 Ranking Functions

- **ROW_NUMBER()** returns the **position** of each row in the result list
- Example

```
SELECT name, salary,
  ROW_NUMBER() OVER (
    ORDER BY salary DESC
  ) AS pos
FROM person
```

person	name	salary
	Simon	45000
	Wolf-Tilo	75000
	Larry	200000000
	Christoph	45000



name	salary	pos
Larry	200000000	1
Wolf-Tilo	75000	2
Christoph	45000	3
Simon	45000	4

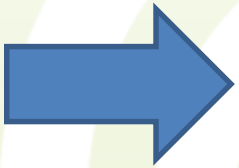
Depending on the implementation, the last two rows may switch positions

9.4 Ranking Functions

– **example:** *At which position is Wolf-Tilo?*

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

person	name	salary
	Simon	45000
	Wolf-Tilo	75000
	Larry	200000000
	Christoph	45000



name	salary	pos
Wolf-Tilo	75000	2

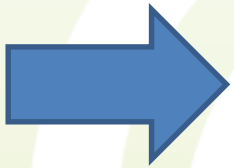
9.4 Ranking Functions

– **example:** *Show only rows at even positions.*

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

modulo

person	name	salary
	Simon	45000
	Wolf-Tilo	75000
	Larry	200000000
	Christoph	45000



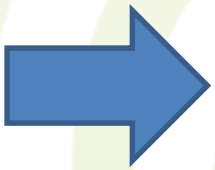
name	salary	pos
Wolf-Tilo	75000	2
Simon	45000	4

9.4 Ranking Functions

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

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

person	name	salary
	Simon	45000
	Wolf-Tilo	75000
	Larry	200000000
	Christoph	45000



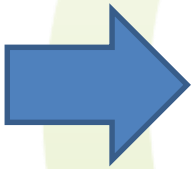
name	salary	rank
Larry	200000000	1
Wolf-Tilo	75000	2
Christoph	45000	3
Simon	45000	3

9.4 Ranking Functions

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

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

person	name	salary
	Simon	45000
	Wolf-Tilo	75000
	Larry	200000000
	Christoph	45000



name	salary	rank	drank
Simon	45000	1	1
Christoph	45000	1	1
Wolf-Tilo	75000	3	2
Larry	200000000	4	3

9.4 CASE Expressions

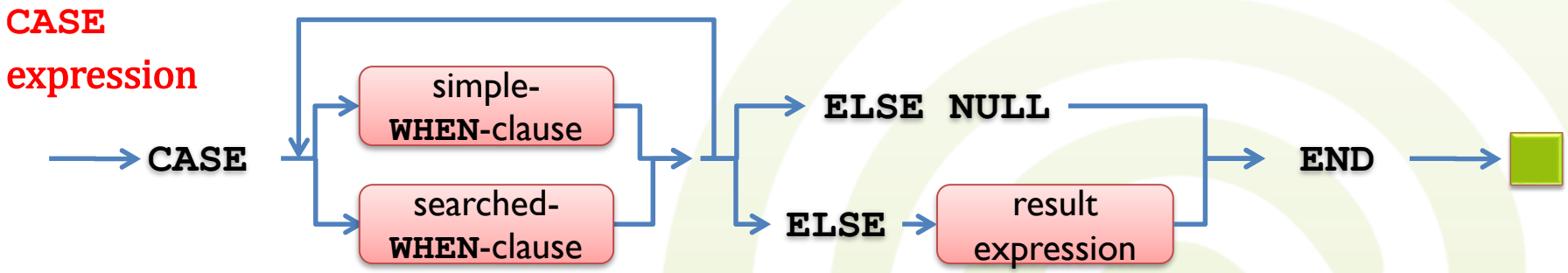
- Very often **codes** are used for storing more complex information
  - retrieving the account information for owner *Shanks* with appropriate account descriptions needs a join
  - Indicate all customers with a negative balance with the string *not creditworthy* in the query result

account	owner	balance	type
	Shanks	367,00	0
	Dido	-675,00	0
	Shanks	54987,00	1

acc_type	type	description
	0	checking account
	1	savings account
	2	credit card account

9.4 CASE Expressions

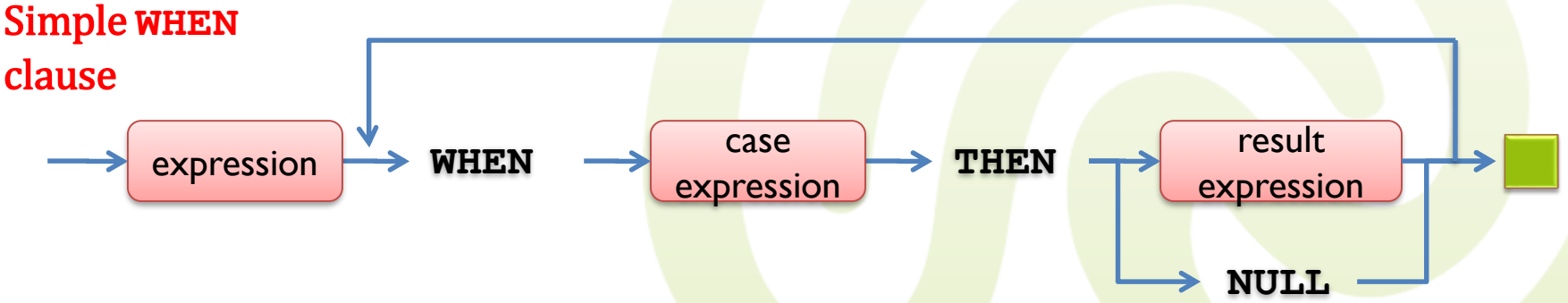
- The **CASE expression** allows a value to be selected based on the evaluation of one or more conditions (similar to *if-then-else*)
  - comes in two flavors



9.4 CASE Expressions

– the **simple `WHEN` clause**

- **compares an expression to each case expression** one by one
  - if expression is equal to search value, the corresponding result expression is returned
- if **no match is found,** then some default (**`ELSE`** clause) is returned
  - if **`ELSE`** is omitted, then **`NULL`** is returned

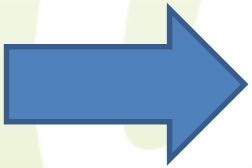


9.4 CASE Expressions

- **Example:** simple **WHEN** clause
  - directly decode the account type

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

account	owner	balance	type
	Shanks	367,00	0
	Dido	-675,00	0
	Shanks	54987,00	1

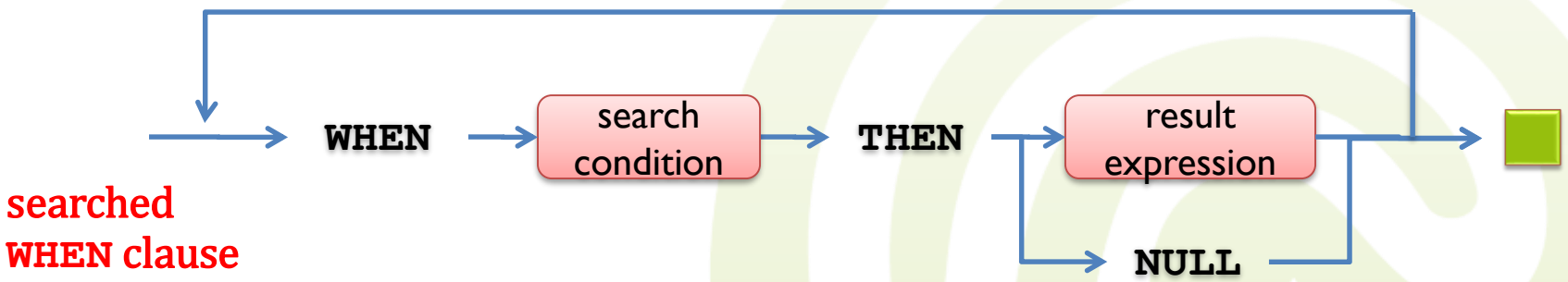


owner	verbose_type
Shanks	checking account
Dido	checking account
Shanks	savings account

9.4 CASE Expressions

– the **searched `WHEN` clause**

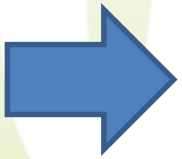
- checks search conditions from left to right
- **stops** as soon as a **search condition evaluates to true**
  - returns the corresponding result then
- if **no condition is true**, the value given by the **`ELSE`** clause is returned (or **`NULL`**, if there is no **`ELSE`** clause)



9.4 CASE Expressions

- **Example:** searched **WHEN** clause
  - retrieve credit rating of customers
  -
```
SELECT owner,
  CASE
    WHEN balance < 0 THEN 'not credit-worthy'
    WHEN balance = 0 THEN 'questionable'
    ELSE 'credit-worthy'
  END AS credit_worthyness
FROM account
WHERE type = 0
```

account	owner	balance	type
	Shanks	367,00	0
	Dido	-675,00	0
	Shanks	54987,00	1



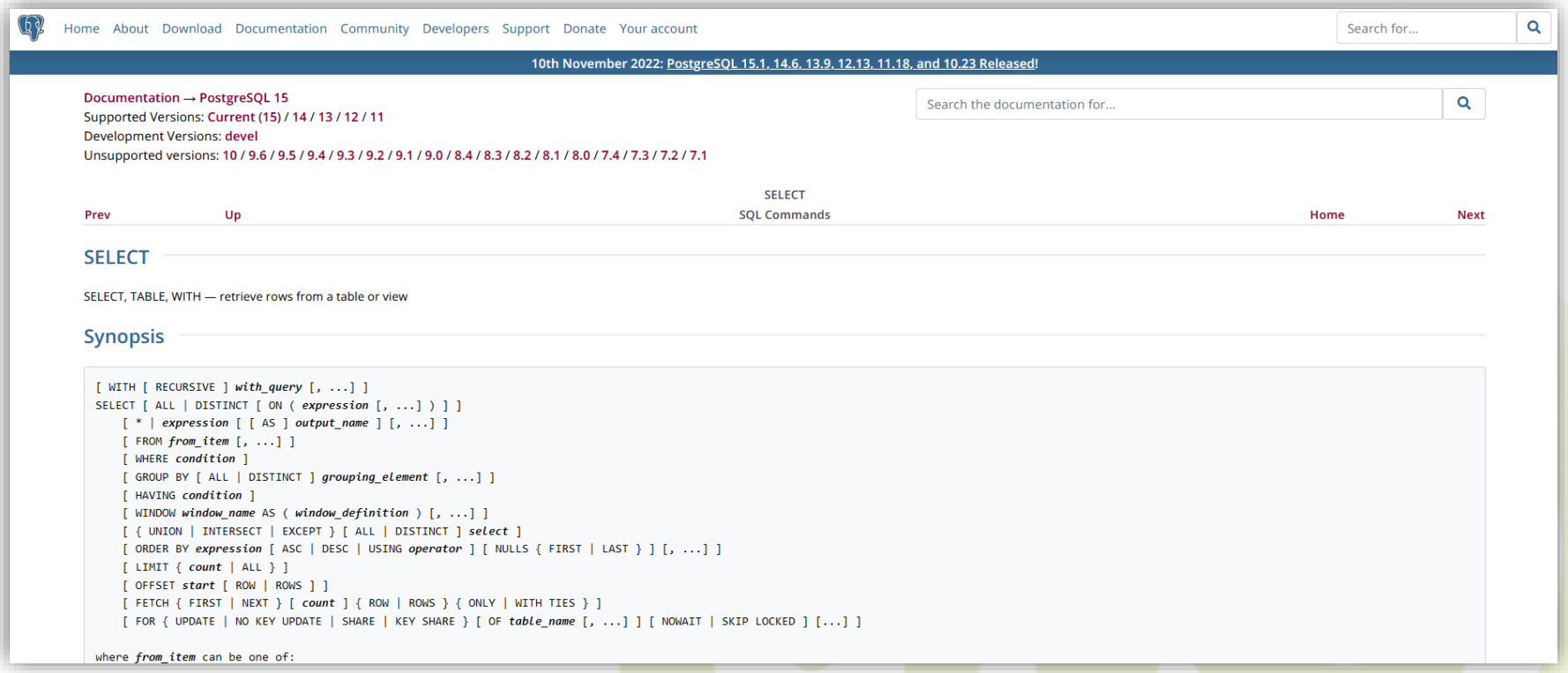
owner	credit_worthyness
Shanks	credit-worthy
Dido	not credit-worthy

9 More on SQL

- There are **many more SQL** statements than we are covering in our lectures
- Moreover, there are many different SQL dialects
- If you don't want to get mad, do the following
  - don't care too much about the SQL standard (unless you are actually implementing an RDBMS)
  - read the **SQL manuals** of **your** RDBMS!

9 More on SQL

- **Example:** PostgreSQL reference (version 14)
  - https://www.postgresql.org/docs/14/reference.html



Home About Download Documentation Community Developers Support Donate Your account

Search for...

10th November 2022: PostgreSQL 15.1, 14.6, 13.9, 12.13, 11.18, and 10.23 Released!

Documentation → PostgreSQL 15
Supported Versions: Current (15) / 14 / 13 / 12 / 11
Development Versions: devel
Unsupported versions: 10 / 9.6 / 9.5 / 9.4 / 9.3 / 9.2 / 9.1 / 9.0 / 8.4 / 8.3 / 8.2 / 8.1 / 8.0 / 7.4 / 7.3 / 7.2 / 7.1

Search the documentation for...

SELECT

Prev Up SQL Commands Home Next

SELECT

SELECT, TABLE, WITH — retrieve rows from a table or view

Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY [ ALL | DISTINCT ] grouping_element [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES } ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ] [ NOWAIT | SKIP LOCKED ] [...] ]

where from_item can be one of:
```

9 Next Lecture

- Normalization
- Functional Dependencies
- Normal Forms
  - 1NF, 2NF, 3NF, BCNF, 4NF, 5NF, 6NF