CMPUT391

SQL: Review and Recursion

Instructor: Denilson Barbosa

University of Alberta

October 12, 2023

Slides by D. Barbosa, with suggested corrections and improvements by (in alphabetical order) C. Bins, D. Caminhas, K. Guhzva, Q. Lautischer, E. Macdonald, M. A. Nascimento, K. Newbury, M. Strobl, D. Sunderman, K. Wang, and K. Wong.

Enforcing constraints

Complex Constraints

Basic SQL

Syntax and Semantics

Set/Bag Operations

Negation and Quantification

Common Table Expressions

Recursive Queries

The relational model of data

Relational databases were introduced in the 1970's, became prevalent in the 1980's and remain the industry standard for applications that manage large volumes of data.

A relational database organizes the application data into relations (a.k.a. tables), which are manipulated (queries and updates) using a relational query language.

A relational database management system (DBMS) is a set of computer programs that provides storage and access to a relational database to multiple users and applications, often concurrently.

Relations and Tables

These concepts are nearly identical, but there are subtle differences between them.

Relations were introduced in the original, theoretical, relational model of data.

A relation is a **set** of tuples, and as such, does not contain duplicates.

Tables are the implementation of the (theoretical) relation in an actual system.

For simplicity, most systems do not remove duplicates from a table by default.

We will refer to relations and tables interchangeably in these notes. Unless otherwise specified, you can assume we are referring to the concept of a “table”, discussing duplicates explicitly when needed.

Relational Languages

In CMPUT291/391 we study two relational languages:

- the **algebra**, which is clean and grounded on solid mathematical foundations
- **SQL**, which is a very complex language with many nuances and intricacies introduced to make programmers lives easier

The algebra and SQL are good friends

While the algebra is procedural and SQL is declarative they are “largely equivalent”. In fact, most DBMSs translate SQL queries into algebraic expressions and evaluate them that way.

The Structured Query Language (SQL)

Major contributor to the success of the relational model.

SQL has two main parts:

- the **Data Definition Language** (DDL) is used to define and modify the schema[1] (e.g., `CREATE TABLE` statements);
- the **Data Manipulation Language** (DML), used to write queries and update statements.[2]

Although SQL is an **ISO standard**, DBMS vendors support *most* of the language, adding their own extensions and modifications.
- check out SQLite's support: `https://sqlite.org/lang.html`

[1] `https://en.wikipedia.org/wiki/Data_definition_language`
[2] `https://en.wikipedia.org/wiki/Data_manipulation_language`

Relation (table)

A relation (table) is a **collection of** similar **facts** relevant to the application.

- **schema**: relation name, attribute names and types, constraints;
- **instance**: tuples (also called rows).

```
CREATE TABLE Movie (title CHAR(20), year INT, imdb FLOAT,
   director CHAR(20), PRIMARY KEY (title, year),
   CHECK(imdb >= 0 AND imdb <= 10));
```

Movie

title	year	imdb	director
Ghostbusters	1984	7.8	Ivan Reitman
Big	1988	7.3	Penny Marshall
Lost in Translation	2003	7.8	Sofia Coppola
Wadjda	2012	8.1	Haifaa al-Mansour
Ghostbusters	2016	5.3	Paul Feig

Relational Database

A database is a **collection of** relations (tables), each with its own schema and set of constraints.

- **schema**: the schemas of the relations *plus* (optionally) inter-relation constraints;
- **instance**: the instances of the relations.

Database constraint

Logical condition over one or more tuples, derived from the application requirements, that must be true at all times.

Examples:

- No two movies from the same year can have the same title.
- The IMDB score of a movie must be between 0 and 10.
- Every actor must be associated with a movie.

Constraints can be defined over a single table or involve multiple tables.

- Single table constraint:
    - No two movies from the same year can have the same title.
- Multi-table constraint:
    - Every actor must be associated with a movie.

Constraints can involve a single tuple or multiple tuples:

- Single-tuple:
    - The IMDB score of a movie must be between 0 and 10.
- Multiple tuples:
    - Movies can have at most 10 actors in the database.

By defining constraints directly *in the database*[3], the DBMS ensures that the data satisfies the constraints at all times, regardless of who is accessing the database.

Legal database instance

A database instance is **legal** if it satisfies all constraints in the schema.

With the constraints defined in the database, any insertion of new data, deletion, and modification to existing data will be checked against the set of constraints.

[3] As opposed to enforcing them in the client-side of the application.

Most common SQL constraint enforcement tools:

(1) `PRIMARY KEY`: specifies attributes that cannot be null and must uniquely identify each tuple.
(2) `UNIQUE`: specifies attributes whose values cannot appear in multiple tuples.
(3) `CHECK`: specifies the values that are allowed for an attribute (i.e., its domain).
(4) `NOT NULL`: specifies attributes that are required.
(5) `FOREIGN KEY`: specifies attributes in a table that "depend" on (i.e., must match some key) a tuple from another.

Example database schema:

```
CREATE TABLE Movie (title CHAR(20), year INT, imdb FLOAT,
   director CHAR(20) NOT NULL, PRIMARY KEY (title, year),
   CHECK(imdb >= 0 AND imdb <= 10));

CREATE TABLE Cinema (name CHAR(20), address CHAR(100),
   PRIMARY KEY(name));

CREATE TABLE Cast (title CHAR(20), year INT,
   actor CHAR(20) NOT NULL, role CHAR(20) NOT NULL,
   FOREIGN KEY (title, year) REFERENCES Movie(title, year));

CREATE TABLE Guide (theater CHAR(20), film CHAR(20), year INT, start INT,
  PRIMARY KEY (theater, film, year, start),
  FOREIGN KEY (theater) REFERENCES Cinema(name),
  FOREIGN KEY (film, year) REFERENCES Movie(title, year));
```

Movie

title	year	imdb	director
Ghostbusters	1984	7.8	Ivan Reitman
Big	1988	7.3	Penny Marshall
Lost in Translation	2003	7.8	Sofia Coppola
Wadjda	2012	8.1	Haifaa al-Mansour
Ghostbusters	2016	5.3	Paul Feig

Cast

title	year	actor	role
Ghostbusters	1984	Bill Murray	Dr. Venkman
Ghostbusters	1984	Sigourney Weaver	Dana Barret
Big	1988	Tom Hanks	Josh
Big	1988	Elisabeth Perkins	Susan
Lost in Translation	2003	Bill Murray	Bob Harris
Wadjda	2012	Waad Mohammed	Wadjda
Ghostbusters	2016	Dan Aykroyd	Cabbie
Ghostbusters	2016	Sigourney Weaver	Dana Barret

Cinema

name	address
Garneau	8712 109 St, Edmonton
Princess	10337 82 Ave, Edmonton
Landmark	10200 102 Ave, Edmonton

Guide

theater	film	year	start
Garneau	Ghostbusters	1984	1140
Garneau	Ghostbusters	2016	1290
Princess	Wadjda	2012	1260

Enforcing constraints

Trivial SQL constraints

`NOT NULL` and `CHECK` constraints are trivial to check, in $O(1)$ time, because they concern individual tuples only.

For example, consider the domain constraint:

```
CREATE TABLE Movie (title CHAR(20), year INT, imdb FLOAT,
   director CHAR(20), PRIMARY KEY (title, year),
   CHECK(imdb >= 0 AND imdb <= 10));
```

Insertions: every time we try to insert a new tuple, the DBMS can verify the `imdb` in the `new` tuple is a number between 0 and 10, and prevent the insertion if that is not the case.

Updates: similarly, every time we try to update a tuple already in the database, the DBMS can verify if the `new` value for `imdb` is legal.

Other Simple SQL Constraints

`PRIMARY KEY` constraints can be easily enforced with the help of appropriate data structures internal to the DBMS.

Enforcing keys with indexes

Assume the DBMS maintains an index (e.g., a B+ tree) with every primary key in the table.

- **Insertions**: check if the primary key of the `new` tuple is already in the index; if so, prevent the insertion. If not, insert the tuple and add the new key to the index.
- **Deletions**: delete the tuple and its key from the index.
- **Updates**: attempt to insert the the `new` tuple, if possible, delete the the `old` one.

All operations above can be done within $O(\log n)$ accesses to the database, where n is the number of tuples (i.e., index keys).

Enforcing keys on linked lists?

Without and index, we need to scan every tuple to make sure the `new` key is not present in the table, which would cost $O(n)$ accesses to the database – way too expensive.

Because primary keys are so common, a successful DBMS will **always** create indexes to support primary keys.

Enforcing `UNIQUE` constraints is essentially the same as enforcing primary keys.

`FOREIGN KEY` Constraints

Foreign keys state a “dependency” between tuples.

Example: “Every actor (i.e., tuple in `Cast`) must be associated with a movie (i.e., tuple in `Movie`).

```
CREATE TABLE Movie (title CHAR(20), year INT, imdb FLOAT,
   director CHAR(20), PRIMARY KEY (title, year),
   CHECK(imdb >= 0 AND imdb <= 10));

CREATE TABLE Cast (title CHAR(20), year INT,
   actor CHAR(20), role CHAR(20),
   FOREIGN KEY (title, year) REFERENCES Movie(title, year);
```

Every tuple in `Cast` depends on the existence of a tuple in `Movie` with the corresponding `title, year` combination!

Inclusion Dependency

A **foreign key** is actually a special case of a more general kind of constraint involving two tables, called “inclusion dependencies”:

Inclusion dependency

Inter-relation constraint, requiring that the values of some attributes in a source relation S exist in a target relation T:

$$S(a_1, \ldots, a_n) \subseteq T(b_1, \ldots, b_n)$$

A foreign key is an inclusion dependency whose target $T(b_1, \ldots, b_n)$ is itself a primary key.

Enforcing Inclusion Dependencies

Enforcing $S(a_1, \ldots, a_n) \subseteq T(b_1, \ldots, b_n)$ means:

(1) We can only **insert** a **new** tuple in S if we find a matching tuple in T.

(2) Similarly, we can only update a tuple in S if we can find a matching tuple in T.

What happens we we try to delete or update a **tuple in** T that matches one or more tuples in S?

- SQL allows the DBA to choose what to do, again, for foreign keys only

```
ON [DELETE | UPDATE] CASCADE | SET NULL | SET DEFAULT
```

How can the DBMS efficiently enforce foreign keys?

- Use an index to quickly finding matching tuples in the *target* relation. Such an index will always exist, anyway as the attributes in the target relation need to be a primary key.
- Use an index on the *source* relation as well, so that the tuples affected by the `ON DELETE` and `ON UPDATE` statements can be found quickly.

Foreign Keys in SQLite

The committee who created the SQL standard decided that general inclusion dependencies were not common enough to be included in the SQL DDL language. Therefore, we have only support for foreign keys.

WARNING

By default, SQLite does not check for foreign keys violations!!!

`https://sqlite.org/foreignkeys.html`

To enable foreign key verification, type this on the prompt (or embed it in your program):

```
sqlite> PRAGMA foreign_keys = ON;
```

Takeaways:

- Data that violates constraints is bad data.
- Checking constraints in the application(s) can lead to duplicated code (and inconsistencies); it is better to **define the constraints once, in the database**.
- SQL allows simple constraints that need at most indexes to enforce:
  (1) `NOT NULL` and `CHECK` can be verified by looking at the tuple being inserted/updated in $O(1)$ time.
  (2) `UNIQUE`, `PRIMARY`/`FOREIGN KEY` constraints require indexes to be checked efficiently.

Complex Constraints

Often, applications have constraints that cannot be enforced with keys and domain constraints alone.

For example, in BearTracks, the number of students enrolled in a course cannot be higher than the number of seats in the classroom assigned to that course.

Event-based programming in SQL

Most DMBSs support some basic programming within SQL.

In this model, one or more `TRIGGER`s are used to assign code to check if a constraint is violated in response to insertions, deletions, and updates in the tables related to the constraint.

For example, we can write code to check, before inserting new students into a course, if the capacity of the room associated with that course has already been reached.

The “official” language for programming in SQL is called SQL/PSM (Persistent Stored Modules)

- `https://en.wikipedia.org/wiki/SQL/PSM`

Many DBMSs have their own programming language however

- `https://en.wikipedia.org/wiki/SQL#Procedural_extensions`

Some allow Turin-complete languages like Java (Oracle, PostgreSQL, ...)

Stored Procedures

Most DBMSs allow you to declare functions as “stored” procedures.
These functions can be used in subsequent SQL queries.
Stored procedures can have side-effects (e.g., modify other parts of the database) too.

`https://en.wikipedia.org/wiki/Stored_procedure`

Database *modifications* and constraints

Since queries do not change the data, we need only enforce constraints when the database is being modified.

The application developer needs to think about which **events** in the normal execution of the application need to trigger checking constraints.

Ex: preventing enrollment higher than classroom size:

- Deleting tuples from the enrollment table? Do nothing.
- Inserting into the enrollment table? Must check if the classroom size has room; if not, abort!
- Updating a tuple in the enrollment table? Must check classroom size as well!

Database *modifications* and constraints

Since queries do not change the data, we need only enforce constraints when the database is being modified. Every time we attempt to modify the database the DBMS must:

(1) Compute the `new` tuples resulting from the update, and the `old` tuples that are to be deleted by the update (if any).

(2) Check if inserting any new tuple or deleting any old tuple causes a constraint to be violated.

(3) Effect the changes only if no violation is found.

The code to check if a complex constraint is violated and to perform remedial actions (e.g., sending an email to someone) is declared into what is called an SQL `TRIGGER`

Kinds of TRIGGERS

SQL defines two kinds of triggers:

(1) **STATEMENT** triggers are run once per transaction (we will see more of that later)

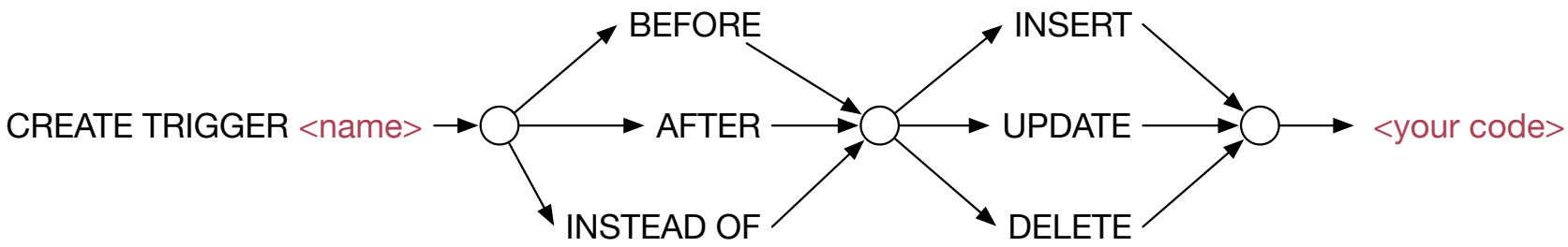
(2) **ROW** triggers are run once for each tuple affected by the transaction

Example:

```
INSERT INTO Cast(title,year,actor,role) VALUES
   ("Groundhog day",1993,"Bill Murray","Phil"),
   ("Groundhog day",1993,"Andie MacDowell","Rita");
```

The update above would *fire* a statement trigger once and a row trigger twice: once for each tuple being inserted.

Row-level Triggers



9 possible “events”:

- **`BEFORE INSERT`**: the code is executed before the new tuple is added;
- **`BEFORE UPDATE`**: the code is executed before the tuple is changed;
- ...

Triggers in SQLite

`https://sqlite.org/lang_createtrigger.html`

‘new’ and ‘old’ tuple variables

There are two system-defined tuple variables in a trigger:

`new`:

- in an INSERT statement is the brand new tuple
- in an UPDATE statement is the modified tuple

`old`:

- in an UPDATE statement is the tuple before the change
- in a DELETE statement is tuple that will be removed

Different systems have different names for these tuple variables. We follow SQLite here.

Read up: `https://sqlite.org/lang_createtrigger.html`

Example constraint: no movie can have more than 10 actors. Recall the foreign key `title,year` in `Cast` to the primary key in `Movie`.

```
CREATE TRIGGER Constraint_1
BEFORE INSERT ON Cast
WHEN 10 <= (SELECT COUNT(*) FROM Cast c
            WHERE new.title=c.title AND new.year=c.year)
BEGIN
    SELECT RAISE(ABORT, "Constraint #1 violated!");
END;
```

What is going on here?

The **WHEN** clause specifies the condition for the trigger to *fire*

```
CREATE TRIGGER Constraint_1
BEFORE INSERT ON Cast
WHEN 10 <= (SELECT COUNT(*) FROM Cast c
            WHERE new.title=c.title AND new.year=c.year)
BEGIN
    SELECT RAISE(ABORT, "Constraint #1 violated!");
END;
```

The condition is checked for every tuple being inserted.

The code between **BEGIN** ... **END**; is often called an *SQL procedure* and defines what to do if the trigger *fires* (in this case, abort the operation).

The **RAISE**(**ABORT**, "...") command creates a runtime error that becomes an exception inside your code (or an error message in the console).

BEFORE or AFTER?

Most of the time, we want to check the constraint BEFORE the update is completed, and therefore we use that kind of trigger.

Keep in mind that a trigger for AFTER an update runs after the database has already been modified, so we cannot use them to *prevent* constraint violations. If you want to do that, the trigger must reverse the update that caused the violation.

Instead, AFTER triggers are normally used for automating tasks.

- Ex: after an insertion into the “orders” table, if the inventory of an item is low, automatically purchase more items from the supplier.

`TRIGGER`s are not retroactive!

- When you add a trigger, the data already in the database is not checked for constraint violations! In our example, movies that had more than 10 actors at the time the `TRIGGER` is created will remain in the database.
- When you add a constraint to a database that already has data, you must run queries to check, manually for violations of that constraint and deal with them accordingly.

Many constraints require multiple `TRIGGER`s

- You need to consider all combinations of insertions, deletions, and tuple updates that can cause which a constraint be violated and deal with them separately.
- In our example, the number of actors in a movie can increase if we `UPDATE` a tuple already in the Cast table, changing its movie. To deal with that, we'd need another trigger, to run `BEFORE UPDATE` statements are done.

`STATEMENT` triggers

Statement triggers can also be used to enforce table-level constraints:

```
CREATE TRIGGER Constraint_1B
BEFORE INSERT ON Cast REFERENCING NEW TABLE AS temp
FOR EACH STATEMENT
WHEN EXISTS (SELECT title,year
             FROM Cast
             GROUP BY title, year
             HAVING (COUNT(*) > 10))
BEGIN
  SELECT RAISE(ABORT, "Constraint #1 violated!");
END;
```

NOTE: SQlite does not support `STATEMENT` triggers at the time of writing...

Dealing with Constraint violations

What should happen when an update statement would result in a constraint being violated?

- normally, we want to **abort** such update
- however, this may not be the default behavior[4]

Note that some systems might allow *some* of the updates to persist and prevent only the ones that violated the constraint.

In our example, if we used a `ROW` trigger and attempted to insert three actors to a movie with 8 actors already, might mean that the first two actors are accepted and only the third is rejected.

[4] SQLite allows the programmer to specify what to do for each individual `UNIQUE`, `NOT NULL` and `CHECK` constraint – `https://sqlite.org/conflict.html`

Side effects?

Can one trigger *fire* another one?

- YES! If a trigger on table `A` contains insert statement into table `B` (note `A` and `B` can be the same), all constraints about table `B` are checked by the DBMS
- in such cases, the execution environment (and the *transient* tables) of the secondary trigger depends on the master trigger

In a complex database with many business rules, even the simplest update can cause many triggers to fire and take a long time to complete.

Read up on `https://www.sqlite.org/pragma.html#pragma_recursive_triggers` to complete assignment 1.

Statement or Row triggers?

SQL offers row-based and statement-based triggers to that the programmer can decide which events are best suited to enforce the business rule in their application. There is no clear cut distinction of when one should be used versus the other.

The *execution context* of a trigger (e.g., the values bound to the `new` and `old` variables) is defined by *transient* tables which the DBMS must materialize.

- Of course, these tables can be quite large, especially for statement triggers.

Even more complex constraints?

Some constraints can't be enforce per statement nor per row...

Example: requiring every movie to have at least two actors (i.e., tuples in `Cast`)?

- Note that this means every time we insert a new movie (statement 1) we insert at least two actors (statement 2)...
- But the foreign key constraint from `Cast` into `Movie` means we **cannot** insert an actor before we insert the movie!

We cannot enforce such circularly-dependent constraints with the standard SQL constraint checking mechanisms.

Ways to deal with such constraints:

- **Delegate to the application**: problematic when there are multiple applications (i.e., need to repeat code)
- Define a **stored procedure** to insert all tuples at once[5]

Some DBMSs offer non-standard solutions, that are not specified by the SQL standard.

- PostgreSQL allows constraint checking to be *deferred*, which implicitly allows the constraint to be checked once per complex transaction.
- MS SQL server and Oracle allow the DBA to schedule the periodic execution of code (similar to a Unix *cron job*) which can bed used to **detect** constraint violations.

[5] Stored procedures reside in the DBMS can can be used by all applications–avoiding code duplication.

Basic SQL

Syntax and Semantics

Basic `SELECT ... FROM ... WHERE` queries

Unlike the algebra, SQL is a very complex language that offers great freedom to the programmer—meaning the same query can be written in many different ways.

CMPUT391 focuses on fundamental aspects of the language only, leaving out more esoteric constructs.

The basic SQL construct looks like this

```
  SELECT value expression 1 [, ... [, value expression n]]
[ FROM table expression 1 [, ... [, table expression m ]]]
[ WHERE predicate 1 [ AND ... [ AND predicate k ]]]
```

Expressions inside square brackets are optional.

As their name suggests, **value expressions** must compute atomic *values* (as opposed to lists of values, tuples, etc.)

A very common *value expression* is just an attribute name.

```
SELECT m.title
FROM Movie AS m;
```

But we can use operators and functions too.[6]

```
SELECT "title:{"||m.title||"}"
FROM Movie AS m;
```

In fact, anything that computes a single value is OK, including full SQL queries or expressions with them.

```
SELECT 100 + (
         SELECT COUNT(*)
         FROM Movie AS m
       );
```

[6] All examples follow SQLite notation; `||` means string concatenation

As their name suggests, **table expressions** must compute tables

The most common *table expression* is just a table name.[7]

```
SELECT m.title
FROM Movie AS m;
```

But we can use full SQL "sub-queries" too.

```
SELECT bmm.title
FROM (SELECT title, year FROM Cast
      WHERE actor="Bill Murray") AS bmm;
```

In fact, anything that results in a table is OK.[8]

```
SELECT MAX(temp.b)
FROM (SELECT 1 AS a, 1 AS b
      UNION
      VALUES (2,2), (3,3)) AS temp;
```

---

[7] The **AS** v clause creates a tuple variable v, making the query longer but *more readable*. It is actually optional: `Movie m` would accomplish the same.

[8] **AS** can also be used to assign *names* of columns and tables.

As for the **predicates**...

Most common: comparisons involving attributes and, possibly, built-in or user-defined functions.

```
SELECT m.title
FROM Movie AS m
WHERE length(m.title) < 5;
```

Using full SQL “sub-queries” is pretty common too.

```
SELECT m.title
FROM Movie AS m
WHERE m.imdb = (SELECT MAX(m2.imdb)
                FROM Movie AS m2);
```

And so is testing set membership.

```
SELECT m.title
FROM Movie AS m
WHERE m.year IN (VALUES (1987),(1988));
```

But there `EXISTS` many other kinds of useful predicates.

How do we execute a simple SQL query?

Here's an algorithm that works (but is not the most efficient):

- Go through each possible assignment of tuples to tuple variables tuple variable in the `FROM` clause.
- For each such assignment, evaluate the `WHERE` clause.
- If the value is `TRUE`, evaluate the expressions in the `SELECT` clause, and add them the to answer to the query.

Example:

```
SELECT g.film, g.year, m.director
FROM Guide AS g, Movie AS m
WHERE g.film = m.title AND g.year = m.year AND
      g.theater <> "Cineplex" AND m.imdb > 5
```

Input: value expressions $\mathcal{V} = \{ve_1, \ldots, ve_n\}$, table expressions $\mathcal{T} = \{te_1, \ldots, te_m\}$,
predicates $\mathcal{P} = \{p_1, \ldots, p_k\}$

Algorithm 1 pseudocode to answer a **basic SQL query**

1: **if** $\mathcal{T} = \emptyset$ **then**
2: **if** $\mathcal{P} = \emptyset$ OR $(p_1 \wedge \cdots \wedge p_k)$ **then** *% no predicates OR all predicates are true*
3: compute tuple $(ve_1, \ldots, ve_n)$ and add it to the result
4: **else**
5: **for each** $te_i \in \mathcal{T}$ **do**
6: $T_i \leftarrow$ table computed by te_i;
7: **for each** $t \in T_1 \times \cdots \times T_m$ **do**
8: **if** $\mathcal{P} = \emptyset$ OR $(p_1(t) \wedge \cdots \wedge p_k(t))$ **then** *% no predicates or t satisfies all predicates*
9: compute tuple $(ve_1, \ldots, ve_n)$ and add it to the result

The `WHERE` clause in a query can connect individual predicates using the logical connectives `AND`, `OR` and `NOT`, and one can use parenthesis to determine the precedence of the predicates and sub-expressions.

In a conjunctive query, the SQL processor will stop the evaluation of the `WHERE` clause once any predicate evaluates to `FALSE`. The processor will evaluate simpler predicates (e.g., comparisons involving constants) before evaluating the more complex ones (e.g., comparisons involving sub-queries).

Recall that logical expressions in SQL are three-valued[9] (true, false, unknown). Combinations of tuples for which the `WHERE` evaluates to false or unknown are not used in the answer of the query.

---

[9] `https://en.wikipedia.org/wiki/Three-valued_logic`

Aggregation in SQL

The SQL:1999 language is strictly more powerful than the "standard" relational algebra as discussed in the notes in the sense that every query expressible in the algebra is expressible in SQL, but not vice-versa.

One feature of SQL that is not in the original algebra is the ability to compute *aggregated values* (typically simple statistics) from **sets of tuples**

Set functions in SQL

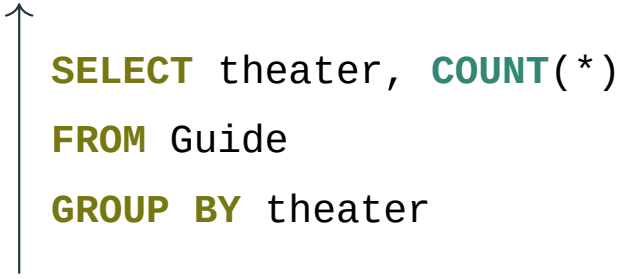
- `AVG()`: average (mean)
- `COUNT()`: number of values
- `MAX()`: largest value
- `MIN()`: smallest value
- `SUM()`: sum of values

Most DBMSs, including SQLite[10], implement more set functions than the ones in the SQL:1999 standard.

[10] `https://sqlite.org/lang_aggfunc.html`

In an aggregation query, one value expression in the `SELECT` clause is a set function:

theater	COUNT
Garneau	2
Princess	1



```
SELECT theater, COUNT(*)
FROM Guide
GROUP BY theater
```

Guide

theater	film	year	start
Garneau	Ghostbusters	1984	1140
Garneau	Ghostbusters	2016	1290
Princess	Wadjda	2012	1260

The `GROUP BY` clause tells SQL how to define the sets of tuples upon which to apply the set function: each (combination of) unique value(s) defines a set.

To answer a simplified aggregation query:

```
  SELECT value expression 1 [, ... [, value expression n]]
[ FROM table expression 1 [, ... [, table expression m ]]]
[ WHERE predicate 1 [ AND ... [ AND predicate k ]]]
[ GROUP BY attribute 1 [, ... [, attribute i ]]]
[ HAVING condition 1 [ AND ... [ AND condition j ]]]
```

STEP 1: process the `FROM` and `WHERE` clauses (slide 45);

STEP 2: divide the tuples from STEP 1 into *sets* according to the `GROUP BY` clause; if there is none, use all tuples as a single set;

STEP 3: if the `HAVING` clause is specified, discard sets of tuples that do not satisfy all conditions;

STEP 4: evaluate all expressions in the `SELECT` clause for each set separately.

SQL supports a lot more in the `GROUP BY` and `HAVING` clauses than we cover in CMPUT391, including many operators for analytical data processing[11], and value expressions.

For example, counting movies by decade:

```
SELECT 10 * (year / 10), COUNT(*)
FROM Movie
GROUP BY year / 10
```

```
1980 | 2
2000 | 1
2010 | 2
```

Note that the value expression in the `SELECT` clause need not be the same as in the `GROUP BY` clause.

[11] https://en.wikipedia.org/wiki/Online_analytical_processing

DISTINCT and ORDER BY

```
  SELECT [DISTINCT] value expression 1 [, ... [, value expression n]]
[ FROM table expression 1 [, ... [, table expression m ]]]
[ WHERE predicate 1 [ AND ... [ AND predicate k ]]]
[ GROUP BY attribute 1 [, ... [, attribute i ]]]
[ HAVING condition 1 [ AND ... [ AND condition j ]]]
[ ORDER BY col_no [ASC|DESC] ]
```

DISTINCT forces the DBMS to remove duplicates in the output. It is evaluated after all tuples are computed as before.

ORDER BY is executed last, even after **DISTINCT**, to sort the tuples in the output by the column specified (by is number from left to right).

Set/Bag Operations

Set/bag operators in SQL

SQL provides operators `UNION`, `INTERSECT`, and `EXCEPT` which are analogous to the algebra operators $\cup$, $\cap$ and $-$, respectively.

Unlike the algebra, however, SQL is a **bag-oriented language**, meaning that duplicate tuples are allowed in a table;[12]
- bags are also called *multisets*.

You can tell the DBMS which "version" of the operators to use by appending the keyword `DISTINCT` (for sets) or `ALL` (for bags).

[12] Many textbooks use the term **relation** to mean a set of tuples, as in the algebra, and **table** to mean a bag of tuples (also called *rows*) as in SQL

Suppose a tuple appears m times in R and n times in S.

The following table shows the number of times it will appear in the result of each set/bag operator:

	DISTINCT	ALL
R **UNION** S	1	$m + n$
R **INTERSECTION** S	1	$\min(m, n)$
R **EXCEPT** S	0	$\max(0, m - n)$

In SQLite...

...two or more simple SQL queries can be connected using `UNION` (which behaves as `UNION DISTINCT`), `UNION ALL`, `INTERSECT` and `EXCEPT`.

Read `https://sqlite.org/lang_select.html`

Negation and Quantification

Negation in SQL

Queries that have only selections, projections, and joins, and for which the `WHERE` clause has only `AND` connectives form a class of queries called *conjunctive queries*.[13]

If we also allow the use of disjunctions, either having `OR` connectives in the `WHERE` clause or using the `UNION` keyword, we have the class of *positive queries*.

These classes of queries are important and very well studied because most queries fall in either class and because they can be highly optimized by the DBMS.

However, these queries cannot express negation as in "finding directors of movies **without** Bill Murray".

[13] More about this on the notes about the relational algebra

One way of achieving negation in SQL is by using the “set difference” **EXCEPT** operator between two queries:

```
SELECT director
FROM Movie
EXCEPT
SELECT m.director
FROM Movie AS m, Cast AS c
WHERE m.title=c.title AND m.year=c.year AND c.actor="Bill Murray";
```

Another, perhaps more readable way:

```
SELECT m.director
FROM Movie AS m
WHERE NOT EXISTS(
    SELECT *
    FROM Movie AS m2, Cast AS c
    WHERE m2.title=c.title AND m2.year=c.year AND
        c.actor="Bill Murray" AND m2.director=m.director);
```

Yet another way of expressing the query directors of movies without Bill Murray:

```
SELECT director
FROM Movie
WHERE director NOT IN (
    SELECT m.director
    FROM Movie AS m, Cast AS c
    WHERE m.title=c.title AND m.year=c.year AND
        c.actor="Bill Murray");
```

Monotonicity

If a query is *monotonic*[14], *adding* tuples to the database either leaves the result of the query unchanged or makes it grow.

All positive queries are monotonic.

- **Ex:** `SELECT DISTINCT year FROM Cast WHERE actor='Bill Murray'`
- Adding a tuple to `Cast` cannot reduce the number of years in which Bill Murray had a movie.

Negative queries are non-monotonic: adding tuples might make the answer of the query smaller.

```
SELECT director
FROM Movie
WHERE director NOT IN (
    SELECT m.director
    FROM Movie AS m, Cast AS c
    WHERE m.title=c.title AND m.year=c.year AND
        c.actor="Bill Murray");
```

[14] https://en.wikipedia.org/wiki/Monotonic_query

Quantifiers in SQL

Besides the algebra, SQL also borrows elements from Logic-based languages. For example, tuple variables in SQL are **existentially quantified**.

```
SELECT m.director
FROM Movie AS m, Cast AS c
WHERE c.actor="Bill Murray"
      AND m.title=c.title
      AND m.year=c.year;
```

The query above seeking directors of movies with Bill Murray helps explain the concept.

Note that for a Movie m to have its director in the answer, all we need is that *there exists* a tuple c in `Cast` such that m and c satisfy the conditions in the `WHERE` clause.

In this case, variable c is existentially quantified in the query.

What if we wanted directors who directed **all** movies in which Bill Murray appears?

In this case, for a Movie m to have its director in the answer it must be the case that **all** tuples in `Cast` that have Bill Murray as actor correspond to a movie directed by that director.

Now, variable c is universally quantified in the query.

Universal quantification via double negation

SQL does not have specific notation for universal quantification.

We use **double negation** instead: the director of a Movie m should appear in the answer if there is no tuple c in `Cast` such that c has Bill Murray and c's movie is not directed by the director of m.

Universal quantification via double negation:

```
SELECT m.director
FROM Movie AS m
WHERE NOT EXISTS (
            SELECT c.title, c.year
            FROM Cast AS c
            WHERE c.actor="Bill Murray"
            EXCEPT
            SELECT m2.title, m2.year
            FROM Movie AS m2
            WHERE m.director=m2.director
    );
```

Common Table Expressions

Table expressions

The `FROM` clause in an SQL statement can contain *any expression* that computes a table.

Example: “which roles did Sigourney Weaver play in a top movie (rated 7 or higher)?”

```
SELECT c.role
FROM Cast c,
     (SELECT title, year FROM Movie WHERE imdb >= 7) AS tm
WHERE c.title=tm.title AND c.year=tm.year AND
      c.actor="Sigourney Weaver";
```

In the query above, `AS` creates tuple variable `tm` (“top movie”).

We can make the query more readable by “declaring” all computed tables before the main query:

```
WITH TopMovie(title, year) AS (
    SELECT title, year FROM Movie WHERE imdb >= 7
)
SELECT c.role
FROM Cast c, TopMovie tm
WHERE c.title=tm.title AND c.year=tm.year AND
      c.actor="Sigourney Weaver";
```

Common Table Expressions

SQL queries declared inside `WITH` clauses are sometimes called **common table expressions** or CTEs

We can have multiple CTEs in the same query:

```
WITH TopMovie (title, year) AS (
  SELECT title, year FROM Movie WHERE imdb >= 7
),
RecentMovie (title, year) AS (
  SELECT title, year FROM Movie
  WHERE strftime("%Y", year||"-01-01") >=
        date("now","start of year")
)
SELECT c.role
FROM Cast c, TopMovie tm, RecentMovie rm
WHERE c.title=tm.title AND c.year=tm.year AND
      c.title=rm.title AND c.year=rm.year AND
      c.actor="Sigourney Weaver";
```

What is going on here?[15]

- `strftime()` produces date/time values from a string;

- the double-pipe `||` operator concatenates two values (casting values to the proper types as needed);

```
WITH TopMovie (title, year) AS (
  SELECT title, year FROM Movie WHERE imdb >= 7
),
RecentMovie (title, year) AS (
  SELECT title, year FROM Movie
  WHERE strftime("%Y", year||"-01-01") >=
        date("now","start of year")
)
SELECT c.role
FROM Cast c, TopMovie tm, RecentMovie rm
WHERE c.title=tm.title AND c.year=tm.year AND
      c.title=rm.title AND c.year=rm.year AND
      c.actor="Sigourney Weaver";
```

- `strftime("%Y", year||"-01-01")` returns the date Jan. 1st of the year of the movie;
- `date("now","start of year")` returns Jan. 1st of the current year.

[15] The functions shown here are specific to SQLite! read up `https://sqlite.org/lang_datefunc.html`

Recursive Queries

Recursion in SQL

There are many applications whose data consist of objects arranged in a *graph*:

- cities connected by roads, rail lines or flights;
- people connected by friendship, familial, or business relations;
- courses connected by the pre-requisite relationship;
- etc.

An important kind of query common to all these applications requires *computing arbitrary paths* through these networks:

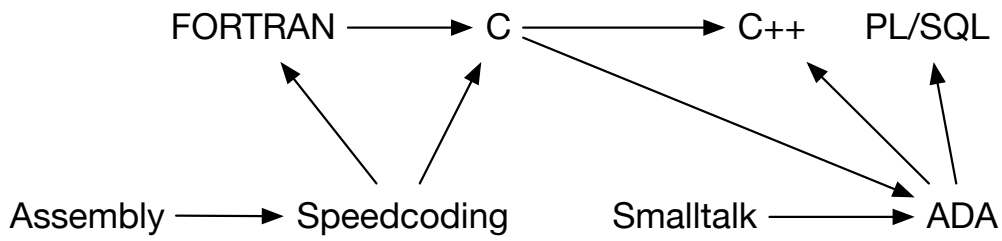
- finding all cities reachable from Edmonton by air with at most 2 connecting flights;
- finding all actors directly or indirectly linked to Kevin Bacon.[16]

[16] `https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon`

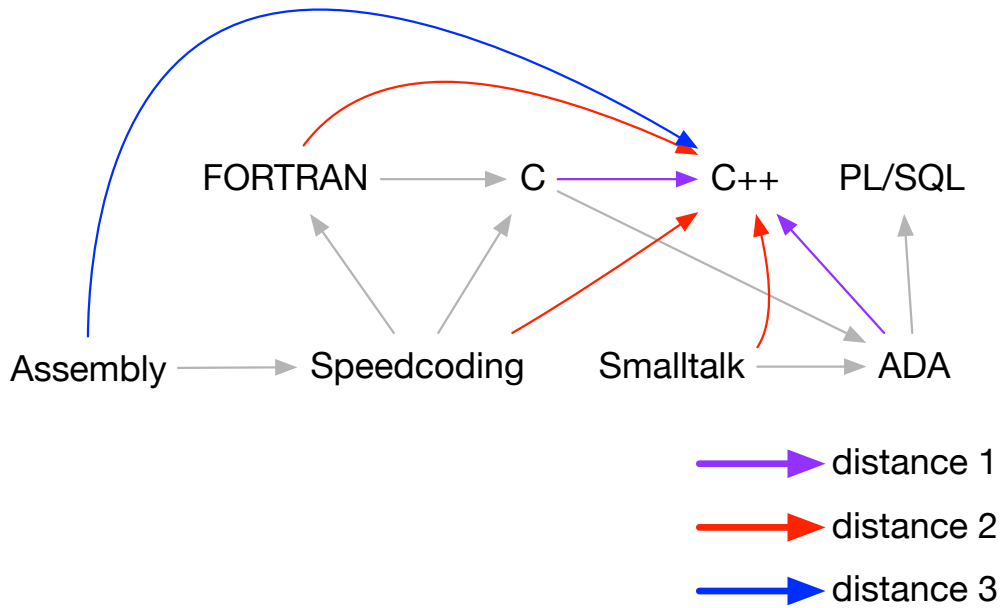
Example: graph of which languages influenced others.[17]

Influence

source	target
Assembly	Speedcoding
Speedcoding	FORTRAN
Speedcoding	C
FORTRAN	C
C	C++
Smalltalk	ADA
ADA	C++
C	ADA
ADA	PL/SQL



Which languages (directly or indirectly) influenced C++?



[17] Gathered browsing Wikipedia

Distance=1: languages that **directly** influenced C++

```
SELECT source
FROM Influence
WHERE target="C++"
```

Distance=2: languages that influenced those that **directly** influenced C++

```
WITH dist1(language) AS (
  SELECT source FROM Influence
  WHERE target="C++"
)
SELECT source FROM Influence
WHERE target IN
  (SELECT language FROM dist1)
```

For this simple graph, the answer is the **union** of the queries above, as there is no language influencing C++ that is a distance 3 or higher...

But can we write a **single query** that works on **all graphs** regardless of the maximum distance?

Distance=n: languages that influenced languages at distance $n-1$ from C++

```
WITH dist_n_1(language) AS ???
SELECT source FROM Influence
WHERE target IN
  (SELECT language FROM dist_n_1)
```

"Standard" SQL cannot compute arbitrary-length paths

- Each tuple in `Influence` is an edge in the graph
- Each tuple variable can bind to a single tuple

Therefore, a SQL query with n tuple variables can only compute a path of length with at most n edges.

Finding languages that influenced C++ by taking the recursively computed **union**[18] of a query that finds languages that find direct influences to C++ and another that finds direct influences to those languages that influenced C++:

```
WITH RECURSIVE Answer(language) AS (
  -- influenced C++ direclty:
  SELECT source FROM Influence WHERE target="C++"
UNION
  -- influenced some language that influenced C++:
  SELECT inf.source
  FROM Influence inf, Answer A
  WHERE inf.target = A.language
)
SELECT * FROM Answer;
```

[18] Recall that in SQLite **UNION** defaults to **UNION DISTINCT**

SQL:1999 Recursion

Any recursive query must be defined as the `UNION` of two queries:

```
WITH RECURSIVE recursive_CTE AS (
  base query
UNION
  recursive query
)
```

The `base query` is non-recursive and must be defined over base tables only.

The `recursive query` must refer to the `recursive_CTE` exactly **once** and can refer to base tables as well.

```
WITH RECURSIVE recursive_CTE AS (
  base query
UNION
  recursive query
)
```

Also: the `recursive query` must be *monotonic* (recall Slide 57).

In particular, it cannot use:

- Aggregation;
- `NOT EXISTS` with a subquery involving itself;
- `EXCEPT` in which it is the right subquery.

SQL:1999 Recursion — Semantics

The *fixpoint* of an operator

Suppose we start with a relation R, and that we repeatedly redefine it based on an operator $\oplus$ like so:

$$R_1 = \oplus(R);\ R_2 = \oplus(R_1);\ R_3 = \oplus(R_2);\ \cdots$$

We reach a *fixpoint* after k applications of $\oplus$ when $R_{k+1} = R_k$ (that is, $\oplus$ no longer modifies the relation after k applications)

SQL:1999 defines recursion in terms of fixpoints.

More precisely, we start with a relation and continuously apply a suitably defined `UNION` operator until we reach a fixpoint.

Computing the influence on C++ with *fixpoints*

- let Answer$_i$ be the answer at step i and $\oplus$ be the recursive query

$$
\begin{aligned}
\text{Answer}_0 &= \{\ \} \\
\text{Answer}_1 &= \oplus\left(\text{Answer}_0\right) = \{(\text{"C"}), (\text{"ADA"})\} \\
\text{Answer}_2 &= \oplus\left(\text{Answer}_1\right) = \{(\text{"C"}), (\text{"ADA"}), (\text{"FORTRAN"}), \\
&\qquad (\text{"Speedcoding"}), (\text{"Smalltalk"})\} \\
\text{Answer}_3 &= \oplus\left(\text{Answer}_2\right) = \{(\text{"C"}), (\text{"ADA"}), (\text{"FORTRAN"}), \\
&\qquad (\text{"Speedcoding"}), (\text{"Assembly"}), (\text{"Smalltalk"})\} \\
\text{Answer}_4 &= \oplus\left(\text{Answer}_3\right) = \text{Answer}_3
\end{aligned}
$$

We reach a *fixpoint* at step 4 because no more languages are added to the answer.

Generalizing Influence — Graph Reachability

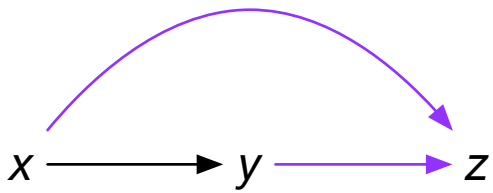
To find out "which languages influenced *an arbitrary* language y" we need to find all languages x such that $x \rightsquigarrow$ y.

Reachability in a graph

In an arbitrary graph, we say node y is reachable from node x if there is a path $x \rightsquigarrow y$.

Computing *reachability* with fixpoints

- let the answer be the original graph
- let $\oplus$ be the union of the current graph and new edges formed by collapsing paths of length 2



To find paths of arbitrary length recursively.

(1) Let P be the set of all paths of length 1 (i.e., all edges in `Influence`).

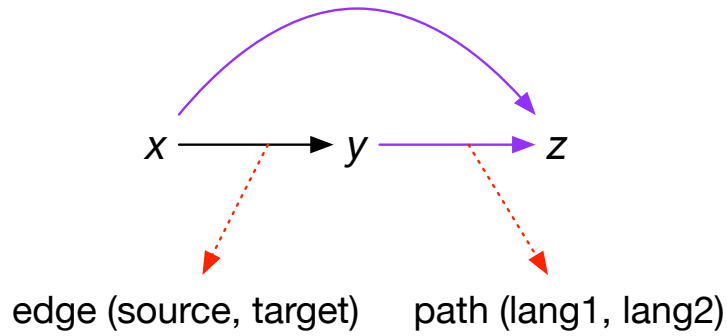
(2) For every pair of paths $p_1 = (s_1, t_1) \in P, p_2 = (s_2, t_2) \in P$ such that $p_1.t_1 = p_2.s_2$ (that is, p_1 ends in the language that p_2 starts):

    - if $p_3 = (s_1, t_2)$ isn't already in P; add it to P

(3) Repeat step (2) until no more new paths are added.

Computing *reachability* on the language influence graph:

```
WITH RECURSIVE Inspire(lang1, lang2) AS (
    SELECT source AS lang1, target AS lang2
    FROM Influence
  UNION
    SELECT edge.source, path.lang2
    FROM Inspire path, Influence edge
    WHERE edge.target=path.lang1
)
SELECT *
FROM Inspire;
```



To find the languages that influenced a language XYZ, filter the reachability graph to those edges incident on XYZ by adding the clause `WHERE lang2="XYZ"` to the query above.

What is going on here?

`Inspire` is the "answer": the table representing the reachability graph.

We start by adding all edges in `Influence` to the answer.

```
WITH RECURSIVE Inspire(lang1, lang2) AS (
    SELECT source AS lang1, target AS lang2
    FROM Influence
  UNION
    SELECT edge.source, path.lang2
    FROM Inspire path, Influence edge
    WHERE edge.target=path.lang1
)
SELECT *
FROM Inspire;
```

Then the **`UNION`** operator computes and adds *new paths* $x \rightsquigarrow z$ whenever it can find an *edge* $x \to y$ in `Influence` to extend a *path* $y \rightsquigarrow z$ already in the reachability graph.

IMPORTANT: because **`UNION`** removes duplicates, we are guaranteed to reach a fixpoint!

What if we want only paths of length up to k?

- we could take the union of queries computing paths of length $1, 2, \ldots, k$;

- **or** we can **impose a bound** on the recursion.

Example ($k = 5$):

```
WITH RECURSIVE Inspire(lang1, lang2, len) AS (
    SELECT source AS lang1, target AS lang2, 1 AS len
    FROM Influence
  UNION
    SELECT edge.source, path.lang2, path.len+1
    FROM Inspire path, Influence edge
    WHERE edge.target = path.lang1 AND path.len < 5
)
SELECT *
FROM Inspire;
```

What if there are cycles in the graph?

Edge

source	target
n_0	n_1
n_1	n_2
n_2	n_1
n_2	n_3

```
WITH RECURSIVE Path(begin_, end_, len) AS (
    SELECT source, target, 1
    FROM Edge
  UNION
    SELECT e.source, p.end_, p.len+1
    FROM Path p, Edge e
    WHERE e.target = p.begin_ AND p.len < 5
)
SELECT *
FROM Path;
```

We get many "copies" of the path $n_1 \rightsquigarrow n_2$ with different lengths:

- length 1: $n_1 \to n_2$
- length 3: $n_1 \to n_2 \to n_1 \to n_2$
- length 5: $n_1 \to n_2 \to n_1 \to n_2 \to n_1 \to n_2$

Recursion on Numbers

Of course, recursion in SQL is not restricted to graph traversals. In fact, one can express many familiar recursive computations in SQL.

Example: the sum of the first $N = 100$ non-negative numbers:

```
WITH RECURSIVE c(n) AS (
   SELECT 1
  UNION
    SELECT n+1 FROM c WHERE n<100
) SELECT SUM(n)
FROM c;
```

DBMS support for recursive SQL

The SQL:1999 standard defined recursion in terms of the fixpoint operator on **common table expressions** as discussed here.

In the 1980's, Oracle introduced a different syntax and mechanism for recursion through so-called *hierarchical queries*.[19]

Today, both Oracle and IBM support both forms of recursion, while Microsoft's SQL Server supports the SQL:1999.

In the open-source world, besides SQLite, PostgreSQL[20], Firebird[21], and CUBRID[22] support the SQL:1999 standard.

---

[19] https://en.wikipedia.org/wiki/Hierarchical_and_recursive_queries_in_SQL

[20] https://www.postgresql.org/docs/current/static/queries-with.html

[21] http://www.firebirdsql.org

[22] http://www.cubrid.org