Information and Computation ••• (••••) ••••••

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems

Susana Nieva *, Fernando Sáenz-Pérez, Jaime Sánchez-Hernández

Facultad de Informática, Universidad Complutense de Madrid, Spain

ARTICLE INFO

Article history:
Received 17 November 2016
Received in revised form 19 July 2019
Accepted 9 October 2019
Available online xxxx

Keywords:
Relational databases
SQL
Recursion
Hypothetical queries
Fixpoint semantics

ABSTRACT

In this work we present a formalization, backed with a contrasted implementation, of a relational database language (called HR-SQL) that extends SQL in two aspects. On the one hand, including non-linear and mutual recursion. On the other hand, including hypothetical relations and queries. Regarding expressiveness, HR-SQL allows a novel form of hypothetical reasoning, completely integrated with recursive definitions. In addition, aggregate functions are added. Regarding formalization, the extended language is founded on a stratified fixpoint semantics based on logic programming techniques. We include results of the existence of such fixpoints. An algorithm that transforms a database containing hypothetical definitions into an equivalent one without hypothesis and its correctness proof are presented. Regarding the implementation, we introduce here a system that incorporates the aforementioned benefits and enhances former implementations in several areas. The current HR-SQL system is targeted to several state-of-the-art relational database systems, and could be used with any SQL-based system with minor modifications in the implementation.

1. Introduction

Current relational database management systems (RDBMS's) include SQL as a *de-facto* query language, which evolves from formal query languages as originally proposed for the relational model: Relational Algebra (RA) and Relational Calculus (RC), which are syntactically different, but semantically equivalent with respect to safe formulas [44]. However, as pointed out by many (see, e.g., [28], [34]), the relational model has several limitations. Current SQL implementations (somewhat following the standard SQL:2011 [26]) depart from the relational model and go beyond. Its acknowledged success builds upon an elegant and yet simple formulation of a data model with relations which can be queried with a language including some basic RA-operators. Original operators became a limitation for practical applications of the model, and others emerged to fill some gaps, including, for instance, aggregate operators. But in general, these additional features lack a formal semantics.

In this paper we address two main lines for enhancing SQL: broadening recursion and incorporating hypothetical definitions and queries. Moreover, we also support aggregates in this context. The extended database language is called HR-SQL. We present a formal semantics for it based on logic programming techniques, and we develop an implementation according with the semantics.

* Corresponding author.
E-mail addresses: nieva@ucm.es (S. Nieva), fernan@sip.ucm.es (F. Sáenz-Pérez), jaime@sip.ucm.es (J. Sánchez-Hernández).

https://doi.org/10.1016/j.ic.2019.104485
0890-5401/

1.1. Main contributions of HR-SQL

On the one hand, we mention recursion. There are several main drawbacks in currents RDBMS's regarding recursion coverage and its formal semantics. Linearity is required, so that relation definitions with calls to more than one recursive relation are not allowed. Some other features are not supported: Mutual recursion, and query solving involving an EXCEPT clause to combine recursion and negation. In general, termination is manually controlled by limiting the number of iterations instead of detecting that there are no further opportunities to develop new tuples. HR-SQL copes with recursive definitions which are not restricted as current RDBMS's do, and also allows neater formulations by providing concise relation definitions (following the assignment RA-operator) and avoiding extensive writings. Regarding formalization, we define a stratified fixpoint semantics. To deal with recursion in the presence of negation we adhere to stratification in order to return a single answer set, i.e., basing our formal framework in considering only one minimal model for the semantics of queries [44], which is a natural expectation from current database users. This semantics is a straightforward guide for a concrete implementation.

On the other hand, several applications require predictive and historical analysis over large amounts of data [25], typically making some sort of assumptions to deduce conclusions. Hypothetical queries, also known as "what-if" queries, can help managers to take decisions on scenarios which are somewhat changed with respect to a given state. Such queries are used, for instance, for deciding which resources must be added, changed or removed to optimize some criterion. Current applications include, among many others, OLAP environments [9,48], business intelligence [20], and e-commerce [47]. So, driven by these needs, we face the inclusion of hypothetical definition and queries in the recursive SQL setting. Our proposal for hypothetical SQL clauses follows a similar syntax to WITH clauses, using the new constructor ASSUME (as in version 2.6 of [39]). Indeed, while a WITH clause allows to define new temporary relations with queries, an ASSUME clause allows to locally modify the definition of already-defined relations by using queries. Modification can be either augmenting the relation (ASSUME `query` IN `relation`) or decreasing it (ASSUME `query` NOT IN `relation`). HR-SQL allows a non-limited combination of recursive and hypothetical relation definitions, allowing in particular mutually recursive hypothetical relations. We present a consistent and novel formalization for recursion and hypothetical reasoning in the SQL frame, interpreting hypothetical queries as implications of the intuitionistic logic.

In addition to recursion and hypothetical reasoning, in this paper we tackle aggregates. Aggregates are commonly provided by current SQL systems to deal with summarizing functions demanded by analytical reasoning. These functions include running sum, average and minimum/maximum with respect to the elements in a set, which is typically built with a select statement including a grouping criterium. This grouping states how many sets are generated, and for all of them both aggregates and conditions including aggregates can be applied [19]. Current RDBMS's expose several limitations in the use of aggregates in recursive queries. In the setting of hypothetical reasoning (which is not supported at all by any RDBMS), source relations over which aggregates apply may be modified by assumptions (either by augmenting with new tuples or reducing by discarding existing tuples). To deal with aggregates in HR-SQL, we take advantage of the stratification technique (also applied to aggregates in other fields as in [36]). This approach extends the formalism used to support HR-SQL in a natural and effective way.

The implementation of HR-SQL that we present here is highly connected to the formalization of the language. The system is developed in Prolog for SWI-Prolog and SICStus Prolog, and provides a simple prompt for processing databases. A Java-based integrated development environment is also connected to the HR-SQL system, providing a better user experience and graphical display of the dependency graph (showing the dependencies between relations). Processing an HR-SQL database results in a Python script whose SQL statements are eventually processed by a RDBMS via a standard ODBC bridge. The script implements the fixpoint semantics by iterative generation of tuples in the external RDBMS. The final result of executing the Python script is the materialization of the HR-SQL relations in the external RDBMS. Afterwards, such relations can be queried either with the native SQL or with the extended SQL as defined in HR-SQL.

This paper presents several novelties with respect to previous ones. The language R-SQL was defined in [4] as a broad extension of SQL with recursion, and a prototype system based on it was implemented. An improved version of this system appears in [5]. Next, HR-SQL [6] emerged as an extension of R-SQL, incorporating hypothetical queries and views over already defined database, but those hypothetical views could not be referenced by other views or relations. Those restrictions are removed in the current work. Here we introduce a full combination of recursive and hypothetical relations, allowing even mutual recursion between hypothetical relations. We apply here the same fixpoint techniques used in [4,6], but in order to give semantics to the whole language, which includes new features, the theoretical framework has been reformulated and widely extended, obtaining a consistent and novel formalization for recursion and hypothetical reasoning in the SQL frame. Additionally, we have carried out a deep study of the aggregate functions, for the first time in this context. It includes a formal semantics, as well as an efficiency analysis related to the calculation of aggregates within hypothetical queries. Moreover, in this paper we include full proofs of the theoretical results. Besides, the system HR-SQL has been completely reimplemented from scratch using the parser and some other modules of the system DES [39]. The result is a much more stable and mature system with respect to its previous versions, incorporating a number of new optimizations in the generated code. We also present novel performance experiments, involving comparison between our system and native SQL.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

1.2. Related work

To the best of our knowledge, there have been no other similar approaches to ours in the literature combining recursion and hypothetical queries as we do allow, avoiding the aforementioned drawbacks of current systems regarding recursion and providing formal support for an SQL-based language. However, we list some related works in both the relational and logic programming fields regarding recursion and hypothetical reasoning.

With respect to recursion, Starburst [29] was the first non-commercial RDBMS to implement it, whereas IBM DB2 was the first commercial one. Regarding formalization, an extension of the RA is presented in [1], with a looping construct and assignment in order to deal with the integration of recursion and negation. [18] is the source of the original SQL:1999 proposal for recursion (as still included in the current SQL standard) which is based on the research in the areas of logic programming and deductive databases [44]. Another example of an approach built on an extension of RA with a fixpoint construct is in [24]. However, as far as we know, these formalizations do not lead to concrete implementations, while our proposal provides an operational mechanism allowing a straightforward implementation.

With respect to hypothetical relational databases, the very first work was presented in [41], where hypotheses were stated by replacing actual data with a REPLACE operator, and assumed data persist until the query is finished. In that early work, recursion was not considered. T. Griffin and R. Hull present in [22] extensions of RA to support hypothetical queries and propose implementation techniques based on equational theories. The hypotheses are represented as sequences of SQL update expressions. Our proposal includes hypothetical queries as well as definitions of hypothetical relations, and introduces recursion. The hypothesis are represented as general SQL queries, that are not treated like updates. The main resemblance between both works is the use of substitutions to replace relations by queries when defining the semantics of a hypothetical query. However, in HR-SQL those substitutions must be applied to the whole definition database since several relation definitions may depend of the assumed hypothesis. In addition, the existence of recursion and possibly negative assumptions makes the study considerably broader. Nonetheless, the MODEL clause in Oracle SQL data warehousing [31] allows an approach to hypothetical queries. However, this only deals with assuming a data array to which queries (even recursive) can be posed, and does not allow to make assumptions on existing relations so that both augmented relations and original relations can be used throughout the whole query processing.

On the logic programming arena, it is well known the use of stratified semantics to deal with deductive databases including recursion, negation and aggregates (see [21] for a survey). We use similar techniques to formalize an extend the SQL database language, and in addition to those facilities we also include hypothetical relations. The work [43] related to DLV^{DB}, translates recursive rules in SQL statements using a semi-naïve strategy, but first, it does not include hypothetical queries and, second, it does not deal with SQL queries, but with the translation of Datalog to SQL. Therefore, no (involved) SQL statements are possible as inputs to the system. Hypothetical Datalog [12,13] fits into intuitionistic logic programming, an extension of logic programming including both embedded implications and negation, and integrates atomic assumptions as hypothetical queries in the inference system. It has been a proposal thoroughly studied from semantic and complexity point-of-views, allowing to add and delete atoms in assumptions in order to proving goals. Based on this work, [38] includes a proposal to deal with duplicates, strong constraints, and intensional assumptions. More recently, in [2] a technique based on derivatives of fixpoints is introduced, which provides a recursive semantics for Datalog that can handle negation, disjunction, aggregates and existential quantification, but the implication is not included. So, hypothetical reasoning is not supported. Further, the educational system DES [39] (as an extended implementation of [38]) also includes hypothetical SQL queries, similar in syntax to the proposal in this paper. However, there are some important differences. First, it is an in-memory database system by contrast to the persistent and materialized approach as shown here. Second, its semantics for assumptions is different because they are not sensitive to their order, in contrast to this work in which successive replacements, according to the assumption order, is proposed. Last, it is not targeted at performance. In another work, [3] we developed a more expressive setting for constraint deductive databases based on Hereditary Harrop formulas. In particular, it provides support for assuming rules as hypothetical queries. Our current work can be understood as porting this feature to relational databases, allowing also to assume negative information.

Other works also deal with database updates as [10,14,15], which can be seen as allowing some sort of hypothetical reasoning via insertions and deletions, similar to the aforementioned paper [12]. Following this work, a transaction logic [11] is proposed to account for database updates. Therefore, for hypothetical reasoning, it is needed for the user to explicitly insert and retract data. Also, temporal reasoning [7] is related as the hypothetical evolution of a database that can be queried without updating the entire database. However, this work is oriented towards reasoning along time as database transactions need.

1.3. Organization of the paper

We continue by introducing the database definition language of HR-SQL in Section 2, emphasizing the expressiveness of the language and comparing its capabilities to current RDBMS's. For this language, in Section 3, we develop a formalization based on stratified interpretations and a fixpoint operator to support theoretical results. In order to compute hypothetical relations, in Section 4 we present an elaborated algorithm to process the assumptions of the database relations, that converts a database including hypothesis into an equivalent one without hypothetical relations. We have proved soundness and completeness of the algorithm, as well as termination by using a sophisticated ordering. Section 5 introduces the syntax and

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

```
db       ::= R sch := query; ... R sch := query;
query    ::= sel_stm | query UNION query | query EXCEPT query
sel_stm  ::= SELECT exp,...,exp [ FROM R,...,R [ WHERE wcond ] ]
sch      ::= (A T,...,A T)
exp      ::= C | R.A | exp m_op exp | - exp
wcond    ::= TRUE | FALSE | exp c_op exp | NOT wcond | wcond [AND|OR] wcond
m_op     ::= + | - | / | *
c_op     ::= = | <> | < | > | >= | <=
```

R stands for relation names, A for attribute names, T for standard SQL types (as INTEGER, FLOAT, VARCHAR(N)), and C for constants of a valid SQL type.

Fig. 1. A grammar for SQL with recursion.

semantics for aggregates in HR-SQL and some examples to showing the expressiveness when combined with hypothetical definitions. Section 6 describes the system implementation, which provides a front-end which interfaces HR-SQL inputs to an external RDBMS, producing the necessary code to efficiently compute the fixpoint of an HR-SQL database, and the corresponding queries. Section 7 includes several experiments, in particular we analyze the performance of the reachability problem in different sorts of graphs. We also study the behavior of our system by computing hypothetical queries involving aggregates over big relations. Some conclusions and points worth of exploring as future work are included in Section 8.

2. Introducing HR-SQL

For the sake of simplicity we proceed in two steps. Firstly, we present our proposal of extending SQL with a broad recursion. In Subsection 2.1, we show the core syntax of our definition language. Relations are defined with assignments allowing for introducing recursion in SQL in a direct and effective way: A relation is defined with an assignment operation as a named query that can contain a self reference, i.e., a relation `R` can be defined as `R sch := SELECT ... FROM ... R ...`, where `sch` is the relation schema. Moreover, by using this assignment it is also possible to define mutually recursive relations. This syntax is similar to the assignment operator in RA (systems as WinRDBI [17] also supports a similar syntax for schemas in both RA and RC). In a second step, in Subsection 2.3, that syntax is extended to allow hypothetical relation definitions, which include hypotheses (or assumptions) by means of an ASSUME clause. We also show the benefits of HR-SQL with respect to current RDBMS's, emphasizing on expressiveness, by means of examples, in Subsections 2.2 and 2.4.

2.1. SQL database definitions using recursion

In this subsection, we introduce our proposal for defining an SQL database using recursion in a neater way than the standard supports. The BNF grammar in Fig. 1 shows the formal syntax of a database, allowing recursive relations. Productions start with lowercase letters, terminals start with uppercase, and SQL terminal symbols use small caps. The grammar defines the following syntactic categories:

- A database `db` is a (non-empty) sequence of relation definitions separated by semicolons (";"). A relation definition assigns a select statement to the relation, that is identified by its name `R` and its schema `sch`.
- A schema `sch` is a tuple of attribute names with their corresponding types.
- An SQL statement `query` is either a simple select statement `sel_stm`, or it is recursively constructed with the UNION and/or EXCEPT operators.
- A select statement `sel_stm` is defined in the usual way. The clauses FROM and WHERE are optional. The relations occurring in the WHERE clause must also occur in the FROM one.
- An expression `exp` can be either a constant value `C`, or an attribute of a relation (denoted by `R.A`), or an arithmetic expression built up with the usual mathematical operators.
- A Boolean condition `wcond` in the WHERE clause of a select statement is built up in the usual way, using also the standard comparison operators.

As a syntactic sugar, we admit * in the projection list of `sel_stm` to represent all the attributes of relations in the FROM part. Note that the grammar does not include the clauses `[NOT] IN` and `[NOT] EXISTS`, but this does not imply an expressiveness limitation because they can be rewritten (in the absence of null values) with the already supported constructs (i.e., aggregates and joins; see, for example [23]).

The relations that are extensionally defined, i.e., those defined with a simple FROM-less `sel_stm`, would correspond to tables in the usual sense, and the intensional ones, defined implicitly with more complex queries, would correspond to usual SQL views. Along the paper we simply use "relation" referring to both. According to the grammar, an HR-SQL database definition is a set of such assignments or relation definitions, and will be treated as a whole when it is interpreted and computed, without distinguishing between tables and views.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

2.2. Expressiveness of SQL with recursion

Next, we illustrate that HR-SQL overcomes some limitations present in current RDBMS's following SQL:1999. Standard SQL defines *common table expressions* (CTE's) as the mechanism to build monotonic recursive queries. A CTE is a temporary view defined as an SQL query which can include a reference to itself, and is defined in the context of a WITH clause. Following [26], this clause allows to define temporary definitions for relations that can be recursive. The syntax of such recursive definitions for a single CTE is:

```
CREATE VIEW r AS WITH RECURSIVE r1 AS q1 (q)
```

where `r` and `r1` are schema templates (including relation and attribute names), and `q` and `q1` are queries (SELECT clauses), where both can reference `r1`. This view definition can be read as: Create a view `r` whose result is defined by the query `q`, and such that `q` can reference the new temporary view definition `r1` which is only visible in the context for `r`.

However, there exist limitations in current RDBMS's (as DB2) with respect to recursive definitions as the following:

1. First query `q1`, must not reference the CTE.
2. For any subsequent SELECT in `q`, only one reference to CTE is allowed (linear recursion).
3. The clause DISTINCT in SELECT is not allowed.
4. The operator UNION DISTINCT is not allowed.

Some RDBMS examples supporting recursive CTE's are IBM DB2, Sybase SQL Anywhere, Microsoft SQL Server, PostgreSQL and Oracle (supported since recent version 11g). Others widely used that lack this support for recursion include MySQL (up to version 5), Sybase ASE, and Microsoft Access.

Therefore, as it is pointed out in [19] and it is just shown, one of the limitations of systems based on SQL:1999 is that they do not allow an arbitrary collection of mutually recursive relations to be written in the WITH RECURSIVE clause. Although any mutual recursion can be converted to direct recursion by inlining [27], our proposal allows to explicitly define mutual recursive relations, which is an advantage in terms of program readability and maintenance.

Example 1. The classical procedure for computing even and odd numbers up to a bound (100 in the example) can be written as:

```
even(x integer) := SELECT 0 UNION SELECT odd.x+1 FROM odd;
odd(x integer)  := SELECT even.x+1 FROM even WHERE even.x<100;
```

Further, as pointed out in the introduction of this subsection, linear recursion in standard SQL restricts the number of allowed recursive calls to be only one, but this limitation is absent in our language.

Example 2. Fibonacci numbers up to a bound (10 in the example) can be defined as follows:

```
fib(n integer, f integer) :=
  SELECT 0,1 UNION SELECT 1,1 UNION
  SELECT fib1.n+1, fib1.f+fib2.f FROM fib AS fib1, fib AS fib2
    WHERE fib1.n=fib2.n+1 AND fib1.n<10;
```

Here, AS is used for renaming the relation `fib` as usual in standard SQL. This clause has not been included in the grammar of Fig. 1 in order to keep it as simple as possible. Moreover, it is only a syntactic sugar of HR-SQL that can be desugared by introducing two new relations `fib1` and `fib2`:

```
fib1(n integer, f integer) := SELECT * FROM fib;
fib2(n integer, f integer) := SELECT * FROM fib;
fib(n integer, f integer)  := SELECT 0,1 UNION SELECT 1,1 UNION
  SELECT fib1.n+1,fib1.f+fib2.f FROM fib1,fib2
    WHERE fib1.n=fib2.n+1 AND fib1.n<10;
```

The important fact in this example is that the definition of `fib` involves two recursive calls, and this is not supported by SQL. This limitation implies that several graph algorithms specified using non-linear recursion cannot be directly expressed in current recursive SQL systems [46].

Non-termination is another problem that arises associated to recursion. For instance, the basic transitive closure over a graph that includes a cycle makes current SQL systems (such as PostgreSQL and DB2) either to reject the query or to go into

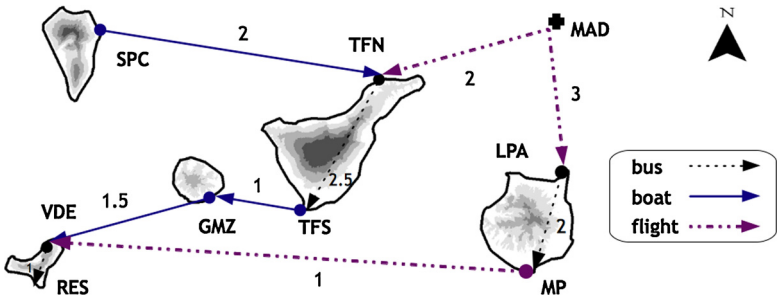


Fig. 2. Travel database for the Canary Islands.

an infinite loop. Nevertheless, the fixpoint computation used by HR-SQL, that discards duplicates, guarantees termination when dealing with finite relations. However, relations built with arithmetic expressions may be infinite. A possibility to guarantee termination is to impose safety conditions. There are several works in the field of deductive databases following this line [35,16]. An exhaustive study of safety conditions for HR-SQL is out of the scope of this paper. Note that recursive SQL is Turing complete and termination undecidability arises. Therefore, a maximum number of iterations can be imposed in HR-SQL, such as a simple termination condition of the fixpoint computation, as most usual RDBMS's do.

The following example, which in particular illustrates this problem, contains a concrete relation defined using the classical transitive closure technique mentioned above. This will be our running example in the next sections.

Example 3. This example is inspired on an example of [15] and it defines a database with the relations `flight`, `bus` and `boat` all they with schema `(ori varchar(10), des varchar(10), time float)` to store information about origin (`ori`), destination (`des`) and time (`time`), for traveling around the Canary Islands. The concrete information about the different transports is sketched in Fig. 2.

The relation `link` collects all the possible transports:

```
link(ori varchar(10), des varchar(10), time float) :=
  SELECT * FROM flight UNION SELECT * FROM boat UNION SELECT * FROM bus;
```

The relation `travel` is the transitive closure of `link`, i.e., it provides all the possible travels of the database, maybe concatenating any of the available transports:

```
travel(ori varchar(10), des varchar(10), time float):=
  SELECT * FROM link UNION
  SELECT link.ori, travel.des, link.time + travel.time
    FROM link, travel WHERE link.des=travel.ori;
```

The relation `reachable` is like `travel` but without taking into account the time information:

```
reachable(ori varchar(10), des varchar(10)) :=
  SELECT link.ori, link.des FROM link UNION
  SELECT link.ori, reachable.des FROM link,reachable
    WHERE link.des = reachable.ori;
```

Both `reachable` and `travel` represent transitive closures of the relation `link`. But notice that if `link` has a cycle, then the relation `travel` that includes times for each trip is infinite, while `reachable` is not. In order to ensure termination, the system might check if some computation limitation have been imposed (as the maximum time for a travel, for example). A warning message would be raised if not.

As mentioned before, for recursive definitions, SQL systems prohibit the clause DISTINCT in the select statement and UNION ALL is required; thus, duplicates cannot be discarded. So, an equivalent formulation (modulo duplicate elimination) to `reachable` in standard SQL is:

```
CREATE VIEW reachable(ori,des) AS
  WITH RECURSIVE cte(ori,des) AS
    (SELECT link.ori, link.des FROM link UNION ALL
     SELECT link.ori, cte.des FROM link,cte WHERE link.des = cte.ori)
  SELECT * FROM cte;
```

Systems such as IBM DB2 raise a warning when there exist the possibility of non-terminating queries, as in this example, and its Data Studio GUI provides the first 500 result tuples. However, Top-N queries (for selecting the first N tuples in the result set) can be used to control termination, as in:

```
SELECT * FROM reachable FETCH FIRST 10 ROWS ONLY;
```

Even so, this might lead to non-termination as the top requirement is not transmitted to relations used along the computation path. Let us consider the following example:

```
SELECT DISTINCT * FROM reachable FETCH FIRST 10 ROWS ONLY;
```

In this case, no warning is issued and the query runs until memory exhaustion.

In contrast, `reachable` can be finitely computed in our system. But, `travel` would produce an infinite set of different tuples, which is an issue due to using infinite relations. In order to ensure termination, the system might check if some computation limitation has been imposed, as the maximum time for a travel, or the general one mentioned above consisting of limiting the number of iterations. A warning message would be raised if not.

We finish this example by defining the relation `avoidMad` that contains possible travels that neither begin nor end in Madrid:

```
avoidMad(ori varchar(10), des varchar(10)) :=
  SELECT * FROM reachable EXCEPT
  SELECT * FROM reachable
    WHERE (reachable.ori = 'MAD' OR reachable.des = 'MAD');
```

This definition includes negation together with recursive relations. This combination can not be expressed in SQL:1999 as it is shown in [18].

2.3. Definition of hypothetical relations

As we pointed out in the introduction, one of the novelties of our proposal with respect to current SQL languages is the ability of dealing with hypothetical queries and even with relations defined using hypothesis. In [4], we presented the language R-SQL as a forerunner of HR-SQL that includes recursion but not hypotheses. In [6] we presented a system including hypothetical queries and views (defined for previously defined databases) with several restrictions in their definitions. The current HR-SQL definition language has been extended to support unrestricted hypothetical relation definitions, allowing even mutual recursion between hypothetical relations.

In this section we establish the formal syntax of such definitions, and next we formulate some examples showing certain expressiveness benefits of the enriched language.

The grammar of Fig. 1 can be easily extended to incorporate hypothetical queries. This extension essentially consists of incorporating the ASSUME clause involving a sequence of hypotheses `hypo`, ..., `hypo` before a query. These hypotheses (`hypo`) represent assumptions over relations, which produces adding or removing into such relations the tuples specified by the query inside `hypo`, before answer the query itself. The formal grammar is:

`query`	`::=`	`sel_stm` | `sel_hyp` | `query` **UNION** `query` | `query` **EXCEPT** `query`
`sel_hyp`	`::=`	**ASSUME** `hypo, ... , hypo query`
`hypo`	`::=`	`query` [**NOT**] **IN** `R`

Now, a relation definition `R sch := query` can include a hypothetical query `sel_hyp`. In the following, we say *hypothetical relation* to refer to relations whose definition contains an assume clause. Each `hypo` part is usually called hypothesis. We impose a natural restriction in the definition of hypothetical relations: their definitions do not include assumptions over themselves. This means that if `sel_hyp` occurs in the definition of a relation `Rh`, and ASSUME `query` [NOT] IN `R` is included in `sel_hyp`, then $R \neq Rh$. This is not a real limitation of the language, but a simplification. In fact, the case $R = Rh$ can be represented using an auxiliary relation Rh' defined as SELECT * FROM `Rh`, and replacing `Rh` by Rh', in `sel_hyp`.

From the logical point of view, a hypothetical statement ASSUME `hypo, ... , hypo query` can be interpreted as an intuitionistic implication: it represents the value of the consequent `query` assuming the antecedent (hypothesis) `hypo, ... , hypo`. Actually, this hypothetical statement is a syntactic sugar of a nested assumption; more precisely, a hypothetical query:

$$\textbf{ASSUME}\ \texttt{query}'_1\ [\textbf{NOT}]\ \textbf{IN}\ R_{a_1}, \ldots\ \texttt{query}'_m\ [\textbf{NOT}]\ \textbf{IN}\ R_{a_m}\ \texttt{query}$$

can be translated into an equivalent one:

$$\text{ASSUME } \texttt{query}'_1 \ [\text{NOT}] \text{ IN } \mathrm{R}_{a_1} (\text{ASSUME} \ldots (\text{ASSUME } \texttt{query}'_m \ [\text{NOT}] \text{ IN } \mathrm{R}_{a_m} \ \texttt{query}) \ldots)$$

Hence, we will use the simplified syntax ASSUME `hypo query` in the formalizations, but we still use the syntactic sugar allowing more than one hypothesis in the examples.

2.4. Expressiveness of HR-SQL

We begin with simple hypothetical definitions referring to the running example.

Example 4. Consider the Canary islands database of Example 3. We are interested in the following information: How long does it take to arrive in Valverde from Madrid (if possible), if for each boat connection to Valverde that takes more than one hour an extra connection to Valverde taking half an hour less is added. In HR-SQL it is possible to define a relation to keep these times as:

```
mad_to_val(time float) :=
  ASSUME
    (SELECT boat.ori, boat.des, boat.time - 0.5 FROM boat
        WHERE boat.des = 'VDE' AND boat.time > 1) IN link
  SELECT travel.time FROM travel
    WHERE travel.ori = 'MAD' and travel.des = 'VDE';
```

Suppose now that, as before, for each boat connection to Valverde that takes more than one hour, an extra connection to Valverde taking half an hour less is added, but the original boat to Valverde from La Gomera is suspended. We can define a new version of `travel` to represent the reachable cities and corresponding times in the Canary islands in this hypothetical scenario:

```
hyp_travel1(ori varchar(10), des varchar(10), time float) :=
  ASSUME
    (SELECT boat.ori, boat.des, boat.time - 0.5 FROM boat
        WHERE boat.des = 'VDE' AND boat.time > 1) IN link,
    (SELECT 'GMZ','VDE',1.5) NOT IN boat
  SELECT * FROM link UNION
  SELECT link.ori, hyp_travel1.des, link.time + hyp_travel1.time
    FROM link, hyp_travel1 WHERE link.des = hyp_travel1.ori;
```

This relation shows the expressive capabilities of the language by combining recursion and hypothetical assumptions within the same relation in a direct and natural way. In this case, by using the previously defined relation `travel` it is also possible to define the relation as:

```
hyp_travel2(ori varchar(10), des varchar(10), time float) :=
  ASSUME
    (SELECT boat.ori, boat.des, boat.time - 0.5 FROM boat
        WHERE boat.des = 'VDE' AND boat.time > 1) IN link,
    (SELECT 'GMZ','VDE',1.5) NOT IN boat
  SELECT * FROM travel;
```

Example 5. The relation `fib` of Example 2 can also be defined in HR-SQL by using the benefits of the ASSUME clause as follows:

```
fibev(n integer, f integer) := SELECT 0,1;
fibod(n integer, f integer) := SELECT 1,1;
fib(n integer, f integer) :=
  ASSUME
    SELECT fibev.n+2, fibev.f + fibod.f FROM fibev, fibod
      WHERE fibod.n = fibev.n +1 AND fibev.n+1<10 IN fibev,
    SELECT fibod.n+2, fibev.f + fibod.f FROM fibev, fibod
      WHERE fibev.n = fibod.n+1 AND fibod.n+1<10 IN fibod
  SELECT * FROM fibev UNION SELECT * FROM fibod;
```

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

This version reduces space and time computation, since Fibonacci numbers are not computed twice, and assumptions are stored only as temporary tables. [1]

The expressiveness of hypothetical relations will be even engaged when it is combined with aggregate functions as we will see in Section 5.

3. A stratified fixpoint semantics for HR-SQL

It is well-known that the combination of negation and recursion in database languages is a difficult task [1]. This problem has been tackled with stratified fixpoint semantics in several works [33,30,40], and this technique can also be applied to our proposal obtaining an operational semantics for HR-SQL. In this section we present a novel semantics of recursive and hypothetical SQL relations, based on stratified fixpoint interpretations, that formalizes the meaning of HR-SQL-databases, and we show how to compute such fixpoint.

Next, we introduce the notions of dependency graph and stratification that provide the basis for the stratified negation formalization we are looking for. Then, we define the concept of stratified interpretation, and prove the existence of the fixpoint of a continuous operator as the required interpretation of a database. The obtained semantics will be the basis of the implementation of a concrete HR-SQL database system.

From now on, we will use the following notations and terminology. $\mathtt{RN}$ stands for the set of relations names $\{\mathtt{R}_1, \ldots, \mathtt{R}_n\}$, that are defined in a database definition. We write $\mathtt{RN}_{\mathtt{query}}$ to denote the set of relation names occurring in a FROM clause inside `query`. More precisely:

- $\mathtt{RN}_{\mathtt{sel\_stm}}$ denotes the set of relation names occurring in the FROM clause of `sel_stm`.
- If $\mathtt{sel\_hyp} \equiv$ ASSUME `hypo query`, $\mathtt{RN}_{\mathtt{sel\_hyp}} = \mathtt{RN}_{\mathtt{query}}$.
- If $\mathtt{query} \equiv \mathtt{query}_1$ UNION $\mathtt{query}_2$, $\mathtt{RN}_{\mathtt{query}} = \mathtt{RN}_{\mathtt{query}_1} \cup \mathtt{RN}_{\mathtt{query}_2}$.
- If $\mathtt{query} \equiv \mathtt{query}_1$ EXCEPT $\mathtt{query}_2$, $\mathtt{RN}_{\mathtt{query}} = \mathtt{RN}_{\mathtt{query}_1} \cup \mathtt{RN}_{\mathtt{query}_2}$.

Notice that relations in the `hypo` part of a `sel_hyp` statement are not taken into account. In addition, for the case of the form $\mathtt{query} \equiv \mathtt{query}_1$ EXCEPT $\mathtt{query}_2$ we also define $\mathtt{RN}^{\neg}_{\mathtt{query}} = \mathtt{RN}_{\mathtt{query}_2}$ (notice that $\mathtt{RN}^{\neg}_{\mathtt{query}} \subseteq \mathtt{RN}_{\mathtt{query}}$). We say that `db` is a database definition over `RN` if for every `R sch:= query` defined in `db` it holds $\mathtt{R} \in \mathtt{RN}$, and every relation name used in `query` (including the `hypo` part) belongs to `RN`. When we speak about subqueries of `query` we do not refer to queries inside the `hypo` part of `query`.

3.1. Dependency graph and stratification

Stratification is based on the definition of a *dependency graph* for a database. The dependency graph associated to `db` establishes the dependencies between the database relations as follows.

Definition 1. Let `db` be a database definition over the set of relation names `RN`. The dependency graph associated to `db`, denoted by $DG_{\mathtt{db}}$, is a directed graph whose nodes are the elements of `RN`, and the edges, that can be *negatively labeled*, are determined as follows. A relation definition of the form `R sch := query` produces edges in the graph from every relation name belonging to $\mathtt{RN}_{\mathtt{query}}$ to `R`. Those edges produced by the relation names belonging to $\mathtt{RN}^{\neg}_{\mathtt{query}}$ are negatively labeled. In addition, every subquery of the form ASSUME `hypo query`, occurring inside the query defining a relation of `db`, also produces the following edges, due to the hypothetical assumptions:

- If $\mathtt{hypo} \equiv \mathtt{query}'$ IN $\mathtt{R}'$, and $\mathtt{R}_i \in \mathtt{RN}_{\mathtt{query}'}$, there is an edge from $\mathtt{R}_i$ to $\mathtt{R}'$, and if $\mathtt{R}_i \in \mathtt{RN}^{\neg}_{\mathtt{query}'}$, such edge is negatively labeled.
- If $\mathtt{hypo} \equiv \mathtt{query}'$ NOT IN $\mathtt{R}'$, and $\mathtt{R}_i \in \mathtt{RN}_{\mathtt{query}'}$, there is a negatively labeled edge from $\mathtt{R}_i$ to $\mathtt{R}'$.

For every two relations $\mathtt{R}_1, \mathtt{R}_2 \in \mathtt{RN}$, we say that $\mathtt{R}_2$ *depends* on $\mathtt{R}_1$ if there is a path from $\mathtt{R}_1$ to $\mathtt{R}_2$ in $DG_{\mathtt{db}}$. $\mathtt{R}_2$ *negatively depends* on $\mathtt{R}_1$ if there is a path from $\mathtt{R}_1$ to $\mathtt{R}_2$ in $DG_{\mathtt{db}}$ with at least one negatively labeled edge.

Definition 2. A *stratification* for a database definition `db`, over the set of relation names `RN`, is a mapping $str : \mathtt{RN} \to \{1, \ldots, n\}$, where $n = |\mathtt{RN}|$, such that:

- $str(\mathtt{R}_i) \leq str(\mathtt{R}_j)$, if $\mathtt{R}_j$ depends on $\mathtt{R}_i$,
- $str(\mathtt{R}_i) < str(\mathtt{R}_j)$ if $\mathtt{R}_j$ negatively depends on $\mathtt{R}_i$.

[1] A study of its performance can be found in an online appendix: http://gpd.sip.ucm.es/trac/gpd/raw-attachment/wiki/GpdSystems/HR-SQL2/AppendixB.pdf.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

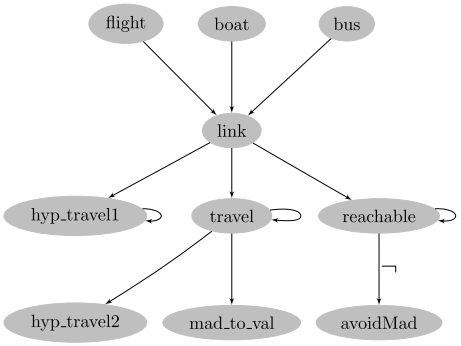


Fig. 3. $DG_{\texttt{db}}$ of Examples 3, 4.

db is *stratifiable* if there exists a stratification for it. In this case, for every R ∈ RN, we say that *str*(R) is the *stratum* of R. We denote by *str*(query) the maximum stratum of the relations of $\texttt{RN}_{\texttt{query}}$.

Intuitively, an EXCEPT operator plays the role of a negation in the deductive database field. A stratification-based solving procedure ensures that when a relation containing an EXCEPT in its definition is going to be calculated, the meaning of the relations involved in the right query of the EXCEPT clause have been completely evaluated, avoiding nonmonotonicity, as it is widely studied in Datalog [44]. We make a novel presentation of a formal semantics for our extended SQL language based on these techniques. Notice that Definition 1 extends the usual notion of dependency between relations, in order to deal with hypothesis.

Example 6. Consider the database of Examples 3 and 4. Its corresponding set of relation names is:

$$\texttt{RN} = \{\texttt{boat}, \texttt{bus}, \texttt{flight}, \texttt{link}, \texttt{travel}, \texttt{mad\_to\_val}, \texttt{reachable}, \texttt{avoidMad}, \texttt{hyp\_travel1}, \texttt{hyp\_travel2}\}$$

Its dependency graph is depicted in Fig. 3, where negatively labeled edges are annotated with ¬.

According to Definition 2 a stratification for this database can assign:

$$\textit{str}(\texttt{boat}) = \texttt{str}(\texttt{bus}) = \texttt{str}(\texttt{flight}) = 1,\ \textit{str}(\texttt{link}) = \texttt{str}(\texttt{travel}) = \texttt{str}(\texttt{mad\_to\_val}) = 1$$
$$\texttt{str}(\texttt{reachable}) = \texttt{str}(\texttt{hyp\_travel1}) = \texttt{str}(\texttt{hyp\_travel2}) = 1,\ \textit{str}(\texttt{avoidMad}) = 2.$$

Of course, this is not the only possible one; another one could be:

$$\textit{str}(\texttt{boat}) = 1,\ \textit{str}(\texttt{bus}) = 2,\ \textit{str}(\texttt{flight}) = 3,\ \textit{str}(\texttt{link}) = 4,\ \textit{str}(\texttt{reachable}) = 5,\ \textit{str}(\texttt{avoidMad}) = 6,$$
$$\textit{str}(\texttt{hyp\_travel1}) = 7,\ \textit{str}(\texttt{travel}) = 8,\ \textit{str}(\texttt{hyp\_travel2}) = 9\ \textit{str}(\texttt{mad\_to\_val}) = 10.$$

For efficiency reasons, this is the stratification as computed by the current implementation, as it will be explained in Section 6.1.

3.2. Stratified interpretations and fixpoint operator

From now on, we consider a fixed set RN with n relation names, a fixed stratification *str*, and we assume that every element R ∈ RN has a fixed schema. We denote by $\mathcal{DB}$ to the set of stratifiable database db over RN, and such that *str* is a stratification for db.

In the previous section, we established that the schema sch of a relation R is a sequence of type declarations for the attributes of R. In order to give meaning to this relation, we assume that every type T included in sch denotes a domain D. In previous examples we have used the types varchar for denoting the domain of strings, and integer and float for denoting numeric domains. We will consider a *universal domain* $\mathcal{D}$ which is the union of the family of the considered domains. A relation of arity k will denote a set of k-tuples included in $\mathcal{D}^k$. In general, every relation denotes a subset of $\mathcal{T} = \bigcup_{n\geq 1} \mathcal{D}^n$.

An interpretation for a database over RN is a function that associates an element of $\mathcal{P}(\mathcal{T})$ to each element of RN. So, considering the usual relational model terminology for schema and instance of a relation, the interpretation for a database definition db of a relation R, in our model, can be seen as the relationship between the schema and the instance. The intended meaning of db should be the interpretation for db which satisfies the definition of R in db, for every R ∈ RN. When the relations are not hypothetical, an interpretation can be seen as a function from RN to $\mathcal{P}(\mathcal{T})$. But when considering

hypothetical relations, it is convenient to define interpretations for the whole $\mathcal{DB}$, as functions applied to the elements of $\mathcal{DB}$, because a database definition db can locally change during the evaluation of a hypothetical query with respect to db. So, it is necessary that the value of an interpretation depends on the concrete element of $\mathcal{DB}$. We illustrate this idea with a simple case. Let sel_hyp ≡ ASSUME SELECT 1,0 IN R_a query; a hypothetical query for db. An interpretation I for db satisfies sel_hyp if it satisfies query, but with respect to a different database db′, with a new definition for R_a, which results from the union of the query defining R_a in db and the tuple 1,0. This implies that it is necessary to know not only the value of I for db, but also the value of I for db′. Another consideration is that in the stratified semantics we propose, interpretations are classified by strata. An interpretation of a stratum i for db gives values for the relations of db at strata less than or equal to i. Next, we formalize the concept of interpretation for the set of database definitions $\mathcal{DB}$ over a stratum.

Definition 3. An *interpretation* I for $\mathcal{DB}$, over the stratum i, $1 \leq i \leq n$ is a function $I : \mathcal{DB} \to (\mathtt{RN} \to \mathcal{P}(\mathcal{T}))$, such that, for each $\mathtt{db} \in \mathcal{DB}$ and each $\mathtt{R} \in \mathtt{RN}$:

- If R has schema $(\mathtt{A}_1\mathtt{T}_1, \ldots, \mathtt{A}_r\mathtt{T}_r)$, and $D_1, \ldots, D_r$ are, respectively, the domains denoted by $\mathtt{T}_1, \ldots, \mathtt{T}_r$, then $I(\mathtt{db})(\mathtt{R}) \subseteq D_1 \times \ldots \times D_r$.
- $I(\mathtt{db})(\mathtt{R}) = \emptyset$, if $str(\mathtt{R}) > i$.

The set of interpretations for $\mathcal{DB}$, over the stratum i, $1 \leq i \leq n$ is denoted by $\mathcal{I}_i^{\mathcal{DB}}$. The following definition provides an order on $\mathcal{I}_i^{\mathcal{DB}}$.

Definition 4. Let $i \geq 1$, and $I_1, I_2 \in \mathcal{I}_i^{\mathcal{DB}}$. I_1 *is less or equal than* I_2 *at stratum* i, denoted by $I_1 \sqsubseteq_i I_2$, if the following conditions are satisfied for every $\mathtt{db} \in \mathcal{DB}$ and every $\mathtt{R} \in \mathtt{RN}$:

- $I_1(\mathtt{db})(\mathtt{R}) = I_2(\mathtt{db})(\mathtt{R})$, if $str(\mathtt{R}) < i$.
- $I_1(\mathtt{db})(\mathtt{R}) \subseteq I_2(\mathtt{db})(\mathtt{R})$, if $str(\mathtt{R}) = i$.

It is straightforward to check that for any i, $1 \leq i \leq n$, the pair $(\mathcal{I}_i^{\mathcal{DB}}, \sqsubseteq_i)$ is a poset. The main question is that when an interpretation over a stratum i increases, for any database, the set of tuples associated to the relations whose stratum is i can increase, but the sets associated to relations of smaller strata remain invariable. In addition, this poset is a cpo, as it is proved in the following lemma.

Lemma 1. *For any* $i \geq 1$, *the pair* $(\mathcal{I}_i^{\mathcal{DB}}, \sqsubseteq_i)$ *is a complete partially ordered set. Moreover, if* $\{I_n\}_{n\geq 0}$ *is a chain of interpretations in* $(\mathcal{I}_i^{\mathcal{DB}}, \sqsubseteq_i)$, *then* $\hat{I}$, *defined as* $\hat{I}(\mathtt{db})(\mathtt{R}) = \bigcup_{n\geq 0} I_n(\mathtt{db})(\mathtt{R})$, *is the least upper bound of* $\{I_n\}_{n\geq 0}$.

Proof. It is easy to prove that $\hat{I} \in \mathcal{I}_i^{\mathcal{DB}}$, and that it is an upper bound. In addition, if I is another upper bound, this implies: If $str(\mathtt{R}) < i$, $I(\mathtt{db})(\mathtt{R}) = I_n(\mathtt{db})(\mathtt{R})$ for every $n \geq 0$, and hence $\hat{I}(\mathtt{db})(\mathtt{R}) = I(\mathtt{db})(\mathtt{R})$. If $str(\mathtt{R}) = i$, $I_n(\mathtt{db})(\mathtt{R}) \subseteq I(\mathtt{db})(\mathtt{R})$ for every $n \geq 0$, then $\bigcup_{n\geq 0} I_n(\mathtt{db})(\mathtt{R}) \subseteq I(\mathtt{db})(\mathtt{R})$, for every $\mathtt{db} \in \mathcal{DB}$ and every $\mathtt{R} \in \mathtt{RN}$. Therefore $\hat{I} \sqsubseteq_i I$, by the definition of $\sqsubseteq_i$.

Next we will formalize the meaning of any query in the context of a concrete interpretation I and a database db. The following notation will be useful: By $ev(\mathtt{exp}[\overline{\mathtt{v}}/\overline{\mathtt{e}}])$ we denote the evaluation of exp after applying the substitution $[\overline{\mathtt{v}}/\overline{\mathtt{e}}]$ to it. We simplify $(ev(\mathtt{exp}_1[\overline{\mathtt{v}}/\overline{\mathtt{e}}]), \ldots, ev(\mathtt{exp}_k[\overline{\mathtt{v}}/\overline{\mathtt{e}}]))$ by $ev((\mathtt{exp}_1, \ldots, \mathtt{exp}_k)[\overline{\mathtt{v}}/\overline{\mathtt{e}}])$. Analogously, by $ev(\mathtt{wcond}[\overline{\mathtt{v}}/\overline{\mathtt{e}}])$ we denote the Boolean evaluation of wcond, after applying the substitution $[\overline{\mathtt{v}}/\overline{\mathtt{e}}]$ to it.

Definition 5. Let $i \geq 1$, and $I \in \mathcal{I}_i^{\mathcal{DB}}$. We recursively define the *interpretation of* query *with respect to* I *for a database* $\mathtt{db} \in \mathcal{DB}$, denoted by $[\![\mathtt{query}]\!]_{\mathtt{db}}^I$, as:

- $[\![\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k]\!]_{\mathtt{db}}^I = \{(ev(\mathtt{exp}_1), \ldots, ev(\mathtt{exp}_k))\}$.
- $[\![\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k \text{ FROM } \mathtt{R}_1, \ldots, \mathtt{R}_m \text{ WHERE } \mathtt{wcond}]\!]_{\mathtt{db}}^I =$
  $\{(ev((\mathtt{exp}_1, \ldots, \mathtt{exp}_k)[\overline{\mathtt{a}}/\overline{\mathtt{A}}]) | \overline{\mathtt{a}} \in I(\mathtt{db})(\mathtt{R}_1) \times \ldots \times I(\mathtt{db})(\mathtt{R}_m)$ and $ev(\mathtt{wcond}[\overline{\mathtt{a}}/\overline{\mathtt{A}}])$ is satisfied$\}$.
  $\overline{\mathtt{A}}$ is a sequence of attributes labeled with their corresponding relation names. More precisely, if $\mathtt{A}_1^j, \ldots, \mathtt{A}_{r_j}^j$ are the attributes of $\mathtt{R}_j$, $1 \leq j \leq m$, then $\overline{\mathtt{A}}$ represents the complete sequence:
  $\mathtt{R}_1.\mathtt{A}_1^1, \ldots, \mathtt{R}_1.\mathtt{A}_{r_1}^1, \ldots, \mathtt{R}_m.\mathtt{A}_1^m \ldots \mathtt{R}_m.\mathtt{A}_{r_m}^m$.
- If $\mathtt{R\ sch\ :=\ query}_{\mathtt{R}}$ is the definition of R in db, then:
  - $[\![\text{ASSUME } \mathtt{query}' \text{ IN R query}]\!]_{\mathtt{db}}^I = [\![\mathtt{query}]\!]_{\mathtt{db}'}^I$,
    where $\mathtt{db}' = \mathtt{db}[\mathtt{R\ sch\ :=\ query}_{\mathtt{R}} \text{ UNION } \mathtt{query}'/\ \mathtt{R\ sch\ :=\ query}_{\mathtt{R}}]$. That is, db′ is the database db replacing $\mathtt{R\ sch\ :=\ query}_{\mathtt{R}}$ by the new definition of R, $\mathtt{R\ sch\ :=\ query}_{\mathtt{R}} \text{ UNION } \mathtt{query}'$.
    Notice that due to the definition of the dependencies derived from a relation definition containing a hypothetical query, $DG_{\mathtt{db}'} = DG_{\mathtt{db}}$, so the stratification of db is also a stratification of db′, hence $\mathtt{db}' \in \mathcal{DB}$.

- $[\![$ASSUME query′ NOT IN R query$]\!]^I_{db} = [\![$query$]\!]^I_{db'}$, where db′ = db[R sch := query_R EXCEPT query′/ R sch := query_R]. As before db′ ∈ $\mathcal{DB}$.
- $[\![\text{query}_1$ UNION $\text{query}_2]\!]^I_{db} = [\![\text{query}_1]\!]^I_{db} \cup [\![\text{query}_2]\!]^I_{db}$.
- $[\![\text{query}_1$ EXCEPT $\text{query}_2]\!]^I_{db} = [\![\text{query}_1]\!]^I_{db} \setminus [\![\text{query}_2]\!]^I_{db}$, where \ represents set difference.

Example 7. Let db be the following database definition (for simplicity, we omit the schema (A integer) for all the relations):

```
R1 := SELECT 1 UNION SELECT 2 UNION SELECT 3;
R2 := query_R2
    where query_R2 ≡ (SELECT 1 UNION SELECT 3 UNION SELECT 5)
                     EXCEPT (SELECT R1.A FROM R1 WHERE R1.A=1 OR R1.A=2);
R3 := SELECT R2.A FROM R2 UNION SELECT R3.A*2 FROM R3 WHERE R3.A<5;
```

Consider the following hypothetical query:

```
sel_hyp ≡  ASSUME
             SELECT R1.A FROM R1 WHERE R1.A < 3 IN R2, SELECT 3 NOT IN R2
           SELECT R3.A FROM R3
```

Remember that this query is a syntax sugar for nested hypothetical queries:

```
sel_hyp ≡  ASSUME SELECT R1.A FROM R1 WHERE R1.A < 3 IN R2
           (ASSUME SELECT 3 NOT IN R2
           SELECT R3.A FROM R3)
```

Let I be an interpretation. Then $[\![\text{sel_hyp}]\!]^I_{db} = [\![$SELECT R3.A FROM R3$]\!]^I_{db'}$, where:

$$\text{db}' = (\text{db})[\text{R2 := query}'_{\text{R2}}/\ \text{R2 := query}_{\text{R2}}]\ [\text{R2 := query}'_{\text{R2}} \text{ EXCEPT SELECT } 3/\text{R2 := query}'_{\text{R2}}]$$

$$\text{query}'_{\text{R2}} \equiv \text{query}_{\text{R2}} \text{ UNION SELECT R1.A FROM R1 WHERE R1.A < 3.}$$

Therefore db′ is the following database:

```
R1 := SELECT 1 UNION SELECT 2 UNION SELECT 3;

R2 := (((SELECT 1 UNION SELECT 3 UNION SELECT 5)
          EXCEPT SELECT R1.A FROM R1 WHERE R1.A=1 OR R1.A=2)
        UNION SELECT R1.A FROM R1 WHERE R1.A<3)
       EXCEPT SELECT 3;

R3 := SELECT R2.A FROM R2 UNION SELECT R3.A*2 FROM R3 WHERE R3.A<5;
```

Example 8. Let db be the database of Example 1, that defines the relations odd and even. Let us suppose a concrete interpretation I such that $I(\text{db})(\text{even}) = \{(0), (2)\}$ and $I(\text{db})(\text{odd}) = \emptyset$. Hence, the interpretation of the select statement that defines the relation odd in db with respect to I is:

$[\![$SELECT even.x+1 FROM even WHERE even.x<100$]\!]^I_{db}$ =
{(even.x+1)[a/even.x] |(a) ∈ I(db)(even) and (even.x<100) [a/even.x] is satisfied} =
{(1), (3)}.

The case of the relation even is analogous:

$[\![$SELECT 0 UNION SELECT odd.x+1 FROM odd$]\!]^I_{db}$ =
$[\![$SELECT 0$]\!]^I_{db} \cup [\![$SELECT odd.x+1 FROM odd$]\!]^I_{db}$ =
{(0)} ∪ {(odd.x+1)[a/odd.x] |(a) ∈ I(db)(odd)} = {(0)}.

Notice that the interpretation $\hat{I}$ such that:

$\hat{I}(\text{db})(\text{even}) = \{(0), (2), \ldots, (100)\}$ and $\hat{I}(\text{db})(\text{odd}) = \{(1), (3), \ldots, (99)\}$ satisfies:

$\hat{I}(\text{db})(\text{even}) = [\![$SELECT 0 UNION SELECT odd.x+1 FROM odd$]\!]^{\hat{I}}_{db}$
$\hat{I}(\text{db})(\text{odd}) = [\![$SELECT even.x+1 FROM even WHERE even.x<100$]\!]^{\hat{I}}_{db}$

The semantics of a database definition db ∈ $\mathcal{DB}$ will be formalized by means of an interpretation for $\mathcal{DB}$ I over the stratum n (the greater possible if RN has n elements), such that for every R ∈ RN, if R sch := query is the definition

of R in db, then $I(\mathtt{db})$ maps R to the set $[\![\mathtt{query}]\!]^I_{\mathtt{db}}$, as the interpretation $\hat{I}$ of Example 8 does. For every stratum i, the appropriate interpretation that gives the complete meaning to each relation of stratum i is the least fixpoint of a continuous operator over the set of interpretations of stratum i. These fixpoint interpretations are constructed sequentially from stratum 1 to n. The fixpoint of the last stratum of db provides the semantics for the whole database. Some technical lemmas are shown in order to ensure the existence of such fixpoint interpretations.

Next we prove some properties of the interpretation of select statements that are useful to ensure the existence of a fixpoint interpretation. The following lemma states that the sets of tuples denoted by a select statement of a stratum i, with respect to two ordered interpretations, satisfy an inclusion relation that is in accordance with the order $\sqsubseteq_i$ between the two interpretations.

Lemma 2. *Let $i \geq 1$, $I_1, I_2 \in \mathcal{I}_i^{\mathcal{DB}}$, such that $I_1 \sqsubseteq_i I_2$. For every* query, *every* $\mathtt{db} \in \mathcal{DB}$, *and every* $\mathtt{R} \in \mathtt{RN}$, *with* $str(\mathtt{R}) = i$, *if* $\mathtt{R}$ sch := $\mathtt{query}_{\mathtt{R}}$ *is the definition of* R *in* db *and* query *is a subquery of* $\mathtt{query}_{\mathtt{R}}$ *then it holds:*

- *If* $str(\mathtt{query}) < i$, *then* $[\![\mathtt{query}]\!]^{I_1}_{\mathtt{db}} = [\![\mathtt{query}]\!]^{I_2}_{\mathtt{db}}$.
- *If* $str(\mathtt{query}) = i$, *then* $[\![\mathtt{query}]\!]^{I_1}_{\mathtt{db}} \subseteq [\![\mathtt{query}]\!]^{I_2}_{\mathtt{db}}$.

Proof. The proof is inductive on the structure of query. Let $\mathtt{db} \in \mathcal{DB}$:

- SELECT $\mathtt{exp}_1, \ldots, \mathtt{exp}_k$.
  $[\![\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k]\!]^{I_1}_{\mathtt{db}} = [\![\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k]\!]^{I_2}_{\mathtt{db}} = \{(\mathtt{exp}_1, \ldots, \mathtt{exp}_k)\}$ independently of I_1 and I_2.
- SELECT $\mathtt{exp}_1, \ldots, \mathtt{exp}_k$ FROM $\mathtt{R}_1, \ldots, \mathtt{R}_l$ WHERE wcond.
  Assume $str(\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k \text{ FROM } \mathtt{R}_1, \ldots, \mathtt{R}_l \text{ WHERE wcond}) = i$, and let
  $(b_1, \ldots, b_k) \in [\![\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k \text{ FROM } \mathtt{R}_1, \ldots, \mathtt{R}_n \text{ WHERE wcond}]\!]^{I_1}_{\mathtt{db}}$,
  then there is $\overline{a} \in I_1(\mathtt{db})(\mathtt{R}_1) \times \ldots \times I_1(\mathtt{db})(\mathtt{R}_l)$, such that $ev(\mathtt{wcond}[\overline{a}/\overline{\mathtt{A}}])$ holds, and $ev(\mathtt{exp}_j[\overline{a}/\overline{\mathtt{A}}]) = b_j$, for $1 \leq j \leq k$. Since $I_1 \sqsubseteq_i I_2$, we have $I_1(\mathtt{db})(\mathtt{R}_1) \subseteq I_2(\mathtt{db})(\mathtt{R}_1), \ldots I_1(\mathtt{db})(\mathtt{R}_l) \subseteq I_2(\mathtt{db})(\mathtt{R}_l)$, because those relations belong to strata less or equal to i, therefore

  $$I_1(\mathtt{db})(\mathtt{R}_1) \times \ldots \times I_1(\mathtt{db})(\mathtt{R}_l) \subseteq I_2(\mathtt{db})(\mathtt{R}_l) \times \ldots \times I_2(\mathtt{db})(\mathtt{R}_l).$$

  Hence, we can conclude that $(b_1, \ldots, b_k)$ is also in $[\![$ SELECT $\mathtt{exp}_1, \ldots, \mathtt{exp}_k$ FROM $\mathtt{R}_1, \ldots, \mathtt{R}_n$ WHERE wcond$]\!]^{I_2}_{\mathtt{db}}$. The case $str(\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_k \text{ FROM } \mathtt{R}_1, \ldots, \mathtt{R}_l \text{ WHERE wcond}) < i$ is similar, because in this case $str(\mathtt{R}_j) < i$, $1 \leq j \leq l$, so $I_1(\mathtt{db})(\mathtt{R}_1) \times \ldots \times I_1(\mathtt{db})(\mathtt{R}_l) = I_2(\mathtt{db})(\mathtt{R}_1) \times \ldots \times I_2(\mathtt{db})(\mathtt{R}_l)$.
- $\mathtt{sel\_stm}_1$ UNION $\mathtt{sel\_stm}_2$.
  $[\![\mathtt{sel\_stm}_1 \text{ UNION } \mathtt{sel\_stm}_2]\!]^{I_1}_{\mathtt{db}} = [\![\mathtt{sel\_stm}_1]\!]^{I_1}_{\mathtt{db}} \bigcup [\![\mathtt{sel\_stm}_2]\!]^{I1}_{\mathtt{db}}$ and also $[\![\mathtt{sel\_stm}_1 \text{ UNION } \mathtt{sel\_stm}_2]\!]^{I_2}_{\mathtt{db}} = [\![\mathtt{sel\_stm}_1]\!]^{I_2} \bigcup [\![\mathtt{sel\_stm}_2]\!]^{I_2}_{\mathtt{db}}$. The induction hypothesis can be used. Notice that the stratum of $\mathtt{sel\_stm}_2$ and $\mathtt{sel\_stm}_1$ are less than or equal to i, so $[\![\mathtt{sel\_stm}_1]\!]^{I_1}_{\mathtt{db}} \subseteq [\![\mathtt{sel\_stm}_1]\!]^{I_2}_{\mathtt{db}}$ and $[\![\mathtt{sel\_stm}_2]\!]^{I_1}_{\mathtt{db}} \subseteq [\![\mathtt{sel\_stm}_2]\!]^{I_2}_{\mathtt{db}}$. The case $str(\mathtt{sel\_stm}_1 \text{ UNION } \mathtt{sel\_stm}_2) < i$ implies that $str(\mathtt{sel\_stm}_1) < i$ and $str(\mathtt{sel\_stm}_2) < i$. Hence, the equality we want to prove is directly obtained using the induction hypothesis.
- $\mathtt{query}_1$ EXCEPT $\mathtt{query}_2$.
  $[\![\mathtt{query}_1 \text{ EXCEPT } \mathtt{query}_2]\!]^{I_1}_{\mathtt{db}} = [\![\mathtt{query}_1]\!]^{I_1}_{\mathtt{db}} \setminus [\![\mathtt{query}_2]\!]^{I_1}_{\mathtt{db}}$. According to Definition 2 $str(\mathtt{query}_2) < i$, because we are assuming that $\mathtt{query}_1$ EXCEPT $\mathtt{query}_2$ is a subquery of the definition of R and $str(\mathtt{R}) \leq i$. Hence $[\![\mathtt{query}_2]\!]^{I_1}_{\mathtt{db}} = [\![\mathtt{query}_2]\!]^{I_2}_{\mathtt{db}}$, by the induction hypothesis. Similarly, it can be proved that $[\![\mathtt{query}_1]\!]^{I_1}_{\mathtt{db}} \subseteq [\![\mathtt{query}_1]\!]^{I_2}_{\mathtt{db}}$. Therefore $[\![\mathtt{query}_1 \text{ EXCEPT } \mathtt{query}_2]\!]^{I_1}_{\mathtt{db}} \subseteq [\![\mathtt{query}_1 \text{ EXCEPT } \mathtt{query}_2]\!]^{I_2}_{\mathtt{db}}$, with equality (instead of $\subseteq$) for the case $str(\mathtt{query}_1 \text{ EXCEPT } \mathtt{query}_2) < i$.
- ASSUME hypo query.
  $[\![\text{ASSUME hypo query}]\!]^{I_1}_{\mathtt{db}} = [\![\mathtt{query}]\!]^{I_1}_{\mathtt{db}'}$, and $[\![\text{ASSUME hypo query}]\!]^{I_2}_{\mathtt{db}} = [\![\mathtt{query}]\!]^{I_2}_{\mathtt{db}'}$, as specified in Definition 5. But, $[\![\mathtt{query}]\!]^{I_1}_{\mathtt{db}'} \subseteq [\![\mathtt{query}]\!]^{I_2}_{\mathtt{db}'}$ (with $[\![\mathtt{query}]\!]^{I_1}_{\mathtt{db}'} = [\![\mathtt{query}]\!]^{I_2}_{\mathtt{db}'}$, if $str(\mathtt{query}) < i$), by induction hypothesis. Therefore $[\![\text{ASSUME hypo query}]\!]^{I_1}_{\mathtt{db}} \subseteq [\![\text{ASSUME hypo query}]\!]^{I_2}_{\mathtt{db}}$, with equality (instead of $\subseteq$) if $str(\text{ASSUME hypo query}) < i$. □

The following lemma underlies the proof of the continuity of the operator whose fixpoint provides the semantics of a database.

Lemma 3. *Let $i \geq 1$, and $\{I_n\}_{n \geq 0}$ be a chain in $\mathcal{I}_i^{\mathcal{DB}}$. For every* query, *every* $\mathtt{db} \in \mathcal{DB}$, *and every* $\mathtt{R} \in \mathtt{RN}$, *with* $str(\mathtt{R}) = i$, *if* $\mathtt{R}$ sch := $\mathtt{query}_{\mathtt{R}}$ *is the definition of* R *in* db *and* query *is a subquery of* $\mathtt{query}_{\mathtt{R}}$ *it holds: If* $\hat{I} = \bigsqcup_{n \geq 0} I_n$, *then there exists* $k \geq 0$, *such that* $[\![\mathtt{query}]\!]^{\hat{I}}_{\mathtt{db}} = [\![\mathtt{query}]\!]^{I_k}_{\mathtt{db}}$.

Proof. Let $\mathtt{db} \in \mathcal{DB}$. The inclusion $\supseteq$ is a consequence of Lemma 2. The inclusion $\subseteq$ can be proved by induction on the structure of query. We show some cases:

- SELECT $\mathtt{exp}_1, \ldots, \mathtt{exp}_m$ FROM $\mathtt{R}_1, \ldots, \mathtt{R}_l$ WHERE wcond.
  Let $(b_1, \ldots, b_m) \in [\![\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_m \text{ FROM } \mathtt{R}_1, \ldots, \mathtt{R}_l \text{ WHERE wcond}]\!]^{\hat{I}}_{\mathtt{db}}$. Then, there is $\overline{a} \in \hat{I}(\mathtt{db})(\mathtt{R}_1) \times \ldots \times \hat{I}(\mathtt{db})(\mathtt{R}_l)$, such that $b_j = ev(\mathtt{exp}_j[\overline{a}/\overline{\mathtt{A}}])$, $1 \leq j \leq m$, and $ev(\mathtt{wcond}[\overline{a}/\overline{\mathtt{A}}])$ holds. According to Lemma 1, $\hat{I}(\mathtt{db})(\mathtt{R}_j) = \bigcup_{n\geq 0} I_n(\mathtt{db})(\mathtt{R}_j)$, $1 \leq j \leq l$. Therefore, it is easy to prove that there is a common $k \geq 0$, such that $\hat{I}(\mathtt{db})(\mathtt{R}_1) \times \ldots \times \hat{I}(\mathtt{db})(\mathtt{R}_l) \subseteq I_k(\mathtt{db})(\mathtt{R}_1) \times \ldots \times I_k(\mathtt{db})(\mathtt{R}_l)$. Therefore, $(b_1, \ldots, b_m) \in [\![\text{SELECT } \mathtt{exp}_1, \ldots, \mathtt{exp}_m \text{ FROM } \mathtt{R}_1, \ldots, \mathtt{R}_l \text{ WHERE wcond}]\!]^{I_k}_{\mathtt{db}}$.
- $\mathtt{query}_1$ EXCEPT $\mathtt{query}_2$.
  $[\![\mathtt{query}_1 \text{ EXCEPT } \mathtt{query}_2]\!]^{\hat{I}}_{\mathtt{db}} = [\![\mathtt{query}_1]\!]^{\hat{I}}_{\mathtt{db}} \setminus [\![\mathtt{query}_1]\!]^{\hat{I}}_{\mathtt{db}}$. By induction hypothesis, there exists $k \geq 0$, such that $[\![\mathtt{query}_1]\!]^{\hat{I}}_{\mathtt{db}} \subseteq [\![\mathtt{query}_1]\!]^{I_k}_{\mathtt{db}}$. On the other hand, we are assuming that $\mathtt{query}_1$ EXCEPT $\mathtt{query}_2$ occurs inside the definition of a relation of stratum i, so $str(\mathtt{query}_2) < i$, that implies $[\![\mathtt{query}_2]\!]^{\hat{I}}_{\mathtt{db}} = [\![\mathtt{query}_2]\!]^{I_k}_{\mathtt{db}}$, by Lemma 2. Hence, we can conclude that there is $k \geq 0$, such that $[\![\mathtt{query}_1 \text{ EXCEPT } \mathtt{query}_2]\!]^{\hat{I}}_{\mathtt{db}} \subseteq [\![\mathtt{query}_1]\!]^{I_n}_{\mathtt{db}} \setminus [\![\mathtt{query}_2]\!]^{I_k}_{\mathtt{db}} = [\![\mathtt{query}_1 \text{ EXCEPT } \mathtt{query}_2]\!]^{I_k}_{\mathtt{db}}$.
- ASSUME hypo query.
  $[\![\text{ASSUME hypo query}]\!]^{\hat{I}}_{\mathtt{db}} = [\![\mathtt{query}]\!]^{\hat{I}}_{\mathtt{db}'}$, where $\mathtt{db}'$ is as specified in Definition 5. There is $k \geq 0$, such that $[\![\mathtt{query}]\!]^{\hat{I}}_{\mathtt{db}'} = [\![\mathtt{query}]\!]^{I_k}_{\mathtt{db}'}$, by induction hypothesis. Notice that query is also a subquery of the definition of a relation of stratum i in $\mathtt{db}'$. $[\![\mathtt{query}]\!]^{I_k}_{\mathtt{db}'} = [\![\text{ASSUME hypo query}]\!]^{I_k}_{\mathtt{db}}$, according to Definition 5.

The remaining cases are straightforward.

Next, for every i, a continuous operator T_i over the set $\mathcal{I}_i^{\mathcal{DB}}$ of interpretations of stratum i is defined. Analogously to the theoretical foundations that support Datalog [44], we choose the least fixpoint of T_i, as the interpretation for $\mathcal{DB}$ over i that for every database $\mathtt{db} \in \mathcal{DB}$ will give meaning to the relation of stratum i defined in db. In accordance with the Knaster-Tarski theorem, this fixpoint can be obtained as the least upper bound of the chain of interpretations resulting by successively applying this operator to a minimal interpretation.

Definition 6. Let $1 \leq i \leq n$. The *operator* $T_i : \mathcal{I}_i^{\mathcal{DB}} \longrightarrow \mathcal{I}_i^{\mathcal{DB}}$ transforms interpretations over i as follows. For every $I \in \mathcal{I}_i^{\mathcal{DB}}$, every $\mathtt{db} \in \mathcal{DB}$, and $\mathtt{R} \in \mathtt{RN}$:

- $T_i(I)(\mathtt{db})(\mathtt{R}) = I(\mathtt{db})(\mathtt{R})$, if $str(\mathtt{R}) < i$.
- $T_i(I)(\mathtt{db})(\mathtt{R}) = [\![\mathtt{query}]\!]^{I}_{\mathtt{db}}$, if $str(\mathtt{R}) = i$ and `R sch := query` is the definition of R in db.
- $T_i(I)(\mathtt{db})(\mathtt{R}) = \emptyset$, if $str(\mathtt{R}) > i$.

This operator is proved to be monotone (it is a consequence of Lemma 2) and continuous for every i.

Lemma 4 *(Monotonicity of T_i). Let $i \geq 1$ and $I_1, I_2 \in \mathcal{I}_i^{\mathcal{DB}}$, such that $I_1 \sqsubseteq_i I_2$. Then, $T_i(I_1) \sqsubseteq_i T_i(I_2)$.*

Proof. Let $\mathtt{db} \in \mathcal{DB}$ and $\mathtt{R} \in \mathtt{RN}$. If $str(\mathtt{R}) < i$, then $I_1(\mathtt{R}) = I_2(\mathtt{R})$, because $I_1 \sqsubseteq_i I_2$. Hence, $T_i(I_1)(\mathtt{db})(\mathtt{R}) = T_i(I_2)(\mathtt{db})(\mathtt{R})$, by definition of T_i. If $str(\mathtt{R}) = i$, using the definition of T_i, $T_i(I_1)(\mathtt{db})(\mathtt{R}) = [\![\mathtt{query}_\mathtt{R}]\!]^{I_1}_{\mathtt{db}}$, $T_i(I_2)(\mathtt{db})(\mathtt{R}) = [\![\mathtt{query}_\mathtt{R}]\!]^{I_2}_{\mathtt{db}}$. Then, using Lemma 2 and the fact that $I_1 \sqsubseteq_i I_2$, we have that $[\![\mathtt{query}_\mathtt{R}]\!]^{I_1}_{\mathtt{db}} \subseteq [\![\mathtt{query}_\mathtt{R}]\!]^{I_2}_{\mathtt{db}}$. So, $T_i(I_1)(\mathtt{db})(\mathtt{R}) \subseteq T_i(I_2)(\mathtt{db})(\mathtt{R})$. We can conclude that $T_i(I_1) \sqsubseteq_i T_i(I_2)$.

Proposition 1 *(Continuity of T_i). Let $i \geq 1$ and $\{I_n\}_{n\geq 0}$ be a chain of interpretations in $\mathcal{I}_i^{\mathcal{DB}}$ ($I_0 \sqsubseteq_i I_1 \sqsubseteq_i I_2 \sqsubseteq_i \ldots$). Then, $T_i(\bigsqcup_{n\geq 0} I_n) =_i \bigsqcup_{n\geq 0} T_i(I_n)$.*

Proof. Let us prove $T_i(\bigsqcup_{n\geq 0} I_n) \sqsubseteq_i \bigsqcup_{n\geq 0} T_i(I_n)$. Let $\mathtt{db} \in \mathcal{DB}$ and $\mathtt{R} \in \mathtt{RN}$ defined in db as `R sch := query;`

- If $str(\mathtt{R}) < i$, then $T_i(\bigsqcup_{n\geq 0} I_n)(\mathtt{db})(\mathtt{R}) = \bigsqcup_{n\geq 0} I_n(\mathtt{db})(\mathtt{R})$, by the definition of T_i. Now, for every $n \geq 0$, $I_n(\mathtt{db})(\mathtt{R}) = T_i(I_n)(\mathtt{db})(\mathtt{R})$, also by definition of T_i. Therefore, $(T_i(\bigsqcup_{n\geq 0} I_n))(\mathtt{db})(\mathtt{R}) = (\bigsqcup_{n\geq 0} T_i(I_n))(\mathtt{db})(\mathtt{R})$.
- If $str(\mathtt{R}) = i$, then $T_i(\bigsqcup_{n\geq 0} I_n)(\mathtt{db})(\mathtt{R}) = [\![\mathtt{query}]\!]^{\bigsqcup_{n\geq 0} I_n}_{\mathtt{db}}$, by definition of T_i. In accordance with Lemma 3, for some $k \geq 0$: $[\![\mathtt{query}]\!]^{\bigsqcup_{n\geq 0} I_n}_{\mathtt{db}} \subseteq [\![\mathtt{query}]\!]^{I_k}_{\mathtt{db}}$. Now $[\![\mathtt{query}]\!]^{I_k}_{\mathtt{db}} = T_i(I_k)(\mathtt{db})(\mathtt{R})$, by definition of T_i, and obviously $T_i(I_k)(\mathtt{db})(\mathtt{R}) \subseteq \bigcup_{n\geq 0} T_i(I_n)(\mathtt{db})(\mathtt{R})$, but $\bigcup_{n\geq 0} T_i(I_n)(\mathtt{db})(\mathtt{R}) = (\bigsqcup_{n\geq 0} T_i(I_n))(\mathtt{db})(\mathtt{R})$, by Lemma 1. Hence, we conclude $T_i(\bigsqcup_{n\geq 0} I_n)(\mathtt{db})(\mathtt{R}) \subseteq (\bigsqcup_{n\geq 0} T_i(I_n))(\mathtt{db})(\mathtt{R})$.

The proof of $\bigsqcup_{n\geq 0} T_i(I_n) \sqsubseteq_i T_i(\bigsqcup_{n\geq 0} I_n)$ is a direct consequence of the monotonicity of T_i (Lemma 4).

Next, the expected result corresponding to the existence of least fixpoint stratum by stratum is shown.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

$T_8^n(fix_4)(\mathtt{db})(\mathtt{travel})$	Set of tuples
$T_8^1(fix_4)(\mathtt{db})(\mathtt{travel})$	{(TFS,GMZ,1.0),(MP,VDE,1.0),(TFN,TFS,2.5), (MAD,TFN,2.0),(SPC,TFN,2.0), (VDE,RES,1.0), (GMZ,VDE,1.5), (LPA,MP,2.0), (MAD,LPA,3.0)}
$T_8^2(fix_4)(\mathtt{db})(\mathtt{travel})$	{(MAD,TFS,4.5), (MP,RES,2.0), (GMZ,RES,2.5), (SPC,TFS,4.5), (LPA,VDE,3.0), (TFN,GMZ,3.5), (TFS,VDE,2.5), (MAD,MP,5.0)}
$T_8^3(fix_4)(\mathtt{db})(\mathtt{travel})$	{(MAD,GMZ,5.5), (SPC,GMZ,5.5), (LPA,RES,4.0), (TFS,RES,3.5),(MAD,VDE,6.0), (TFN,VDE,5.0)}
$T_8^4(fix_4)(\mathtt{db})(\mathtt{travel})$	{(MAD,RES,7.0), (MAD,VDE,7.0), (SPC,VDE,7.0), (TFN,RES,6.0)}
$T_8^5(fix_4)(\mathtt{db})(\mathtt{travel})$	{(SPC,RES,8.0), (MAD,RES,8.0)}

Fig. 4. Obtaining $fix_5(\mathtt{db})(\mathtt{travel})$.

Lemma 5. *The operator T_1 has a least fixpoint, which is $\bigsqcup_{n\geq 0} T_1^n(\emptyset)$, where $\emptyset : \mathcal{DB} \to (\mathtt{RN} \to \mathcal{P}(\mathcal{T}))$ is the interpretation such that for every $\mathtt{db} \in \mathcal{DB}$, $\emptyset(\mathtt{db})(\mathtt{R}) = \emptyset$ for every $\mathtt{R} \in \mathtt{RN}$.*

Proof. By the Knaster-Tarski fixpoint theorem [42], using Proposition 1.

We will denote $\bigsqcup_{n\geq 0}(T_1)^n(\emptyset)$ by fix_1, i.e., $fix_1(\mathtt{db})$ represents the least fixpoint at stratum 1 of the database db.

Consider now the sequence $\{T_2^n(fix_1)\}_{n\geq 0}$ of interpretations in $(\mathcal{I}_2^{\mathcal{DB}}, \sqsubseteq_2)$ greater than fix_1. Using the definition of T_i and the fact that $fix_1(\mathtt{db})(\mathtt{R}) = \emptyset$ for every $\mathtt{db} \in \mathcal{DB}$ and every R such that $str(\mathtt{R}) \geq 2$, it is easy to prove, by induction on $n \geq 0$, that this sequence is a chain:

$$fix_1 \sqsubseteq_2 T_2(fix_1) \sqsubseteq_2 T_2(T_2(fix_1)) \sqsubseteq_2 \ldots, \sqsubseteq_2 T_2^n(fix_1), \ldots$$

As before, in accordance with Proposition 1, $\{T_2^n(fix_1)\}_{n\geq 0}$ has a least upper bound, $\bigsqcup_{n\geq 0} T_2^n(fix_1)$, in $(\mathcal{I}_2^{\mathcal{DB}}, \sqsubseteq_2)$ that is the least fixpoint of T_2 containing fix_1. We denote this interpretation by fix_2.

By proceeding successively, for every i, $1 < i \leq n$, a chain:

$$fix_{i-1} \sqsubseteq_i T_i(fix_{i-1}) \sqsubseteq_i T_i(T_i(fix_{i-1})) \sqsubseteq_i \ldots \sqsubseteq_i T_i^n(fix_{i-1}) \ldots$$

can be defined, and a fixpoint of T_i, $fix_i = \bigsqcup_{n\geq 0} T_i^n(fix_{i-1})$, can be found.

Example 9. Using db to denote the database of Example 4 and the second stratification introduced in Example 6, Fig. 4 shows the tuples corresponding to the successive applications of the operator T_8 until $fix_5(\mathtt{db})(\mathtt{travel})$ is obtained.

Theorem 1. *There is a fixpoint interpretation $fix : \mathcal{DB} \to (\mathtt{RN} \to \mathcal{P}(\mathcal{T}))$, such that for every $\mathtt{db} \in \mathcal{DB}$ and every $\mathtt{R} \in \mathtt{RN}$, if* R sch := query *is the definition of* R *in* db, *then $fix(\mathtt{db})(\mathtt{R}) = [\![\mathtt{query}]\!]_{\mathtt{db}}^{fix}$.*

Proof. If k is the maximum value of the stratification, the interpretation fix we are looking for is fix_k, the least fixpoint of the operator T_k, applied to fix_{k-1}. As it has been pointed out, this fixpoint exists and verifies $fix_1 \sqsubseteq_k fix_2 \sqsubseteq_k \ldots \sqsubseteq_k fix_k$. Moreover, if $str(\mathtt{R}) = i$, $1 \leq i \leq k$, and it is defined by query in a database definition db, then $fix(\mathtt{db})(\mathtt{R}) = fix_i(\mathtt{db})(\mathtt{R}) = T_i(fix_i)(\mathtt{db})(\mathtt{R})$, because fix_i is the fixpoint of T_i. Now, $T_i(fix_i)(\mathtt{db})(\mathtt{R}) = [\![\mathtt{query}]\!]_{\mathtt{db}}^{fix_i}$, by definition of T_i. We can conclude $fix(\mathtt{db})(\mathtt{R}) = [\![\mathtt{query}]\!]_{\mathtt{db}}^{fix}$, trivially if $i = k$, or using Lemma 2, if $i < k$, because $fix_i \sqsubseteq_k fix$.

Therefore, fixed a stratification and the corresponding $\mathcal{DB}$, the interpretation fix defines the fixpoint semantics of every database definition $\mathtt{db} \in \mathcal{DB}$. This semantics is the support of the database system prototype we have implemented, which is described in Section 6.

Example 10. Consider the stratifiable database db and the hypothetical query sel_hyp of Example 7. Let *str* be a stratification for db, such that $str(\mathtt{R1}) = 1$, $str(\mathtt{R2}) = 2$, $str(\mathtt{R3}) = 3$. In this case, *str* is also a stratification for the modified database db′, obtained from db changing the definition of R2, the assumed relation of sel_hyp, with the corresponding assumptions, as it is detailed in Example 7. It is easy to check that:

$$fix(\mathtt{db})(\mathtt{R1}) = \{(1),(2),(3)\}, \quad fix(\mathtt{db})(\mathtt{R2}) = \{(3),(5)\}, \quad fix(\mathtt{db})(\mathtt{R3}) = \{(3),(5),(6)\}.$$

Now, in order to obtain the tuples satisfying $[\![\mathtt{sel\_hyp}]\!]_{\mathtt{db}}^{fix}$, it is necessary to know $fix(\mathtt{db}')$, because $[\![\mathtt{sel\_hyp}]\!]_{\mathtt{db}}^{fix} = [\![\text{SELECT R3.A FROM R3}]\!]_{\mathtt{db}'}^{fix}$. The computation of $fix(\mathtt{db}')$ stratum by stratum gives:

- For the stratum 1, the relation definitions coincide in db and db′. Hence $fix_1(\texttt{db}') = fix_1(\texttt{db})$.
- For stratum 2:
  $fix_2(\texttt{db}')(\texttt{R1}) = fix_1(\texttt{db}')(\texttt{R1})$, since $str(\texttt{R1}) < 2$,
  $fix_2(\texttt{db}')(\texttt{R3}) = \emptyset$, since $str(\texttt{R3}) > 2$,
  $fix_2(\texttt{db}')(\texttt{R2}) = \{(1), (2), (5)\}$.
- For stratum 3:
  $fix_3(\texttt{db}')(\texttt{R1}) = fix_1(\texttt{db}')(\texttt{R1})$, since $str(\texttt{R1}) = 1$,
  $fix_3(\texttt{db}')(\texttt{R2}) = fix_2(\texttt{db}')(\texttt{R2})$, since $str(\texttt{R2}) = 2$,
  $fix_3(\texttt{db}')(\texttt{R3}) = \{(1), (2), (4), (5), (8)\}$.

Then:
$[\![\texttt{sel\_hyp}]\!]^{fix}_{\texttt{db}} = [\![\textsc{select}\ \texttt{R3.A}\ \textsc{from}\ \texttt{R3}]\!]^{fix}_{\texttt{db}'} = fix_3(\texttt{db}')(\texttt{R3}) = \{(1), (2), (4), (5), (8)\}$.

Example 11. Consider that db is again the database of Examples 7 and 10, but extended with the hypothetical relation Rh defined below.

```
Rh := ASSUME SELECT R1.A FROM R1 WHERE R1.A < 3 IN R2,
                SELECT 3 NOT IN R2
      SELECT R3.A FROM R3 UNION
      SELECT Rh.A*3 FROM Rh WHERE Rh.A < 3;
```

Notice that Rh includes hypothesis as well as recursion. Consider an extension of the stratification of Example 10 with $str(\texttt{Rh}) = 4$. Then we have:

$$fix(\texttt{db})(\texttt{Rh}) = [\![\textsc{select}\ \texttt{R3.A}\ \textsc{from}\ \texttt{R3}\ \textsc{union}\ \textsc{select}\ \texttt{Rh.A*3}\ \textsc{from}\ \texttt{Rh}\ \textsc{where}\ \texttt{Rh.A<3}]\!]^{fix}_{\texttt{db}'}$$

where db′ is described in Example 7, but extended with the definition of Rh. For $1 \le i \le 3$, $fix_i(\texttt{db}')$ has been shown in Example 10. Now, $fix_4(\texttt{db}')(\texttt{Rh}) = (\bigsqcup_{m\ge 0}(T_4^m(fix_3))(\texttt{db}')(\texttt{Rh})$, obtaining:

$$T_4^1(fix_3)(\texttt{db}')(\texttt{Rh}) = fix_3(\texttt{db}')(\texttt{R3}) \cup \emptyset = \{(1), (2), (4), (5), (8)\}$$
$$T_4^2(fix_3)(\texttt{db}')(\texttt{Rh}) = \{(1), (2), (4), (5), (8)\} \cup \{(1*3), (2*3)\} = \{(1), (2), (3), (4), (5), (6), (8)\}$$
$$T_4^3(fix_3)(\texttt{db}')(\texttt{Rh}) = T_4^2(fix_3)(\texttt{db}')(\texttt{Rh})$$

Hence $fix_4(\texttt{db}')(\texttt{Rh}) = T_4^2(fix_3)(\texttt{db}')(\texttt{Rh})$.
Therefore: $fix(\texttt{db})(\texttt{Rh}) = \{(1), (2), (3), (4), (5), (6), (8)\}$.

Example 12. This example illustrates how the fixpoint computation preserves the locality of the assumptions. Two different relations R1 and R2 make assumptions over the same relation Ra. Then R3 is the Cartesian product of both.

```
Ra(A integer) := SELECT 0;
R1(A integer) := ASSUME SELECT 1 IN Ra SELECT * FROM Ra
R2(A integer) := ASSUME SELECT 2 IN Ra SELECT * FROM Ra
R3(A integer,A integer) := SELECT * FROM R1, R2
```

Notice that $fix(\texttt{db})(\texttt{R3}) = \{(0, 0), (1, 0), (1, 0), (1, 2)\}$, showing that when computing R3, the assumption in Ra to compute R1 is independent of the assumption in Ra to compute R2.

4. Computing hypothetical relations

Our goal is to have a relational database system that implements the fixpoint semantics of HR-SQL. In [4] a proof of concept system guided by the stratified fixpoint semantics for R-SQL (the subset of HR-SQL not containing ASSUME clauses) was presented. An improved implementation of that system was introduced in [5]. In order to extend the implementation of R-SQL to support hypothetical relations and queries, while maintaining soundness with respect to the theory, some consideration will be done.

The computation of the tuples corresponding to the different relations defined in a database db, requires to calculate $fix_i(\texttt{db})$ from $i = 1$ to *numstr* (the maximum stratum). When dealing with a hypothetical relation $\texttt{R}_h$ (of a stratum i) whose definition in db contains a hypothetical subquery sel_hyp, some extra-computation must be done because, according to the semantics of hypothetical queries, db should be locally modified to compute the hypothetical part sel_hyp, as we notice next.

During the computation of $fix_i(\texttt{db})(\texttt{R}_h)$, a limit for the following sequence, corresponding to the successive applications of the fixpoint operator at stratum i, must be found: $[\![\texttt{sel\_hyp}]\!]^{fix_{i-1}}_{\texttt{db}} \subseteq [\![\texttt{sel\_hyp}]\!]^{T_i(fix_{i-1})}_{\texttt{db}} \subseteq \ldots \subseteq [\![\texttt{sel\_hyp}]\!]^{T_i^n(fix_{i-1})}_{\texttt{db}} \ldots$
The information provided by $fix_{i-1}(\texttt{db})$ is not enough to compute this hypothetical part, because if for instance sel_hyp ≡ ASSUME query′ [NOT] IN $\texttt{R}_a$ query, then:

```
1: i := 1
2: while i ≤ numstr do
3:     while num_assum(i) ≠ 0 do
4:         db := tr(i, db)
5:     end while
6:     i := i + 1
7: end while
```

Fig. 5. Algorithm `transform`.

$$[\![\text{ASSUME } \texttt{query}' \text{ [NOT] IN } \mathtt{R}_a \texttt{ query}]\!]_{\mathtt{db}}^{T_i^n(fix_{i-1})} = [\![\texttt{query}]\!]_{\mathtt{db}'}^{T_i^n(fix_{i-1})}$$

which forces the computation of

$$fix_{i-1}(\mathtt{db}'), T_i(fix_{i-1}(\mathtt{db}')), \ldots, T_i^n(fix_{i-1})(\mathtt{db}') \ldots fix_i(\mathtt{db}')$$

where db′ is obtained from db by replacing the definition $\mathtt{R}_a$ $\mathtt{sch}_a$:= $\mathtt{query}_a$ by R $\mathtt{sch}_a$:= $\mathtt{query}_a$ (UNION | EXCEPT) query′ (from now on we use the notation (UNION | EXCEPT) as the alternative for IN and NOT IN cases, respectively, in order to explain these two similar cases together).

In practice, instead of locally computing each $fix_i(\mathtt{db}')$, corresponding to a hypothetical relation in db, during the computation of $fix_i(\mathtt{db})$, we have adopted a transformational approach. The initial database will be transformed into an equivalent one containing no hypothetical relations, but adding auxiliary relation definitions if necessary. Then, the fixpoint of the transformed database is computed stratum by stratum, as it is done for R-SQL databases.

We describe the general algorithm, called `transform`, that given an HR-SQL database definition db, obtains an equivalent R-SQL database. In the rest of the section we use the standard notation of one-hole context to denote the replacement of a subquery within a query. We use the symbols ⟨_⟩ instead of the usual [_] because the later have been used with other purposes in this paper. The notation query⟨_⟩ stands for query in which a subquery has been replaced with a hole; the notation query⟨query′⟩ denotes the application of the context to the query query′, i.e., the hole of query⟨_⟩ is filled with query′.

For simplicity of explanations, and without loss of generality, we suppose that $\mathtt{R}_h$ is defined in db as $\mathtt{R}_h$ $\mathtt{sch}_h$:= sel_hyp, where sel_hyp is a *hypothetical query in normal form*, i.e., sel_hyp ≡ ASSUME query′ [NOT] IN $\mathtt{R}_a$ query, and query′ does not contain any hypothetical subquery. Moreover, recall that it can be assumed that $\mathtt{R}_h \neq \mathtt{R}_a$, by the initial syntax assumptions. For the case that a hypothetical query sel_hyp′ occurs in query′, a new auxiliary relation R′ can be defined whose definition is sel_hyp′, and query′ is replaced by query′⟨select ∗ from R′⟩. This process is recursively applied to sel_hyp′ if it contains an assumption with some hypothetical subquery.

The generalized case, in which sel_hyp is a strict subquery of the definition ($\mathtt{query}_h$) of $\mathtt{R}_h$, where sel_hyp corresponds to a non-nested ASSUME, will be considered later in this section.

Taking advantage of the stratification, the transformation proposed can be done stratum by stratum as shown in Fig. 5. The expression *num_assum*(*i*) denotes the number of ASSUME clauses at stratum *i* in db. The main function *tr*(*i*, db) transforms db into a new database, where (at least) one ASSUME clause of a hypothetical relation $\mathtt{R}_h$ of the stratum *i* has been eliminated, and some new relation definitions are added. Some of these new relations are copies of existing ones. The total number of ASSUME clauses in the new database (without considering these copies) is less than in the original one. But considering copies, this number will increase when the application of the function *tr* introduces a copy of a hypothetical relation. So, the termination of this algorithm requires further elaboration that will be detailed later in this section.

In order to introduce the function *tr*, next we motivate its definition based on the semantics of hypothetical queries. As pointed out before, the tuples of $\mathtt{R}_h$ can be obtained by successively computing $[\![\texttt{query}]\!]_{\mathtt{db}'}^{T_i^n(fix_{i-1})}$, $n \geq 0$, where $\mathtt{db}' = \mathtt{db}[\mathtt{R}\ \mathtt{sch}_a$:= $\mathtt{query}_a$ (UNION | EXCEPT) query′/$\mathtt{R}_a$ $\mathtt{sch}_a$:= $\mathtt{query}_a]$. But notice that for computing $[\![\texttt{query}]\!]_{\mathtt{db}'}^{fix_i}$, it is only necessary to consider the relations (explicitly or implicitly) involved in query. In addition, $fix_i(\mathtt{db}')$ coincides with $fix_i(\mathtt{db})$ for those relations which do not depend on the assumed new relation $\mathtt{R}_a$.

Thus, let $S = \{\mathtt{R} \mid \mathtt{R}$ depends on $\mathtt{R}_a$ and ($\mathtt{R} \in \mathtt{RN}_{\mathtt{query}}$ or there is $\mathtt{R}' \in \mathtt{RN}_{\mathtt{query}}$ such that R′ depends on R$)\}$. Notice that *S* can be directly obtained from the dependency graph of db.

Let $\{\mathtt{R}_1, \ldots, \mathtt{R}_k\} = S \setminus \{\mathtt{R}_h, \mathtt{R}_a\}$. Let $\mathtt{R}'_1, \ldots \mathtt{R}'_k, \mathtt{R}'_a$, new identifiers not appearing in db. Let σ be the following replacement of relation names: $[\mathtt{R}'_1/\mathtt{R}_1], \ldots, [\mathtt{R}'_k/\mathtt{R}_k], [\mathtt{R}'_a/\mathtt{R}_a]$. Let $\mathtt{R}_j$ $\mathtt{sch}_j$:= $\mathtt{query}_j$ be the definition of $\mathtt{R}_j$, $j = 1 \,..\, k$, in db. Then, *tr*(*i*, db) is obtained from db as follows:

- Add the following relation definitions to db:
  $\mathtt{R}'_1$ $\mathtt{sch}_1$:= $\mathtt{query}_1\sigma$; ... $\mathtt{R}'_k$ $\mathtt{sch}_k$:= $\mathtt{query}_k\sigma$; $\mathtt{R}'_a$ $\mathtt{sch}_a$:= $\mathtt{query}_a\sigma$ (UNION | EXCEPT) query′σ ;
  with strata $str(\mathtt{R}'_j) := str(\mathtt{R}_j)$, $j = 1 \,..\, k$, $str(\mathtt{R}'_a) := str(\mathtt{R}_a)$
- Replace the definition $\mathtt{R}_h$ $\mathtt{sch}_h$:= sel_hyp by $\mathtt{R}_h$ $\mathtt{sch}_h$:= queryσ

The result of an application of $tr(i, \mathtt{db})$ appears schematized in Fig. 6.

We have introduced the transformation algorithm assuming some simplifications on the form of hypothetical queries for the sake of simplicity in the presentation. This algorithm can be easily extended for the general case that sel_hyp

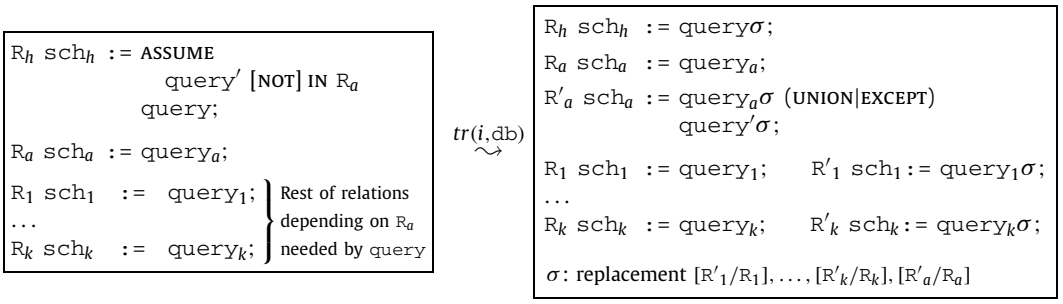


Fig. 6. Transformation function *tr*.

is a strict subquery of the definition ($\texttt{query}_h$) of $\texttt{R}_h$, where sel_hyp corresponds to a non-nested ASSUME, namely $\texttt{query}_h\langle\texttt{sel\_hyp}\rangle$ and sel_hyp is not in the scope of another ASSUME clause. Proceeding as before, the new definition of $\texttt{R}_h$ will be $\texttt{query}_h\langle\texttt{query}\sigma\rangle$. That means: replace sel_hyp by $\texttt{query}\sigma$ inside $\texttt{query}_h$. For instance, consider a relation R3 depending on R1 and

- $\texttt{query}_h\langle\texttt{sel\_hyp}\rangle \equiv$ (SELECT * FROM R1) EXCEPT
  ASSUME SELECT R2.A FROM R2 IN R1 (SELECT R1.A FROM R1 UNION SELECT * FROM R3)
  Then $\sigma = [\texttt{R1}'/\texttt{R1}][\texttt{R3}'/\texttt{R3}]$ and
- $\texttt{query}_h\langle\texttt{query}\sigma\rangle \equiv$ (SELECT * FROM R1) EXCEPT (SELECT R1'.A FROM R1' UNION SELECT * FROM R3').

Example 13. Consider again Example 7, and the hypothetical relation Rh of Example 11. Rh can be rewritten by using nested assumptions, as explained in Section 2.3, to:

```
Rh := ASSUME SELECT R1.A FROM R1 WHERE R1.A < 3 IN R2
      (ASSUME SELECT 3 NOT IN R2 (SELECT R3.A FROM R3 UNION
                                  SELECT Rh.A*3 FROM Rh WHERE Rh.A < 3));
```

The first stratum with a hypothetical relation is $str(\texttt{Rh}) = 4$. In order to transform Rh to remove the most external ASSUME, the corresponding set of relations to be considered is $S = \{\texttt{Rh}, \texttt{R2}, \texttt{R3}\}$.

The new relations, introduced by $tr(\texttt{db}, 4)$, will be:

```
R2' := ((SELECT 1 UNION SELECT 3 UNION SELECT 5)
        EXCEPT SELECT R1.A FROM R1 WHERE R1.A=1 OR R1.A=2)
       UNION SELECT R1.A FROM R1 WHERE R1.A<3;

R3' := SELECT R2'.A FROM R2' UNION
       (SELECT R3'.A*2 FROM R3' WHERE R3'.A < 5);
```

And the definition of Rh is replaced by the following one:

```
Rh := ASSUME SELECT 3 NOT IN R2'
      (SELECT R3'.A FROM R3' UNION
      SELECT Rh.A*3 FROM Rh WHERE Rh.A < 3);
```

Another application of *tr* is needed in order to remove the remaining ASSUME, obtaining the new relations:

```
R2'' := ((SELECT 1 UNION SELECT 3 UNION SELECT 5)
         EXCEPT SELECT R1.A FROM R1 WHERE R1.A=1 OR R1.A=2)
        UNION SELECT R1.A FROM R1 WHERE R1.A<3
       EXCEPT SELECT 3;

R3'' := SELECT R2''.A FROM R2'' UNION
        (SELECT R3''.A*2 FROM R3'' WHERE R3''.A < 5);
```

And replacing Rh by

```
Rh := SELECT R3''.A FROM R3'' UNION SELECT Rh.A*3 FROM Rh WHERE Rh.A < 3;
```

It is easy to prove that if $\texttt{db}_t$ is the final database, resulting after this transformation process, then:

$fix(\texttt{db}_t)(\texttt{Rh}) = fix(\texttt{db})(\texttt{Rh}) = \{(1), (2), (3), (4), (5), (6), (8)\}$. In addition:

$fix_2(\texttt{db}_t)(\texttt{R2}'') = \{(1), (2), (5)\}$, $fix_3(\texttt{db}_t)(\texttt{R3}'') = \{(1), (2), (4), (5), (8)\}$.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

Thus, confronting Example 11: $fix_2(\mathtt{db}_t)(\mathtt{R2}'') = fix_2(\mathtt{db}')(\mathtt{R2})$, $fix_3(\mathtt{db}_t)(\mathtt{R3}'') = fix_3(\mathtt{db}')(\mathtt{R3})$, where (see Example 7) $\mathtt{db}'$ is the initial database db replacing the definition of R2 ($\mathtt{query}_{\mathtt{R2}}$) by the definition of R2″:

```
query_R2 UNION SELECT R1.A FROM R1 WHERE R1.A<3 EXCEPT SELECT 3;
```

Example 14. As no restrictions are imposed on the mutual recursion between hypotheses, the algorithm also flattens this kind of relations, as shown in this example. Consider the following mutual recursive and hypothetical relations:

```
R1(a integer) := ASSUME SELECT 0 IN R2 SELECT * FROM R2;
R2(a integer) := ASSUME SELECT 1 IN R1 SELECT * FROM R1;
```

This database definition is transformed into the following equivalent one:

```
R1(a integer) := SELECT * FROM R2';
R2(a integer) := SELECT * FROM R1';
R2'(a integer) := SELECT * FROM R1" UNION SELECT 0;
R1'(a integer) := SELECT * FROM R2' UNION SELECT 1;
R1"(a integer) := SELECT * FROM R2' UNION SELECT 1;
```

The result for both R1 and R2 will be the expected one $\{(0), (1)\}$.

A more general formulation of this kind of mutual recursion as:

```
R1(a integer) := ASSUME query_1 IN R2 SELECT * FROM R2;
R2(a integer) := ASSUME query_2 IN R1 SELECT * FROM R1;
```

will result in the union of the responses to query_1 and query_2

4.1. Soundness and completeness of the transformation algorithm

In this section we prove that this algorithm always finishes with an stratifiable database without assumes that is equivalent to the initial one. Moreover, we study the complexity of the algorithm.

The algorithm transform **finishes.** To prove termination of the algorithm, we must define an appropriate termination ordering. Notice that the main procedure $tr(i, \mathtt{db})$ eliminates an ASSUME clause at each application, but it introduces a number of new relation definitions that can contain ASSUME clauses (if other hypothetical relations were in the stratum in process, which means there is a mutual recursion between them), so the proof of termination needs some elaboration. The key idea for ensuring termination is that the new relations are either copies of existing ones, or they rename a relation $\mathtt{R}_a$ involved in the hypothesis of an assumption. In the last case, the definition of the renaming $\mathtt{R}'_a$ is an extension of the original definition of $\mathtt{R}_a$ with a query that, by hypothesis, does not contain any ASSUME clause. So, in any case the new relations have the same number of ASSUME clauses that the relation from which they come. In addition, the new relations are incorporated to the stratum of their corresponding original relations

To define the termination ordering, suppose that the ASSUME clauses of the relations in the first $i-1$ strata have been eliminated, and the tr function is now applied to the stratum i. As pointed out before, the process for stratum i does not introduce any hypothetical relation in previous strata, because the new relations are at the same stratum of the original ones and have the same number of ASSUME's of them. Suppose that there are n hypothetical relations $\mathtt{R}_1, \ldots, \mathtt{R}_n$ at stratum i (the order is arbitrary). Consider the tuple of sets $T_0 = (\{\mathtt{R}_1\}, \ldots, \{\mathtt{R}_n\})$, and let $K_0 = (k_1^0, \ldots, k_n^0)$ be a tuple such that each k_j^0, $j = 1..n$, represents the number of ASSUME clauses of $\mathtt{R}_j$. Without loss of generality, we can consider that the ASSUME elimination algorithm proceeds from left to right in the tuple T_0. In the first application of tr, the outermost ASSUME clause of $\mathtt{R}_1$ is eliminated, but also a renaming $\mathtt{R}'_j$, for certain relations $\mathtt{R}_j$, $j = 2..n$, can be introduced. Then let $T_1 = (R_1^1, \ldots, R_n^1)$, where R_1^1 contains the redefinition of $\mathtt{R}_1$, and every R_j^1, $j = 2..n$, is the set containing $\mathtt{R}_j$ and $\mathtt{R}'_j$, if some renaming of it is introduced. In general, after m applications of tr, we have the tuple of sets $T_m = (R_1^m, \ldots, R_n^m)$, where R_j^m, $j = 1..n$, represents the set containing $\mathtt{R}_j$ and the new relations $\mathtt{R}'_j$, $\mathtt{R}''_j$, ..., that successively come from it, after the m of applications of tr. Let $K_m = (k_1^m, \ldots, k_n^m)$ be the tuple whose components represent, respectively, the number of ASSUME's of each set of T_m. Let R_j^m, be the first set in T_m with some hypothetical relation, then $k_1^m, \ldots, k_{j-1}^m = 0$, and $k_j^m, \ldots, k_n^m > 0$. A new application of tr for eliminating an ASSUME of a relation in R_j^m can introduce renamings on the rest of the sets. These new relations can not have any ASSUME if they correspond to the sets $R_1^m, \ldots, R_{j-1}^m$, then the number of ASSUME clauses can only be increased in the sets $R_{j+1}^m, \ldots, R_n^m$. This means that the tuple $K_{m+1} = (k_1^{m+1}, \ldots, k_n^{m+1})$ corresponding to the $m+1$ application of tr, is such that $k_1^{m+1}, \ldots, k_{j-1}^{m+1} = 0$, $k_j^{m+1} = k_j^m - 1$, and $k_{j+1}^{m+1} \geq k_{j+1}^m, \ldots, k_n^{m+1} \geq k_n^m$. Then a lexicographic ordering on the tuples $K_0, K_1, \ldots, K_m, \ldots$ allows to prove termination of the algorithm.

The result of transform **is an R-SQL stratifiable database.** On the one hand, to prove that the new database is stratifiable, notice that in accordance with Definition 1 (in particular using the dependencies generated by sel_hyp), the strata

assigned by *tr* to the new relations, correctly extends *str* to a stratification of the new database. In fact, the final database is defined over a set of relation names $\mathtt{RN}'$ such that $\mathtt{RN} \subseteq \mathtt{RN}'$. In order to simplify the notation, from now on we assume that $\mathcal{DB}$ is extended to represent the set of database definitions over $\mathtt{RN}'$ stratifiable with *str*.

On the other hand, it is obvious that the result is an R-SQL database, because we have proved termination, and the algorithm finishes when no ASSUME clause remains. Then, both the initial and final databases of the transformation algorithm belong to $\mathcal{DB}$.

Preservation of the semantics. Now we prove that the resulting database, without hypothetical definitions, is equivalent to the initial one.

The following lemma is useful to prove correction. It relates the renaming relation of the new database to the original ones.

Lemma 6. *Let* db *be an HR-SQL database definition over the set of relation names* RN*. Let* σ *be a renaming of some elements of* RN*, using fresh names,* $\sigma = [\mathtt{R}'_1/\mathtt{R}_1], \ldots, [\mathtt{R}'_k/\mathtt{R}_k]$*. Assume that* $str(\mathtt{R}'_j) = str(\mathtt{R}_j)$, $j = 1..k$*. Let* db′ *be an HR-SQL database definition over* $\mathtt{RN}' = \mathtt{RN} \cup \{\mathtt{R}'_1, \ldots, \mathtt{R}'_k\}$*, extending* db *as follows:*

$$\mathtt{db}' = \mathtt{db} \cup \mathtt{R}_1\sigma \texttt{ sch := query}_1\sigma \texttt{ ; } \ldots \mathtt{R}_k\sigma \texttt{ sch := query}_k\sigma \texttt{ ;}$$

Then it holds: $fix(\mathtt{db})(\mathtt{R}) = fix(\mathtt{db}')(\mathtt{R}\sigma)$, *for every* $\mathtt{R} \in \mathtt{RN}$.

Proof. It is obvious that if $\mathtt{db}\sigma$ is just a renaming of db, then for every query using relation names of RN, it holds: $[\![\mathtt{query}]\!]^{fix}_{\mathtt{db}} = [\![\mathtt{query}\sigma]\!]^{fix}_{\mathtt{db}\sigma}$. The proof of the lemma is straightforward from this fact.

The next lemmas show technical properties, that will be used below to establish the connection between the formal semantics of the hypothetical relation $\mathtt{R}_h$ of the original database, and the semantics of this relation in the transformed database. Regarding the semantics of the original $\mathtt{R}_h$, for each fixpoint iteration, the definition of the relation $\mathtt{R}_a$ involved in the hypothesis is replaced and the original definition of $\mathtt{R}_h$ is preserved. However, in the transformed database, the ASSUME clause of $\mathtt{R}_h$ is eliminated and a renaming of $\mathtt{R}_a$, instead of a replacement, is considered. The following definitions will facilitate the formulation of the mentioned lemmas.

Definition 7. We say that two queries $\mathtt{query}_1$, $\mathtt{query}_2$ are *equivalent* if for every interpretation I over $\mathcal{DB}$ and every database definition $\mathtt{db} \in \mathcal{DB}$, $[\![\mathtt{query}_1]\!]^I_{\mathtt{db}} = [\![\mathtt{query}_2]\!]^I_{\mathtt{db}}$. Let db, $\mathtt{db}' \in \mathcal{DB}$, we say that they are *equivalent* if for every relation $\mathtt{R} \in \mathtt{RN}$, defined as R sch := query and R sch := query′ in db and db′, respectively, it holds that query and query′ are equivalent. We say that db and db′ are *fixpoint equivalent* if $fix(\mathtt{db}) = fix(\mathtt{db}')$.

An interpretation I over $\mathcal{DB}$ is *coherent for equivalent databases* if for every two equivalent databases $\mathtt{db}, \mathtt{db}' \in \mathcal{DB}$, it holds $I(\mathtt{db}) = I(\mathtt{db}')$.

Definition 8. Let $\mathtt{R}_h$ be a hypothetical relation. We say that two databases db, $\mathtt{db}' \in \mathcal{DB}$ are *equal but assumption on* $\mathtt{R}_h$, denoted by $\mathtt{db} \approx_{\mathtt{R}_h} \mathtt{db}'$, if:

- For every $\mathtt{R} \in \mathtt{RN} \setminus \{\mathtt{R}_h\}$, the definition of R in db is equal to the definition of R in db′.
- The definition of $\mathtt{R}_h$ in db is $\mathtt{query}_h\langle$ASSUME query′ [NOT] IN $\mathtt{R}_a$ query$\rangle$, while the definition of $\mathtt{R}_h$ in db′ is $\mathtt{query}_h\langle\mathtt{query}\rangle$, where the argument of the context $\mathtt{query}_h\langle\ \rangle$ is not under the scope of an ASSUME clause.
- The definition of $\mathtt{R}_a$ in db and db′ is equivalent to $\mathtt{query}_a$(UNION|EXCEPT) query′, for some $\mathtt{query}_a$.

An interpretation I over $\mathcal{DB}$ is $\mathtt{R}_h$*-coherent* if for every $\mathtt{db}, \mathtt{db}' \in \mathcal{DB}$ with $\mathtt{db} \approx_{\mathtt{R}_h} \mathtt{db}'$ it holds $I(\mathtt{db}) = I(\mathtt{db}')$.

The intended meaning of $\mathtt{db} \approx_{\mathtt{R}_h} \mathtt{db}'$ is that $\mathtt{R}_h$ is an hypothetical relation in db, whose definition contains ASSUME query′ [NOT] IN $\mathtt{R}_a$ query, as the outermost hypothetical subquery, and this hypothetical subquery is replaced by query in the definition of $\mathtt{R}_h$ in db′. In addition the definition of $\mathtt{R}_a$ in both databases is equivalent to $\mathtt{query}_a$ UNION query′ or $\mathtt{query}_a$ EXCEPT query′, if the hypo part of the hypothetical subquery of $\mathtt{R}_h$ in db is query′ IN $\mathtt{R}_a$ or query′ NOT IN $\mathtt{R}_a$, respectively.

Lemma 7. *Let* I *be an interpretation over* $\mathcal{DB}$*, if it is coherent for equivalent databases, then for every* db, $\mathtt{db}' \in \mathcal{DB}$ *equivalent databases, and for every* query *it holds* $[\![\mathtt{query}]\!]^I_{\mathtt{db}} = [\![\mathtt{query}]\!]^I_{\mathtt{db}'}$.

Proof. By induction on the structure of query. We prove the non trivial case of a hypothetical query ASSUME query′ [NOT] IN $\mathtt{R}_a$ query. The induction hypothesis states that $[\![\mathtt{query}]\!]^I_{\mathtt{db}} = [\![\mathtt{query}]\!]^I_{\mathtt{db}'}$, for every db, $\mathtt{db}' \in \mathcal{DB}$ equivalent databases. Let db and db′ equivalent databases. $[\![$ASSUME query′ IN $\mathtt{R}_a$ query$]\!]^I_{\mathtt{db}} = [\![\mathtt{query}]\!]^I_{\mathtt{db}_1}$, and $[\![$ASSUME query′ IN $\mathtt{R}_a$ query$]\!]^I_{\mathtt{db}'} = [\![\mathtt{query}]\!]^I_{\mathtt{db}_2}$, where:

$\mathtt{db}_1 = \mathtt{db}[\mathtt{R}_a\ \mathtt{sch}_a := \mathtt{query}_a\ \textsc{union}\ \mathtt{query}'/\mathtt{R}_a\ \mathtt{sch}_a := \mathtt{query}_a]$.
$\mathtt{db}_2 = \mathtt{db}'[\mathtt{R}_a\ \mathtt{sch}_a := \mathtt{query}'_a\ \textsc{union}\ \mathtt{query}'/\mathtt{R}_a\ \mathtt{sch}_a := \mathtt{query}'_a]$.

It is easy to see that $\mathtt{db}_1$ and $\mathtt{db}_2$ are equivalent as well, because $\mathtt{query}_a$ and $\mathtt{query}'_a$ are equivalent by hypothesis, since db, db′ are equivalent. Hence, applying the induction hypothesis, $[\![\mathtt{query}]\!]^I_{\mathtt{db}_1} = [\![\mathtt{query}]\!]^I_{\mathtt{db}_2}$, which concludes the proof of this case. If the assumption is NOT IN $\mathtt{R}_a$, the proof is analogous.

Lemma 8. *Let I be an interpretation over $\mathcal{DB}$, and let $\mathtt{R}_h$ be a hypothetical relation, if I is $\mathtt{R}_h$-coherent, then for every* db, db′ ∈ $\mathcal{DB}$ *with* db $\approx_{\mathtt{R}_h}$ db′, *and for every* query *it holds* $[\![\mathtt{query}]\!]^I_{\mathtt{db}} = [\![\mathtt{query}]\!]^I_{\mathtt{db}'}$.

Proof. By induction on the structure of query. For the base case, notice that for every db, db′ with db $\approx_{\mathtt{R}_h}$ db′, and every R, $I(\mathtt{db})(\mathtt{R}) = I(\mathtt{db}')(\mathtt{R})$. The non trivial recursive case refers to a query including an assumption over $\mathtt{R}_h$, say ASSUME $\mathtt{query}_1$ [NOT] IN $\mathtt{R}_h$ $\mathtt{query}_2$. The induction hypothesis states that $[\![\mathtt{query}]\!]^I_{\mathtt{db}} = [\![\mathtt{query}]\!]^I_{\mathtt{db}'}$, for every db, db′ databases equal but assumption on $\mathtt{R}_h$. Let db, db′ ∈ $\mathcal{DB}$ with db $\approx_{\mathtt{R}_h}$ db′. Let $\mathtt{query}_h$ be the definition of $\mathtt{R}_h$ in db, containing $\mathtt{sel\_hyp} \equiv$ ASSUME $\mathtt{query}'$ [NOT] IN $\mathtt{R}_a$ query, and let $\mathtt{query}'_h$ be the definition of $\mathtt{R}_h$ in db′, containing query instead of sel_hyp. $[\![\textsc{assume}\ \mathtt{query}_1\ [\textsc{not}]\ \textsc{in}\ \mathtt{R}_h\ \mathtt{query}_2]\!]^I_{\mathtt{db}} = [\![\mathtt{query}_2]\!]^I_{\mathtt{db}_1}$, and $[\![\textsc{assume}\ \mathtt{query}_1\ [\textsc{not}]\ \textsc{in}\ \mathtt{R}_h\ \mathtt{query}_2]\!]^I_{\mathtt{db}'} = [\![\mathtt{query}_2]\!]^I_{\mathtt{db}_2}$, where:

$\mathtt{db}_1 = \mathtt{db}[\mathtt{R}_h\ \mathtt{sch}_h := \mathtt{query}_h(\textsc{union}|\textsc{except})\ \mathtt{query}_1/\mathtt{R}_h\ \mathtt{sch}_h := \mathtt{query}_h]$.
$\mathtt{db}_2 = \mathtt{db}'[\mathtt{R}_h\ \mathtt{sch}_h := \mathtt{query}'_h(\textsc{union}|\textsc{except})\ \mathtt{query}_1/\mathtt{R}_h\ \mathtt{sch}_h := \mathtt{query}'_h]$.

Again $\mathtt{db}_1$ and $\mathtt{db}_2$ are equal but assumption on $\mathtt{R}_h$. Hence, applying the induction hypothesis, $[\![\mathtt{query}_2]\!]^I_{\mathtt{db}_1} = [\![\mathtt{query}_2]\!]^I_{\mathtt{db}_2}$, which concludes the proof.

Lemma 9. *Let* db, db′ ∈ $\mathcal{DB}$ *be equivalent databases. Then, for every* query *it holds* $[\![\mathtt{query}]\!]^{fix}_{\mathtt{db}} = [\![\mathtt{query}]\!]^{fix}_{\mathtt{db}'}$.

Proof. In order to prove the lemma we prove that fix is coherent for equivalent databases, then the result follows by applying Lemma 7.

It can be proved that for every $i \geq 1$, it holds fix_i is coherent for equivalent databases. We use induction on i. Assuming that fix_{i-1} is coherent for equivalent databases, we prove this fact for fix_i, showing that for every $n \geq 0$, $T_i^n(fix_{i-1})$ is coherent for equivalent databases, which can be proved by induction on the number of iterations n.

The base case is trivial by hypothesis.

For the inductive step, let $n \geq 0$, we call $I^n = T_i^n(fix_{i-1})$, and assume that I^n is coherent for equivalent databases. Let db, db′ equivalent databases, and R ∈ RN, we prove that $T_i(I^n)(\mathtt{db})(\mathtt{R}) = T_i(I^n)(\mathtt{db}')(\mathtt{R})$.

If $\mathtt{query}_{\mathtt{R}}$ is the definition of R in db, and $\mathtt{query}'_{\mathtt{R}}$ is the definition of R in db′, then:

$$\begin{array}{rcll} T_i(I^n)(\mathtt{db})(\mathtt{R}) & = & [\![\mathtt{query}_{\mathtt{R}}]\!]^{I^n}_{\mathtt{db}} & \text{by definition of } T_i, \\ & = & [\![\mathtt{query}'_{\mathtt{R}}]\!]^{I^n}_{\mathtt{db}} & \text{by equivalence}, \\ & = & [\![\mathtt{query}'_{\mathtt{R}}]\!]^{I^n}_{\mathtt{db}'} & \text{by the induction hypothesis and Lemma 7}, \\ & = & T_i(I^n)(\mathtt{db}')(\mathtt{R}) & \text{by definition of } T_i. \end{array}$$

The argument for the base case $fix_1(\mathtt{db}) = fix_1(\mathtt{db}')$ is similar. Notice that in this case the assumption $\bot(\mathtt{db}) = \bot(\mathtt{db}')$ is obviously true.

Lemma 10. *Let* db, db′ ∈ $\mathcal{DB}$ *with* db $\approx_{\mathtt{R}_h}$ db′. *Then, for every* query *it holds* $[\![\mathtt{query}]\!]^{fix}_{\mathtt{db}} = [\![\mathtt{query}]\!]^{fix}_{\mathtt{db}'}$.

Proof. This proof is similar to the proof of Lemma 9. We prove that fix is $\mathtt{R}_h$-coherent, then the result can be deduced applying Lemma 8.

Let $i = str(\mathtt{R}_h)$. We prove that for every $n \geq 0$ $T_i^n(fix_{i-1})$ is $\mathtt{R}_h$-coherent, by induction on n.

The base case is trivial, because if db and db′ verify db $\approx_{\mathtt{R}_h}$ db′, they coincide for strata 1 to $i-1$, so for every R ∈ RN, $fix_{i-1}(\mathtt{db})(\mathtt{R}) = fix_{i-1}(\mathtt{db}')(\mathtt{R})$. For the inductive case, assume that $I^n = T_i^n(fix_{i-1})$ is $\mathtt{R}_h$-coherent. Let us prove that for every db, db′ equal but assumption on $\mathtt{R}_h$, it holds $T_i(I^n)(\mathtt{db})(\mathtt{R}) = T_i(I^n)(\mathtt{db}')(\mathtt{R})$, for every R ∈ RN.

- If $\mathtt{R} \neq \mathtt{R}_h$, the definition of R in db, $\mathtt{query}_{\mathtt{R}}$, coincides with the definition of R in db′, therefore:

$$\begin{array}{rcll} T_i(I^n)(\mathtt{db})(\mathtt{R}) & = & [\![\mathtt{query}_{\mathtt{R}}]\!]^{I^n}_{\mathtt{db}} & \text{by definition of } T_i, \\ & = & [\![\mathtt{query}_{\mathtt{R}}]\!]^{I^n}_{\mathtt{db}'} & \text{by the induction hypothesis and Lemma 8}, \\ & = & T_i(I^n)(\mathtt{db}')(\mathtt{R}) & \text{by definition of } T_i. \end{array}$$

- If $\mathtt{R} = \mathtt{R}_h$, in order to simplify the presentation, let us consider that the definition of $\mathtt{R}_h$ in db is: $\mathtt{sel\_hyp} \equiv$ ASSUME $\mathtt{query}'$ NOT IN $\mathtt{R}_a$ query.

$$\begin{array}{rcll} T_i(I^n)(\mathtt{db})(\mathtt{R}_h) & = & [\![\mathtt{sel\_hyp}]\!]^{I^n}_{\mathtt{db}} & \text{by the definition of } T_i, \\ & = & [\![\mathtt{query}]\!]^{I^n}_{\mathtt{db}_1} & \text{by semantics of } \mathtt{sel\_hyp}, \quad (1) \\ & = & [\![\mathtt{query}]\!]^{I^n}_{\mathtt{db}} & \text{by Lemma 9}, \quad (2) \\ & = & [\![\mathtt{query}]\!]^{I^n}_{\mathtt{db}'} & \text{by the induction hypothesis}, \\ & = & T_i(I^n)(\mathtt{db}')(\mathtt{R}_h) & \text{by definition of } T_i. \end{array}$$

In (1), $\mathtt{db}_1 \equiv \mathtt{db}[\mathtt{R}_a\ \mathtt{sch}_a := \mathtt{query}'_a\ \textsc{except}\ \mathtt{query}'/\mathtt{R}_a\ \mathtt{sch}_a := \mathtt{query}'_a]$, where $\mathtt{query}'_a$ is the definition of $\mathtt{R}_a$ in db.

In (2), Lemma 9 can be applied because, by hypothesis, $\mathtt{query}'_a$ is equivalent to $\mathtt{query}_a$ EXCEPT $\mathtt{query}'$ for certain $\mathtt{query}_a$, then $\mathtt{query}'_a$ EXCEPT $\mathtt{query}'$ is equivalent to $\mathtt{query}_a$ EXCEPT $\mathtt{query}'$ EXCEPT $\mathtt{query}'$, that is equivalent to $\mathtt{query}_a$ EXCEPT $\mathtt{query}'$, so equivalent to $\mathtt{query}'_a$. Then db and $\mathtt{db}_1$ coincide in every R, but in $\mathtt{R}_a$, and the definitions of $\mathtt{R}_a$ in db and db′ are equivalent, therefore db and db′ are equivalent.

Therefore $fix_i(\mathtt{db}) = fix_i(\mathtt{db}')$. If $j > i$, $fix_j(\mathtt{db}) = fix_j(\mathtt{db}')$ is easy to prove because the database definitions db and db′ coincide for every stratum greater than i.

Theorem 2. *Let* db *be an HR-SQL database definition in* $\mathcal{DB}$*, and let* db′ *be the result of applying the algorithm* transform *to* db*. Then* db′ *is a stratifiable R-SQL database definition, such that:*

$$fix(\mathtt{db}) = fix(\mathtt{db}')|_{\mathtt{RN}}.$$

Proof. We have proved that for every input $\mathtt{db} \in \mathcal{DB}$, the result db′ is a stratifiable R-SQL database definition. Now we prove that db and db′ are fixpoint equivalent.

Let $1 \le i \le numstr$ be a stratum and db be an HR-SQL with at least one hypothetical relation in stratum i. Let $\mathtt{db}_t$ be the output of applying $tr(i, \mathtt{db})$. We claim that $fix(\mathtt{db})(\mathtt{R}) = fix(\mathtt{db}_t)(\mathtt{R})$, for every relation R defined in db. Therefore, by transitivity, $fix(\mathtt{db})(\mathtt{R}) = fix(\mathtt{db}')(\mathtt{R})$ for every $\mathtt{R} \in \mathtt{RN}$, because the final db′, resulting of applying transform to db, is obtained by successive applications of tr.

Now we prove the claim: $fix(\mathtt{db})(\mathtt{R}) = fix(\mathtt{db}_t)(\mathtt{R})$, *for every relation* $\mathtt{R} \in \mathtt{RN}$. Let $\mathtt{R}_h\ \mathtt{sch}_h$:= ASSUME query′ [NOT] IN $\mathtt{R}_a$ query be the hypothetical relation of db changed to obtain $\mathtt{db}_t$. Then, $\mathtt{R}_h$ is defined in $\mathtt{db}_t$ as $\mathtt{R}_h\ \mathtt{sch}_h$:= queryσ, where σ represents the name replacement $[\mathtt{R'}_1/\mathtt{R}_1]\ldots[\mathtt{R'}_k/\mathtt{R}_k][\mathtt{R'}_a/\mathtt{R}_a]$, specified in the definition of $tr(i, \mathtt{db})$. Since the definition in db of any $\mathtt{R} \in \mathtt{RN}$, but $\mathtt{R}_h$, coincides with its definition in $\mathtt{db}_t$, the claim is obvious if R is a relation which does not depend on $\mathtt{R}_h$. So we focus here on the proof of $fix(\mathtt{db})(\mathtt{R}_h) = fix(\mathtt{db}_t)(\mathtt{R}_h)$.

Let $\mathtt{db}' = \mathtt{db}[\mathtt{R}_a\ \mathtt{sch}_a := \mathtt{query}_a(\textsc{union}|\textsc{except})\ \mathtt{query}'/\mathtt{R}_a\ \mathtt{sch}_a := \mathtt{query}_a]$.

Let $\mathtt{db}'' = \mathtt{db}'[\mathtt{R}_h\ \mathtt{sch}_h := \mathtt{query}\ /\ \mathtt{R}_h\ \mathtt{sch}_h := \mathtt{sel\_hyp}]$.

Notice that db′ and db″ are equal but the assumption on $\mathtt{R}_h$.

$$\begin{array}{rcll} fix(\mathtt{db})(\mathtt{R}_h) & = & [\![\mathtt{query}]\!]^{fix}_{\mathtt{db}'} & \text{by the semantics of } \mathtt{sel\_hyp}, \\ & = & [\![\mathtt{query}]\!]^{fix}_{\mathtt{db}''} & \text{by Lemma 10}, \\ & = & fix(\mathtt{db}'')(\mathtt{R}_h) & \text{by fixpoint}, \\ & = & fix(\mathtt{db}_t)(\mathtt{R}_h) & \text{by Lemma 6}. \end{array}$$

From this fact, it can be deduced that $fix(\mathtt{db})(\mathtt{R}) = fix(\mathtt{db}_t)(\mathtt{R})$ also for the relations $\mathtt{R} \in \mathtt{RN}$ that depend on $\mathtt{R}_h$. Therefore $fix(\mathtt{db}) = fix(\mathtt{db}')|_{\mathtt{RN}}$.

The complexity of the algorithm transform **for nested** ASSUME**'s.** The algorithm transform can include a great number of new auxiliar relations, this number is related to the number of assumptions in the database definition and to the number of relations that depend on the relations over which assumptions are done. But, although the database can grow up exponentially in the presence of mutual recursion between hypothetical relations, in the opposite case this measure is polynomial in the worst case, as we will see.

Let us establish a bound for the number of auxiliary relations introduced by the process of ASSUME elimination if there is not a mutual recursion between hypothetical relations. If this is the case, there is at most one hypothetical relation in every stratum s.

Let $\mathtt{R}_h\ \mathtt{sch}$:= ASSUME $\mathtt{hypo}_1, \ldots, \mathtt{hypo}_{n_s}$ query, where $\mathtt{hypo}_1, \ldots, \mathtt{hypo}_{n_s}$ do not contain nested hypothesis, the hypothetical relation of stratum s if there is any. And $\mathtt{hypo}_i \equiv \mathtt{query}_{a_i}$ [NOT] IN $\mathtt{R}_{a_i}$, $1 \le i \le n_s$.

In the elimination of $\mathtt{hypo}_i$, $1 \le i \le n_s$, the number of auxiliary relations introduced will be the cardinal of the set $S_i = \{\mathtt{R} \mid \mathtt{R}$ depends on $\mathtt{R}_{a_i}$ and ($\mathtt{R} \in \mathtt{RN}_{\mathtt{query}}$ or there is $\mathtt{R}' \in \mathtt{RN}_{\mathtt{query}}$ such that R′ depends on R)$\} \setminus \mathtt{R}_h$. Moreover, any relation in S_{i+1} is also in S_i or is a copy of a relation in S_i that is not in S_{i+1}, because the dependencies concerning the renamed relations relate to the dependencies between the corresponding original relations, and if a relation is renamed, it will not appear in the elimination of the next ASSUME. In addition, the elimination of a hypothesis may give rise to the elimination of an edge in the dependency graph. Then $|S_{i+1}| \le |S_i|$, $1 \le i < n_s$. On the other side, the cardinal of S_1 is

always less than or equal to the number m_s of relations whose stratum is less than or equal to s, after eliminating the ASSUME clauses of the previous strata. Therefore, in the worst case there will be $m_s \times n_s$ new relations in the stratum s.

The total number of new relations introduced to process a database definition db with strata st will be $\sum_{s=1}^{st} m_s \times n_s$, in the worst case. Moreover, we can distinguish two cases:

A. *There is not any dependency between hypothetical relations*. Then $\sum_{s=1}^{st} m_s \times n_s \leq m \times n$, where m is the total number of relations in db and n the total number of hypothesis in db. Because hypothetical relations do not depend on auxiliary relations of previous strata, m_s coincides with the number of relations whose stratum is less than or equal to s of the original db.

B. *Hypothetical relations could depend on hypothetical relations of previous strata*. In this case, the hypothetical relation in the stratum $s > 1$ can depend on the introduced relations in previous strata. Then, the number m_s that bounds the cardinal of the sets $S_1, \ldots, S_{n_s}$ is not the number of relations whose stratum is less than or equal to s of the original db. In fact, in order to estimate the bound m_{s+1}, $1 \leq s < st$, the original amount of relations under stratum $s+1$ should be increased with the $m_s \times n_s$ relations introduced in the process of stratum s. But also in this case the complexity is polynomial.

As mentioned before, the more general case that includes mutual recursive hypothetical relations can result in an exponential number of new auxiliary relations, since renaming a hypothetical relation increases the number of hypothesis in a stratum. Although the language is expressive enough to allow this situation, it is not usual.

5. HR-SQL with -aggregates

Aggregate functions are useful for summarizing data by computing single values from a set of numerical or other-type values. For instance, common predefined functions of relational query languages include the average and the maximum of a numeric attribute. In attempting to add standard aggregation constructs from the basic relational model to our extended language, several obstacles come up. Aggregate functions take a set of values as input and return a single value. So, a crucial question concerns with the use of recursion and the fact that hypothetical queries can locally increase or decrease the database. We have taken advantage of certain aspects of the operational semantics of our database system in order to deal with these problems, as we show in this section. First we introduce the syntax and give some examples that corroborate the gains in expressiveness of our extension. The syntax is defined as:

```
query   ::= sel_stm | sel_hyp | sel_agg | query UNION query | query EXCEPT query
sel_agg ::= SELECT exp_agg,...,exp_agg FROM R,...,R
            [GROUP BY R.A,...,R.A] [HAVING hcond]
exp_agg ::= C | R.A | exp_agg m_op exp_agg | - exp_agg | agg
agg     ::= COUNT(*) | AVG(R.A) | SUM(R.A) | MAX(R.A) | MIN(R.A)
hcond   ::= TRUE | FALSE | exp_agg c_op exp_agg | NOT hcond | hcond [AND|OR] hcond
```

The syntax of query is now extended with the new base case sel_agg. So, aggregates can occur in queries as well as in relation definitions. The notation used before is extended to incorporate this new case in a natural way. For instance, $\mathrm{RN}_{\texttt{sel\_agg}}$ denotes the set of relation names occurring in the FROM and HAVING clauses of sel_agg.

Recursive definitions in current RDBMS's can include restricted applications of aggregates (as introduced in Section 2, GROUP BY and HAVING are not allowed). They can be used to control termination in some problems, as we will show in Example 15.

In the setting of hypothetical reasoning (which is not supported at all by any current RDBMS), source relations over which aggregates apply may be modified by assumptions (either by augmenting with new tuples or reducing by discarding existing tuples). This enables decision support and business intelligence applications. Business intelligence [45] uses data integration, data warehousing, analytic processing and other techniques. In particular, one of these techniques refer to "what-if" applications. Hypothetical queries are useful, for instance, in decision support systems as they allow to submit a query by assuming that the knowledge in the database is changed. Such knowledge can be changed by adding and deleting (locally and temporarily) both tuples and production rules (queries). Tuples can be simply stated in from-less select statements and production rules as general select statements. With the extension of aggregates in HR-SQL, aggregate functions can be included in any select statement, both in assumed (intensional) data and in regular SQL statements, as we exemplify next.

Example 15. As an example of using aggregates to control termination, and recalling the database of the Canary Islands, the sum of all the times of the relation link can be used as a bound to finding out the time needed to travel from a location to another (cf. the relation travel in Example 3). Obviously, this number must be less than or equal to that sum. So, if max_travel_time is defined as:

```
max_travel_time(total float) := SELECT SUM(time) FROM link;
```

then, another version for the relation travel can be written as follows:

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

```
limited_travel(ori varchar(10), des varchar(10), time float) :=
  SELECT link.ori, link.des, link.time FROM link UNION
  SELECT link.ori, limited_travel.des, link.time + limited_travel.time
    FROM link, limited_travel, max_travel_time
    WHERE link.des = limited_travel.ori AND
      link.time + limited_travel.time <= max_travel_time.total;
```

An equivalent formulation (modulo duplicates) in standard SQL is:

```
CREATE VIEW limited_travel(ori, des, time) AS
  WITH RECURSIVE cte(ori, des, time) AS
    (SELECT link.ori, link.des, link.time FROM link UNION ALL
     SELECT link.ori, cte.des, link.time + cte.time
       FROM cte, link WHERE link.des = cte.ori)
  SELECT cte.* FROM cte, max_travel_time
    WHERE cte.time < max_travel_time.total;
```

Even with this upper bound, as we mention in Section 2.2, current RDBMS's can lead to non-termination when cycles are present in the graph. But in HR-SQL the use of this aggregate allows to set a upper bound to ensure termination. Note also that SQL systems do not allow duplicate elimination (they require UNION ALL and disallow UNION) in recursive queries as HR-SQL does.

The relation `limited_travel` of HR-SQL can be used even if we consider there is a new flight connection from El Hierro to La Palma airports, which implies a cycle in the graph:

```
ASSUME SELECT 'RES','SPC',1 IN flight SELECT * FROM limited_travel;
```

As before, since `limited_travel` is limited by the aggregate SUM, the result set is finite and the computation terminates.

Example 16. Another simple example that shows how aggregates can explicitly occur in a regular SQL statement under an assumption is the following query, which asks for the average time for `travel` in the Canary Islands (Example 3) if the shorter direct links are eliminated from `travel`:

```
ASSUME SELECT ori, des, MIN(time) FROM link GROUP BY ori, des NOT IN travel
  SELECT AVG(time) FROM travel;
```

Example 17. Let us consider now the hypothetical scenario considered in Example 4: for each boat connection to Valverde that takes more than one hour an extra connection to Valverde taking half an hour less is added, but the original boat to Valverde from La Gomera is suspended. For knowing the minimum time to travel between two locations in those assumptions, the following relation can be defined using the hypothetical relation `hyp_travel2`:

```
min_travel(ori varchar(10), des varchar(10), time float) :=
  SELECT ori, des, MIN(time) FROM hyp_travel2 GROUP BY ori, des;
```

And the following example represents the increased minimum times due to the hypothetical changes in the connections with respect to the normal scenario:

```
inc_min_travel(ori varchar(10), des varchar(10), time float) :=
  SELECT travel.ori, travel.des, MIN(hyp_travel2.time) - MIN(travel.time)
    FROM travel, hyp_travel2
    WHERE travel.ori = hyp_travel2.ori AND travel.des = hyp_travel2.des
    GROUP BY travel.ori, travel.des
    HAVING MIN(hyp_travel2.time)-MIN(travel.time)>0;
```

We finish this section with an example that combines some complex features of the language.

Example 18. The syntax of our language allows to using mutual recursion between hypothetical relations as well as aggregates. This example is presented just to show this possibility.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

```
R1(x float) := ASSUME SELECT 'TFS', 'LPA', 1.5 IN boat
  SELECT * FROM R2 UNION SELECT AVG(time) FROM link;

R2(x float) := ASSUME SELECT 'MP', 'RES', 1.0 NOT IN flight
  SELECT * FROM R1 UNION SELECT MIN(time) FROM link;
```

The result of this example is: $\texttt{R1} = \{(1.0), (1.75)\}$, $\texttt{R2} = \{(1.0), (1.75)\}$.

5.1. Semantics of aggregates

The stratified fixpoint computation, designed to support negation, is a good framework to incorporate aggregates. The rationale behind this is ensuring monotonicity as, along building the fixpoint for a given stratum, the result of an aggregate function taking values in this stratum may change. For instance, the count of tuples corresponding to a relation is only known after such a relation has been completely computed. This is guaranteed if the sum is computed in a stratum above the relation stratum. This can be achieved by introducing a negative dependency in the dependency graph. Therefore, we extend the notion of dependency graph to incorporate the dependencies generated by the use of aggregates, as follows:

If the definition of a relation R contains a sel_agg (as defined in the grammar for aggregates) every $\texttt{R}' \in \texttt{RN}_{\texttt{sel\_agg}}$ produces a negatively labeled edge from R′ to R. As before, $str(\texttt{sel\_agg}) = max\{s(\texttt{R}) | \texttt{R} \in \texttt{RN}_{\texttt{sel\_agg}}\}$.

We introduce some notation in order to define the interpretation of sel_agg.

Let S be a multiset of tuples of the same arity and such that each component of the tuples corresponds to a relation attribute. Let $\texttt{A}_1, \ldots, \texttt{A}_n$ be n of such attributes, $n \geq 1$. If $\overline{a} \in S$, $\overline{a}_{\texttt{A}_1 \ldots \texttt{A}_n}$ represents the components of the tuple $\overline{a}$ corresponding to the attributes $\texttt{A}_1, \ldots, \texttt{A}_n$. By $S|_{\texttt{A}}$ we denote the projection $\{\{\overline{a}_{\texttt{A}} \mid \overline{a} \in S\}\}$.

Let agg be an aggregate function defined over some of the attributes associated to S. The value $agg{\downarrow}_S$ is the result of evaluating the aggregate function involved in agg (which is a mathematical function) over the corresponding projection of S. For instance, for $\texttt{agg} \equiv \textsc{avg}(\texttt{A})$, $agg{\downarrow}_S = avg(S|_{\texttt{A}})$, which means the average value of the multiset $S|_{\texttt{A}}$. For $\texttt{agg} \equiv \textsc{count}(*)$, $agg{\downarrow}_S = |S|$, the cardinal of S. If $\overline{\texttt{agg}}$ is a sequence of aggregate functions $\overline{agg{\downarrow}_S}$ represents the sequence of the corresponding evaluations for S.

Definition 9. Let $i \geq 1$, and $\mathcal{I}_i^{\mathcal{DB}}$. Let sel_agg be a select statement including aggregates. The interpretation of sel_agg with respect to I for a database $\texttt{db} \in \mathcal{DB}$, denoted by $[\![\texttt{sel\_agg}]\!]^I_{\texttt{db}}$, is recursively defined as follows. For the sake of clarity, we define $[\![\texttt{sel\_agg}]\!]^I_{\texttt{db}}$ distinguishing if the optional parts are included in sel_agg or not, and considering the GROUP BY clause grouping by only one attribute:

- $[\![\textsc{select}\ \texttt{exp\_agg}_1, \ldots, \texttt{exp\_agg}_k\ \textsc{from}\ \texttt{R}\ ]\!]^I_{\texttt{db}} =$
  $\{(ev((\texttt{exp\_agg}_1, \ldots, \texttt{exp\_agg}_k)[\overline{agg{\downarrow}_{I(\texttt{db})(\texttt{R})}}/\overline{\texttt{agg}}])\}$, where $\overline{\texttt{agg}}$ represents the sequence of the different aggregate functions that occur in sel_agg.
- $[\![\textsc{select}\ \texttt{exp\_agg}_1, \ldots, \texttt{exp\_agg}_k\ \textsc{from}\ \texttt{R}\ \textsc{having}\ \texttt{hcond}]\!]^I_{\texttt{db}} =$
  $\{(ev((\texttt{exp\_agg}_1, \ldots, \texttt{exp\_agg}_k)[\overline{agg{\downarrow}_{I(\texttt{db})(\texttt{R})}}/\overline{\texttt{agg}}]) \mid ev(\texttt{hcond}[\overline{agg{\downarrow}_{I(\texttt{db})(\texttt{R})}}/\overline{\texttt{agg}}])$ is satisfied$\}$, where $\overline{\texttt{agg}}$ is defined as before.
- $[\![\textsc{select}\ \texttt{exp\_agg}_1, \ldots, \texttt{exp\_agg}_k\ \textsc{from}\ \texttt{R}\ \textsc{group by}\ \texttt{A}\ ]\!]^I_{\texttt{db}} =$
  $\{(ev((\texttt{exp\_ag}_1, \ldots, \texttt{exp\_ag}_k)[\overline{agg{\downarrow}_{S_d}}/\overline{\texttt{agg}}][d/\texttt{A}]) \mid d \in D\}$, where $\overline{\texttt{agg}}$ is defined as before, D is the domain denoted by the type of the attribute A, and for each $d \in D$, $S_d = \{\overline{a} \in I(\texttt{db})(\texttt{R}) \mid \overline{a}_{\texttt{A}} = d\}$, that means the tuples of $I(\texttt{db})(\texttt{R})$ such that the value of the component corresponding to the attribute A is d. Notice that since A can occur in some $\texttt{exp\_agg}_j$, it is replaced by the value d, previously to the evaluation.
- $[\![\textsc{select}\ \texttt{exp\_agg}_1, \ldots, \texttt{exp\_agg}_k\ \textsc{from}\ \texttt{R}\ \textsc{group by}\ \texttt{A}\ \textsc{having}\ \texttt{hcond}\ ]\!]^I_{\texttt{db}} =$
  $\{(ev((\texttt{exp\_agg}_1, \ldots, \texttt{exp\_agg}_k)[\overline{agg{\downarrow}_{S_d}}/\overline{\texttt{agg}}][d/\texttt{A}]) \mid S_d \in \mathcal{S}\}$,
  where $\mathcal{S} = \{S_d \mid d \in D$ and $ev(\texttt{hcond}[\overline{agg{\downarrow}_{S_d}}/\overline{\texttt{agg}}][d/\texttt{A}])$ is satisfied$\}$.

For the general case of grouping by n attributes, the previous definition can be generalized as follows:

$[\![\textsc{select}\ \texttt{exp\_agg}_1, \ldots, \texttt{exp\_agg}_k\ \textsc{from}\ \texttt{R}\ \textsc{group by}\ \texttt{A}_1, \ldots, \texttt{A}_n]\!]^I_{\texttt{db}} =$

$\{(ev((\texttt{exp\_ag}_1, \ldots, \texttt{exp\_ag}_k)[\overline{agg{\downarrow}_{S_{\overline{d}}}}/\overline{\texttt{agg}}][\overline{d}/\overline{\texttt{A}}]) \mid \overline{d} \in D_1 \times \ldots \times D_n\}$

where D_i is the domain denoted by the type of the attribute $\texttt{A}_i$, $1 \leq i \leq n$, and for each $\overline{d} \in D_1 \times \ldots \times D_n$, $S_{\overline{d}} = \{\overline{a} \in I(\texttt{db})(\texttt{R}) \mid \overline{a}_{\texttt{A}_1 \ldots \texttt{A}_n} = \overline{d}\}$, that means the tuples of $I(\texttt{db})(\texttt{R})$ such that, for every i, $1 \leq i \leq n$, the value of the component corresponding to the attribute $\texttt{A}_i$ is d_i.

The generalization for the HAVING clause, grouping by more than one attribute is analogous, but considering the set collection: $\mathcal{S} = \{S_{\overline{d}} \mid \overline{d} \in D_1 \times \ldots \times D_n$ and $ev(\texttt{hcond}[\overline{agg{\downarrow}_{S_{\overline{d}}}}/\overline{\texttt{agg}}][\overline{d}/\overline{\texttt{A}}])$ is satisfied$\}$.

A direct extension of Lemma 2, considering the case sel_agg, can be directly obtained from the fact that if R sch := sel_agg is the definition of R and $str(\texttt{R}) = i$, then the stratum of each relation R' in sel_agg is less than i. So if

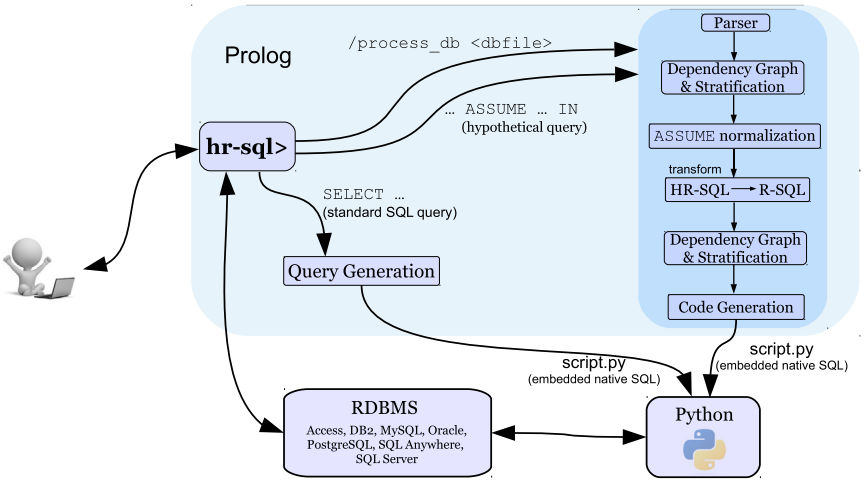


Fig. 7. HR-SQL system.

$I_1, I_2 \in \mathcal{I}_i^{\mathcal{DB}}$ and $I_1 \sqsubseteq I_2$, then $I_1(\mathtt{db})(R') = I_2(\mathtt{db})(R')$. Hence, $agg{\downarrow}_{I_1(\mathtt{db})(\mathtt{R}')} = agg{\downarrow}_{I_2(\mathtt{db})(\mathtt{R}')}$, from which it is easy to prove that $[\![\mathtt{sel\_agg}]\!]^{I_1}_{\mathtt{db}} = [\![\mathtt{sel\_agg}]\!]^{I_2}_{\mathtt{db}}$. As a consequence, the continuity of the fixpoint operator is preserved.

Example 19. Consider the following query:

`SELECT A, (MIN(B) + MAX(B)) / 2.0 FROM R GROUP BY A HAVING COUNT(*) > 1`

that makes reference to a relation `R (A integer, B integer)`.

Since the database is not relevant here, apart from `R`, we omit it in the notation. Let us assume a concrete interpretation I such that $I(\mathtt{R}) = \{(0,0), (1,0), (1,2), (2,0), (2,1)\}$.

In order to get the interpretation of this query with respect to I, we identify:

$S_d = \{(a,b) \in I(\mathtt{R}) \mid a = d\}$, since a only takes the values 0, 1 and 2, in $I(\mathtt{R})$, we only have:

$$S_0 = \{(0,0)\}, S_1 = \{(1,0), (1,2)\}, S_2 = \{(2,0), (2,1)\}$$

And $\overline{\mathtt{agg}} \equiv \mathtt{MIN(B)}, \mathtt{MAX(B)}, \mathtt{COUNT(*)}$. For instance, for $d = 1$, $\overline{agg{\downarrow}_{S_1}} = 0, 2, 2$, because:

$$min(S_1|_B) = min(\{0,2\}) = 0, max(S_1|_B) = max(\{0,2\}) = 2, |S_1|_B| = |\{0,2\}| = 2$$

Analogously, $\overline{agg{\downarrow}_{S_0}} = 0, 0, 1$, and $\overline{agg{\downarrow}_{S_2}} = 0, 1, 2$. In order to evaluate $\mathtt{hcond} \equiv \mathtt{COUNT(*)} > 1$, only the component of $\overline{\mathtt{agg}}$, corresponding to the aggregate $\mathtt{COUNT(*)}$, will be taken into account. So, we have:

- for S_1: $ev((\mathtt{COUNT(*)} > 1)[2/\mathtt{COUNT(*)}]) \equiv 2 > 1$ that is true;
- analogously, for S_2: $ev((\mathtt{COUNT(*)} > 1)[2/\mathtt{COUNT(*)}]) \equiv 2 > 1$ is true,
- but for S_0: $ev((\mathtt{COUNT(*)} > 1)[1/\mathtt{COUNT(*)}]) \equiv 1 > 1$ is false.
- Then: $\mathcal{S} = \{S_d \mid d \in \{0,1,2\}$ and $ev(\mathtt{hcond}[\overline{agg{\downarrow}_{S_d}}/\overline{\mathtt{agg}}][d/A])$ is satisfied $\} = \{S_1, S_2\}$

Now $[\![\mathtt{SELECT\ A,\ (MIN(B)+MAX(B))/2.0\ FROM\ R\ GROUP\ BY\ A\ HAVING\ COUNT(*)>1}]\!]^I$ =
$\{ev((\mathtt{A}, (\mathtt{MIN(B)} + \mathtt{MAX(B)})/2.0)[0,2,2/\mathtt{MIN(B)}, \mathtt{MAX(B)}, \mathtt{COUNT(*)}][1/\mathtt{A}])$,
$ev((\mathtt{A}, (\mathtt{MIN(B)} + \mathtt{MAX(B)})/2.0)[0,1,2/\mathtt{MIN(B)}, \mathtt{MAX(B)}, \mathtt{COUNT(*)}][2/\mathtt{A}])\} =$
$\{(1, (0+2)/2.0), (2, (0+1)/2.0)\} = \{(1, 1.0), (2, 0.5)\}$.

6. The HR-SQL system

In this section we present a concrete implementation for HR-SQL. The system is programmed in Prolog (there are available versions for SWI-Prolog and SICStus Prolog) and generates Python code for computing the database relations using a state-of-the-art RDBMS. Both Prolog and Python are used as an easy way for developing a prototype, but the system HR-SQL can be implemented in any other language. Currently the system can operate with a variety of RDBMS's: Access, DB2, MySQL, Oracle, PostgreSQL, SQL Anywhere and SQL Server. Python plays the role of a scripting language that provides an easy access to those RDBMS's and also control structures (in particular, loops) for implementing our fixpoint algorithm.

The system HR-SQL is available at gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/HR-SQL2 together with a bundle of examples including those introduced in previous sections. This URL contains also some system session examples and a brief description of the main commands of the system.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

This system is an improvement over the former version [5] and it now deals with assumptions not only in views but also in queries. The system features ODBC connections instead of a single specific connector for PostgreSQL. The system performs now a transformation of the HR-SQL database into an equivalent R-SQL database, independent of the concrete RDBMS, allowing a modular treatment of complex databases. Finally, the system has been tweaked to fit the Java-based integrated development environment ACIDE [37] requirement, and thus a graphical user interface is now available (including syntax highlighting, project management, multiple editor windows, relocatable panels, and graphical display of the dependency graph).

The structure of the system is depicted in Fig. 7. It provides a simple interface to interact with the user from the prompt `HR-SQL>`. The command `/process_db <dbfile>` loads and processes the file `<dbfile>` containing an HR-SQL database definition, i.e., it parses the file, builds the corresponding dependency graph and a stratification (if possible; otherwise, an error message is raised). Then it converts every `sel_hyp` into its normal form (ASSUME normalization), providing a suitable HR-SQL database that is transformed into an R-SQL database which does not contain any ASSUME according to the `transform` algorithm (Fig. 5). At this point, the current R-SQL database contain auxiliary relations due to the previous flattening and elimination of ASSUME's, then the system must build a new dependency graph and stratification containing the new relations. This stratification is guaranteed to exist, since the new database is a conservative extension of the original one (the transformations can not introduce cycles with negative dependencies). Finally, the system produces a Python script in order to generate the tables corresponding to the database relations according to the fixpoint semantics (see Section 6.2). Then the system calls Python to run this script, which connects to the external RDBMS and which executes the SQL instructions and materializes the relations in tables.

When submitting a query, the system first determines if it is either a standard SQL query or a hypothetical query (contains some ASSUME clause). For the first case it directly submits such a query to the relational database via ODBC. For the second case, it is necessary to compute the fixpoint of the database fragment demanded by the query, and the complete process of compilation must be performed for such a query. The results are retrieved from the RDBMS via ODBC.

In the next sections we detail the implementation of the different components of the system. Section 6.1 introduces an elaborated stratification algorithm that will allow to improve the efficiency of the fixpoint algorithm. Section 6.2 shows the algorithm to produce the Python script for computing the fixpoint. The processing of queries is explained in Section 6.3.

6.1. Stratification

According to Definition 2, given a database and its corresponding dependency graph, in general, there can be a number of different stratifications for it. In order to improve the efficiency of the fixpoint computation, the stratification algorithm chooses a stratification that minimizes the number of relations in each strata. The underlying motivation is that every iteration of the fixpoint operator at stratum i requires to process each relation `R` of stratum i, even if either `R` has been saturated or there is no relation on stratum i that depends on `R`. This algorithm was presented in [5] and in Appendix A we show it in detail.

Example 6 shows two possible stratifications for the database of Example 4. According to the previous ideas, the stratification *str* obtained by this algorithm is the second one, i.e., $str(\texttt{boat}) = 1$, $str(\texttt{bus}) = 2$, $str(\texttt{flight}) = 3$, $str(\texttt{link}) = 4$, $str(\texttt{reachable}) = 5$, $str(\texttt{avoidMad}) = 6$, $str(\texttt{hyp\_travel1}) = 7$, $str(\texttt{travel}) = 8$, $str(\texttt{hyp\_travel2}) = 9$ $str(\texttt{val\_to\_mad}) = 10$.

6.2. Python script for fixpoint computation

In this section we introduce the algorithm for generating the physical database of an HR-SQL database definition by computing its fixpoint. As we pointed out, we consider the database defined as the set of its extensional relations (SQL tables) and intensional relations (SQL views), without an explicit difference between both of them. Following this approach, when its fixpoint is calculated in order to compute the concrete instance, all the relations are materialized. Here we explain the process of *Code Generation* of Fig. 7, so we assume: First, a stratification for the database has been obtained by applying the algorithm of the previous section. And, second, the algorithm `transform` of Section 4 has already been applied, obtaining an R-SQL database definition `db`.

Fig. 8 (a) outlines the algorithm for computing the fixpoint of `db` by means of SQL-statements (CREATE and INSERT) whose execution will build the corresponding physical database. In the following, *numStr* denotes the number of strata in that stratification, $\texttt{RN}$ is the set of relation names in `db`, $\texttt{RN}_i$ is the set of relation names of `db` at stratum i, and $\texttt{RN}' \subseteq \texttt{RN}_{\texttt{db}}$ is the set of auxiliary relations introduced by the calls to the transformation function *tr* of Fig. 6 needed to transform the original database into the corresponding R-SQL database. According to this, the set $\texttt{RN} \setminus \texttt{RN}'$ contains the relation names of the original HR-SQL database.

First of all, a table is created for each relation $\texttt{R sch} := \texttt{sel\_stm}_{\texttt{R}}$ of the original HR-SQL database (lines 1-3), while for auxiliary relations, tables are declared as temporary (lines 4-6), since they are not relevant once the fixpoint computation ends. Temporary tables are useful devices with respect to performance because, first, they avoid locks for handling concurrent access as they are private for the session. Second, they avoid writing to the database log as a rollback simply deletes the table. Finally, as they are stored primarily in main memory, the storage manager does not intervene if not needed by

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

```
1  for all R ∈ RN \ RN' do
2     CREATE TABLE R sch;
3  end for
4  for all R ∈ RN' do
5     CREATE TEMPORARY TABLE R sch;
6  end for
7  i = 1
8  while i ≤ numStr do
9     for all R ∈ RN_i do
10       INSERT INTO R out(sel_stm_R);
11    end for
12    if recursive_str(i) then
13       newTuples = true;
14       while newTuples do
15          size = rel_size(RN_i)
16          for all R ∈ RN_i do
17             INSERT INTO R (in(sel_stm_R)
                  EXCEPT SELECT * FROM R);
18          end for
19          newTuples = (size != rel_size(RN_i));
20       end while
21    end if
22    i = i + 1
23 end while
```

(a) Basic Algorithm

```
1  if recursive_str(i) then
2     for all R ∈ RN_i do
3        CREATE TEMPORARY TABLE D1_R sch_R;
4        CREATE TEMPORARY TABLE D2_R sch_R;
5        INSERT INTO D1_R (SELECT * FROM R);
6     end for
7     newTuples = true;
8     while newTuples do
9        newTuples = false;
10       for all R ∈ RN_i do
11          INSERT INTO D2_R trDif(R)
12                EXCEPT (SELECT * FROM R);
13          if (size(D2_R) > 0) then
14             newTuples = true;
15             INSERT INTO R (SELECT * FROM D2_R);
16             DELETE FROM D1_R;
17             INSERT INTO D1_R (SELECT * FROM D2_R);
18             DELETE FROM D2_R;
19          end if
20       end for
21    end while
22 end if
```

(b) Differential Optimization

Fig. 8. Algorithm to compute the fixpoint.

memory exhaustion. Therefore, they can provide memory scalability and performance. Temporary tables are erased once the fixpoint computation has ended.

The external `while` at line 8 successively computes the fixpoints $fix_1(\mathtt{db})$, $fix_2(\mathtt{db})$, ..., $fix_{numStr}(\mathtt{db})$. Following the theoretical presentation, each $fix_i(\mathtt{db})$ is calculated for every relation of $\mathtt{RN}_i$, by iterating the operators T_i of Definition 5, i.e., the internal `while` (lines 14-20) at iteration n ($n > 1$) computes $T_i^n(fix_{i-1})(\mathtt{db})$ and it is only necessary for *recursive strata* (those that contain recursive relations). The function *recursive_str*(i) checks if i is a recursive stratum. The loop is iterated whenever a tuple is added to a table of the current stratum; the expression *rel_size*($\mathtt{RN}_i$) denotes the total number of tuples of the relations at stratum i and the variable *size* (updated at line 15 and used at line 19) is used to check if some relation of the current stratum has changed. In a practical implementation, in order to prevent infinite loops for infinite relations, a bound on the number of iterations can be fixed. The number of iterations is initialized to 0, before the `while` at line 14, and increased for each loop, which finishes if that bound is reached.

This algorithm enhances the one introduced in [4] by reducing the work in each iteration of the internal `while` loop, by simplifying the operations done for filling the tables, so improving the efficiency. The idea is that for nonrecursive relations the operator T_i is iterated only once, and successive iterations are applied only to recursive relations; moreover, only the recursive fragment of the select statements defining them is needed. With this aim we have defined the functions *in* and *out* to split each `sel_stm` into the (recursive) fragment that must be used in the INSERT statements inside the loop, and the fragment that can be processed before (and out of) the loop, as the base case of the recursive definition. The `for` loop at lines 9-11 processes the *out* fragment for each relation, while the INSERT statement at line 17 processes the *in* fragment, that is necessary only for recursive relations. The *in* and *out* fragments of a `sel_stm` can be easily determined using the stratum of its components. Recall from the stratification defined in Section 6.1: if a relation $\mathtt{R}_1$ in stratum i depends on another relation $\mathtt{R}_2$, then the stratum of $\mathtt{R}_2$ is lower than i, so it must be previously computed, or it is exactly i (if they are mutually recursive) and both relations must be computed simultaneously. Therefore, if for instance `R := sel_stm`$_1$ UNION `sel_stm`$_2$, $str(\mathtt{R}) = i$, and $str(\mathtt{sel\_stm}_1) < i$, then `sel_stm`$_1$ will be part of the *out* fragment, and the corresponding tuples can be inserted before the loop, because the involved relations are already calculated in the computation of a previous stratum. Functions *in* and *out* for a stratum i are defined by recursion on the structure of `query` as follows:

- If $str(\mathtt{query}) < i$, then we have: $in(\mathtt{query}) = \emptyset$ and $out(\mathtt{query}) = \mathtt{query}$
- If $str(\mathtt{query}) = i$, the functions are defined by recursion on the structure of `query`:
  - `query` ≡ SELECT exp ... exp FROM R ... R WHERE wcond
    $in(\mathtt{query}) = \mathtt{query}$ and $out(\mathtt{query}) = \emptyset$
  - `query` ≡ `query`$_1$ UNION `query`$_2$
    $in(\mathtt{query}) = in(\mathtt{query}_1)$ UNION $in(\mathtt{query}_2)$ and
    $out(\mathtt{query}) = out(\mathtt{query}_1)$ UNION $out(\mathtt{query}_2)$
  - `query` ≡ `query`$_1$ EXCEPT `query`$_2$
    $in(\mathtt{query}) = in(\mathtt{query}_1)$ EXCEPT `query`$_2$ and
    $out(\mathtt{query}) = out(\mathtt{query}_1)$ EXCEPT `query`$_2$

For the relation `reachable(ori varchar(10), des varchar(10))` of Example 4 defined with the `query`:

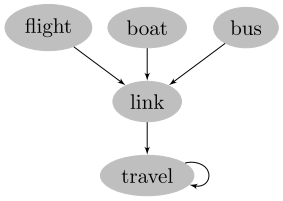


Fig. 9. DG_{db} for the Relations of Example 20.

```
1  CREATE TABLE boat (ori varchar(10), des varchar(10), time float);
   ... (analogous for the rest of relations)
2  # Code generated for Stratum 1
3  INSERT INTO boat (SELECT 'SPC','TFN',2) UNION
                     (SELECT 'TFS','GMZ',1) UNION
                     (SELECT 'GMZ','VDE',1.5);
   ... (analogous for strata 2 and 3, for bus and flight)
4  # Code generated for Stratum 4
5  INSERT INTO link (SELECT * FROM flight) UNION
                    (SELECT * FROM boat)
                    UNION (SELECT * FROM bus);
6  # Code generated for Stratum 5
7  INSERT INTO travel (SELECT * FROM link);
8  newTuples = true
9  while newTuples do
10   INSERT INTO travel
       (((SELECT link.ori,travel.des,link.time+travel.time
       FROM link,travel WHERE link.des = travel.ori))
       EXCEPT (SELECT * FROM travel));
11   newTuples = (cursor.rowcount > 0)
12 end while
```

Fig. 10. Sketch of a concrete script for computing the fixpoint for Example 20.

```
SELECT link.ori, link.des FROM link UNION
SELECT link.ori, reachable.des FROM link,reachable
  WHERE link.des = reachable.ori;
```

we have:

- *out*(sel_stm) = SELECT link.ori, link.des FROM link;
- *in*(sel_stm) = SELECT link.ori, reachable.des FROM link, reachable WHERE link.des=reachable.ori;

The next example illustrates how the algorithm of Fig. 8 (a) is implemented, sketching the script for computing the fixpoint in a concrete case. We use smallcaps for the SQL statements that must be submitted to the database system using the appropriate Python functions (for the sake of easing the presentation, we omit some details such as these functions). We must also note that the `while` loop at line 8 in Fig. 8 (a) is expanded and made explicit for each stratum, so that it does not appear in the script. The same is done for the `for` loop at lines 9-11. The code below follows standard SQL, but the implementation generates statements for each concrete RDBMS. For instance, Oracle uses MINUS instead of EXCEPT. Also, MySQL does not enjoy this operator and we use an alternative formulation based on the NOT IN operator.

Example 20. In this example we consider a database containing only (for simplicity) the relations `flight`, `boat`, `bus`, `link`, and `travel` as defined in Example 3. Fig. 9 shows the dependency graph for this database. According to this graph, the stratification algorithm obtains $str(\texttt{boat}) = 1$, $str(\texttt{bus}) = 2$, $str(\texttt{flight}) = 3$, $str(\texttt{link}) = 4$, $str(\texttt{travel}) = 5$.

The script for this database has the form illustrated in Fig. 10. First of all, tables are created for each relation. Then, in stratum 1, the relation `boat` has only an *out* part (line 3) as it is extensionally defined, and analogously for relations `bus` and `flight` at strata 2 and 3 respectively. The relation `link` at stratum 4 again has only an *out* part (line 5) and therefore it does not need the loop for computing the *in* part. For stratum 5, the relation `travel` produces an *out* part (line 7) and a `while` loop for the *in* part (lines 9-12). This loop inserts new tuples into the table `travel` (line 10) until reaching a

```
1  CREATE TABLE even(x integer);
2  CREATE TABLE odd(x integer);
3  # Code generated for Stratum 1
4  INSERT INTO even (SELECT 0);
5  CREATE TEMPORARY TABLE even_delta1(x integer);
6  CREATE TEMPORARY TABLE even_delta2(x integer);
7  CREATE TEMPORARY TABLE odd_delta1(x integer);
8  CREATE TEMPORARY TABLE odd_delta2(x integer);
9  INSERT INTO even_delta1 (SELECT * FROM even);
10 INSERT INTO odd_delta1 (SELECT * FROM odd);
11 newTuples = true
12 while newTuples do
13   newTuples = false
14   INSERT INTO even_delta2 ((SELECT odd_delta1.x+1 FROM odd_delta1)
       EXCEPT (SELECT * FROM even));
15   if cursor.rowcount > 0 then
16     newTuples = true
17     INSERT INTO even (SELECT * FROM even_delta2);
18     DELETE FROM even_delta1;
19     INSERT INTO even_delta1 (SELECT * FROM even_delta2);
20     DELETE FROM even_delta2;
21   end if
22   INSERT INTO odd_delta2 ((SELECT even_delta1.x+1 FROM even_delta1
       WHERE even_delta1.x < 100) EXCEPT (SELECT * FROM odd));
23   if cursor.rowcount > 0 then
24     newTuples = true
25     INSERT INTO odd (SELECT * FROM odd_delta2);
26     DELETE FROM odd_delta1;
27     INSERT INTO odd_delta1 (SELECT * FROM odd_delta2);
28     DELETE FROM odd_delta2;
29   end if
30 end while
```

Fig. 11. Sketch of a script for computing the fixpoint with differential optimization for Example 1.

fixpoint, i.e., there are not new tuples to add. The Python accessor `rowcount`, that returns the tuples computed in the previous INSERT statement, is used to check if such fixpoint is reached. Fig. 4 shows exactly this computation: before the while loop, `travel` contains the tuples corresponding to $T_5^1(fix_4)$ (the *out* part of the relation); the first iteration of the loop adds the tuples to $T_5^1(fix_4)$, corresponding to $T_5^2(fix_4)$; and so on until $T_5^5(fix_4)$.

The full Python scripts generated for the complete database of Canary Islands, as well as the description of the previous compilation stages, can be found at http://gpd.sip.ucm.es/trac/gpd/raw-attachment/wiki/GpdSystems/HR-SQL2/AppendixC.pdf.

Now we propose an optimization of the fixpoint computation, based on the semi-naïve differential optimization [44,8, 46], that is sound and complete for linear recursion. Its aim is to avoid generating the same tuples when computing the fixpoint for recursive relations. This optimization is carried out by changing the processing for recursive strata in the fixpoint algorithm. The `if` beginning at line 12 and ending at line 21 of the fixpoint algorithm in Fig. 8 (a) is replaced by the one shown in Fig. 8 (b). If the stratum i is recursive, in order to have a finer control on the tuples that have been inserted at each iteration, we use two auxiliary relations $\texttt{D1}_\texttt{R}$ and $\texttt{D2}_\texttt{R}$ for each relation `R` in it. At the beginning of each iteration of the `while` loop at line 8, we have that, for each `R` in the stratum, $\texttt{D1}_\texttt{R}$ contains the tuples added to the corresponding relation `R` in the previous iteration and $\texttt{D2}_\texttt{R}$ is empty. For each iteration, the `for` loop at line 10 inserts new tuples into the relations if possible. Such tuples are obtained considering only the tuples inserted at the previous iteration, which are the only ones that can add new information. For each `R`, if $\texttt{RN}_\texttt{i} = \{\texttt{R}_1, \ldots, \texttt{R}_\texttt{n}\}$, $trDif(\texttt{R})$ is defined as

$$(in(\texttt{sel\_stm}_\texttt{R}))[\texttt{D1}_{\texttt{R}_1}, \ldots, \texttt{D1}_{\texttt{R}_\texttt{n}}/\texttt{R}_1, \ldots, \texttt{R}_\texttt{n}]$$

so it corresponds to those hypothetical new tuples. They are added to $\texttt{D2}_\texttt{R}$ in line 11. If some tuple is added, the flag `newTuples` reflects that fact (line 13) for performing a new iteration. The insertions and deletions on lines 15-18 insert the new tuples into `R` and prepare the corresponding $\texttt{D1}_\texttt{R}$ and $\texttt{D2}_\texttt{R}$ for a new iteration. If no new tuples are added ($size(\texttt{D2}_\texttt{R}) = 0$), `newTuples` ends with the value false assigned at line 9 and the `while` loop stops.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

Example 21. In Fig. 11, we show the code corresponding to the fixpoint algorithm of Fig. 8 (b), for the database of Example 1 defining the relations `even` and `odd`.

These relations are mutually recursive and produce a unique stratum. Line 4 is due to the *out* part of them (only `even` has *out* part in this case). Lines 5-10 correspond to lines 2-6 in Fig. 8 (b). The `for` beginning at line 10 in Fig. 8 (b) is unrolled into the lines 14-29 here. The Python accessor `rowcount` is used (lines 15 and 23) to check if some tuple has been added, that corresponds to line 13 in Fig. 8 (b). The relations `even` and `odd` are built as follows:

- Before the `while` loop beginning at line 12 we have `even` = {0}, `even_delta1` = {0}, `odd` = {}, `odd_delta1` = {}.
- The first iteration of the `while` obtains `even_delta2` = {} at line 14, so the `if` true branch at lines 15-21 is not executed, and `even` as well as `even_delta1` do not change. However, for the relation `odd`, the INSERT at line 22 makes `odd_delta2` = {1}, and the execution of the `if` true branch at lines 23-29 makes `odd_delta1` = `odd` = {1}.
- The next iteration again considers only the previous relation `odd_delta1` to build `even_delta2` = {2} at line 14, then `even` = {(0), (2)} by the INSERT at line 17, and `even_delta1` = {2} by the INSERT at line 19. Analogously, regarding `odd`, `odd_delta2` = {3} is obtained using the previous `even_delta1`, therefore `odd_delta1` = {3}, and `odd` = {(1), (3)} is obtained at line 25.
- Working as before, in the next iteration we get `even` = {(0), (2), (4)}, `odd` = {(1), (3), (5)}.
- The loop finishes when both INSERT at lines 14 and 22 have no effect, giving `even` = {(0), (2), . . . , (100)} and `odd` = {(1), (3), . . . , (101)}. The Python accessor `rowcount` is used (lines 15 and 23) to check that no tuples are added so `even_delta2` and `odd_delta2` are empty (see line 13 in Fig. 8 (b)).

6.3. Implementing query solving

According to Fig. 7, and in addition to other available commands, the system accepts and distinguishes two types of queries: SQL queries, and those containing some ASSUME clause, i.e., hypothetical queries. The first ones can be directly submitted to the RDBMS in use; the Python script in this case is only used for facilitating the connection to the database.

Hypothetical queries require further computation, since assumptions involve to consider a local database, whose fixpoint must be computed in order to answer the query. Consider a database `db` loaded in the system. The process to solve a hypothetical query `sel_hyp` ≡ ASSUME `hypo`, . . ., `hypo query` is as follows:

- First of all, it is necessary to check if the local database required to compute `sel_hyp` is stratifiable. This means that the edges added to the original dependency graph due to the `hypo` parts of `sel_hyp`, in accordance to Definition 1, do not introduce cycles with some negatively labeled edge. If it is not stratifiable, an error message is shown and the process is finished; otherwise:
- If i is the stratum of `query`, `sel_hyp` is transformed into a standard query with no ASSUME clause, by applying repeatedly the function $tr(i, \texttt{db})$, introduced in Section 4.[2] As it was explained, each application of the function tr removes an ASSUME clause of `sel_hyp` and can introduce new auxiliary relations.
- Such auxiliary relations are incorporated to the dependency graph of the initial database with their corresponding edges.
- When all ASSUME clauses have been removed, the resulting dependency graph still admits a stratification, since these edges correspond to the dependencies associated to the `hypo` parts of `sel_hyp`, and the existence of a stratification for such extended graph was checked on the first step.
- Then a Python script is generated for computing the corresponding fixpoint. But notice that the whole database is not recomputed. The fixpoint is only calculated for the auxiliary relations obtained by the transformation of `sel_hyp`. These relations will be computed as new temporary tables and added to the current database.
- Finally, the answer is obtained by solving the (standard SQL) query resulting from the transformation, and the temporary tables are dropped.

Example 22. In order to illustrate the process of solving a hypothetical query, consider again the database of Example 20, and the relation `hyp_travel2` of Example 4. Instead of specifying `hyp_travel2` as a relation, the hypothetical query which defines it can be directly submitted:

```
HR-SQL>
  ASSUME
    SELECT * FROM bus WHERE bus.ori = 'VDE' UNION
      SELECT * FROM flight NOT IN link,
    SELECT 'MP','TFS',2.0 IN boat
  SELECT * FROM travel;
```

[2] The only difference is that `sel_hyp` is converted into the corresponding `query`σ, without ASSUME clauses, but it is not assigned to a new relation $\texttt{R}_h$ $\texttt{sch}_h$, as it is done when hypothetical relations are treated (see Fig. 6).

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

then, the system detects the ASSUME clause and processes it as a hypothetical query. First of all, the dependency graph of the database (depicted in Fig. 9) is enlarged with new edges coming from the `hypo` parts of this query, $\texttt{flight} \vec{\rightarrow} \texttt{link}$ and $\texttt{bus} \vec{\rightarrow} \texttt{link}$, which in this case admits a stratification. So, the transformation process for eliminating the ASSUME clause starts and some auxiliary relations are added:

```
boat_0(ori varchar(10), des varchar(10), time float) :=
 (SELECT 'SPC','TFN',2)   UNION (SELECT 'TFS','GMZ',1) UNION
 (SELECT 'GMZ','VDE',1.5) UNION (SELECT 'MP','TFS',2.0);

link_1(ori varchar(10), des varchar(10), time float) :=
 (SELECT * FROM flight) UNION
 (SELECT * FROM boat_0) UNION
 (SELECT * FROM bus   ) EXCEPT
    ((SELECT * FROM bus WHERE bus.ori = 'VDE' ) UNION
     (SELECT * FROM flight));

travel_2(ori varchar(10), des varchar(10), time float) :=
 (SELECT * FROM link_1) UNION
 (SELECT link_1.ori,travel_2.des,link_1.time + travel_2.time
  FROM link_1, travel_2 WHERE link_1.des = travel_2.ori );
```

The new relations `boat_0` and `link_1` come from `boat` and `link`, respectively, according to the first and second `hypo` parts of the hypothetical query. The relation `travel_2` is a renaming of `travel`, and it is added because `travel` appears in the `query` part of the hypothetical query and depends on both relations `link` and `boat` involved in the ASSUME clause.

Then, the original hypothetical query is transformed into the following one:

```
SELECT * FROM travel_2;
```

The fixpoint is calculated by generating the corresponding script, as explained before, but only for the new relations. During the computation, the relations `flight` and `bus` of the original database are used, but not recomputed. When needed, they are treated as belonging to previous strata. The dependencies involving exclusively these new relations (the only required dependencies to compute such fixpoint) correspond to those between the relations from which they come, that appear in Fig. 9, hence they are: $\texttt{boat\_0} \rightarrow \texttt{link\_1}$, $\texttt{link\_1} \rightarrow \texttt{travel\_2}$, and $\texttt{travel\_2} \rightarrow \texttt{travel\_2}$.

7. Experiments

In this section we analyze the performance of the system with respect to some factors. First, we develop a recursion analysis by targeting the system to different current state-of-the art relational systems. In particular, the reachability problem is analyzed by exploiting several sets of benchmark data structures. Regarding hypothetical reasoning, we include experiments testing hypothetical queries concerning aggregates over big databases.

For all the experiments we provide, each benchmark has been run 10 times, the maximum and the minimum measurements have been discarded, and then the average has been computed for the rest. All times are given in milliseconds and have been run on an Intel Core i7-3770 at 3.4 GHz and 8 GB RAM, running Windows 10 Pro 64 bit. As RDBMS's we have considered IBM DB2 and PostgreSQL as two widely-used and high-end systems.[3] The first one is a proprietary database system which traces its roots as early as 1970's and has evolved to cover many platforms, notably including mainframes, and sensible data handling. In turn, the open-source PostgreSQL database system evolved from the Ingres project in 1980's and it is available for Unix and Windows based platforms. Our experience reveals PostgreSQL as one of the most performant database systems when solving recursive queries. Both systems are accessed through the Python 3.3 ODBC driver `pyodbc` and are configured as by default. A time-out of 120 s has been imposed.

7.1. Recursion analysis

Here, we present some benchmarks in order to compare the performance of recursive relations in HR-SQL versus the corresponding native SQL formulations. This will lead us to introduce the benefits of the *semi-naïve differential optimization* [44,8,46] for linear recursive queries.

[3] The system can also operate with Access, MySQL, Oracle, SQL Anywhere, and SQL Server.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

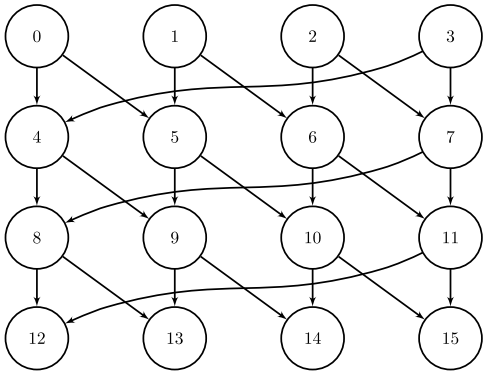


Fig. 12. Cylinder graph of size 16.

For this study we tackle the problem of reachability in a graph. The relation `reachable` in Example 3 is just a formulation of this problem. The measure of complexity of a graph must consider its size in terms of the number of nodes and edges, but they are not enough to provide an appropriate measure, because the topology of the graph itself (number of cycles, length of paths, etc.) has a deep impact on the performance of the reachability problem.[4] For this reason we analyze reachability on several kind of graphs covering a range of concrete topologies that provide a good idea of the performance in different scenarios. These topologies include graphs with and without cycles. For the last case we control the number and the length of such cycles. In particular we use the following kind of graphs:

- Full binary trees, i.e., binary trees where every node other than the leaves has two children. These graphs must have an odd number of nodes and are generated starting with a root (that is, a full binary tree of size 1) and successively adding pairs of left and right children to randomly chosen leaves.
- Cylinder graphs, that can be seen as a grid of nodes, where each node in a row is only connected with two nodes of the following row (except for the last row, whose nodes are not connected). We will consider an equal number M of rows and columns for these kind of graphs, getting $N = M * M$ nodes. Fig. 12 shows the cylinder graph for $N = 16$.
- Cylinder graphs with cycles of *type 1*, defined by adding a new edge from the last node to the first one in a cylinder graph. For the graph of Fig. 12, we would add an edge from 15 to 0.
- Cylinder graphs with cycles of *type 2*, defined by adding M edges from the nodes of the last row to those in the same column of the first row. For the graph of Fig. 12, the following set of edges would be added $\{12 \to 0, 13 \to 1, 14 \to 2, 15 \to 3\}$.
  Notice that the three cylinder graphs variants are obtained in a deterministic way.[5]

Fig. 13 shows the results of our experiments for the four types of graphs we have considered. Horizontal axes depict the size of the benchmark (number of nodes), and vertical axes depict times in milliseconds in a logarithmic representation (base 2). All curves are identified by a label *system-dbms* for each configuration, where *system* is either hr (HR-SQL) or sql (native SQL), and *dbms* is either postgresql or db2. Curves are shown up to the parameter for which the 120 s time-out has not been reached (for example, in graph (b), there is no available a fifth measure for hr-db2, and only the first 4 measures have been possible).

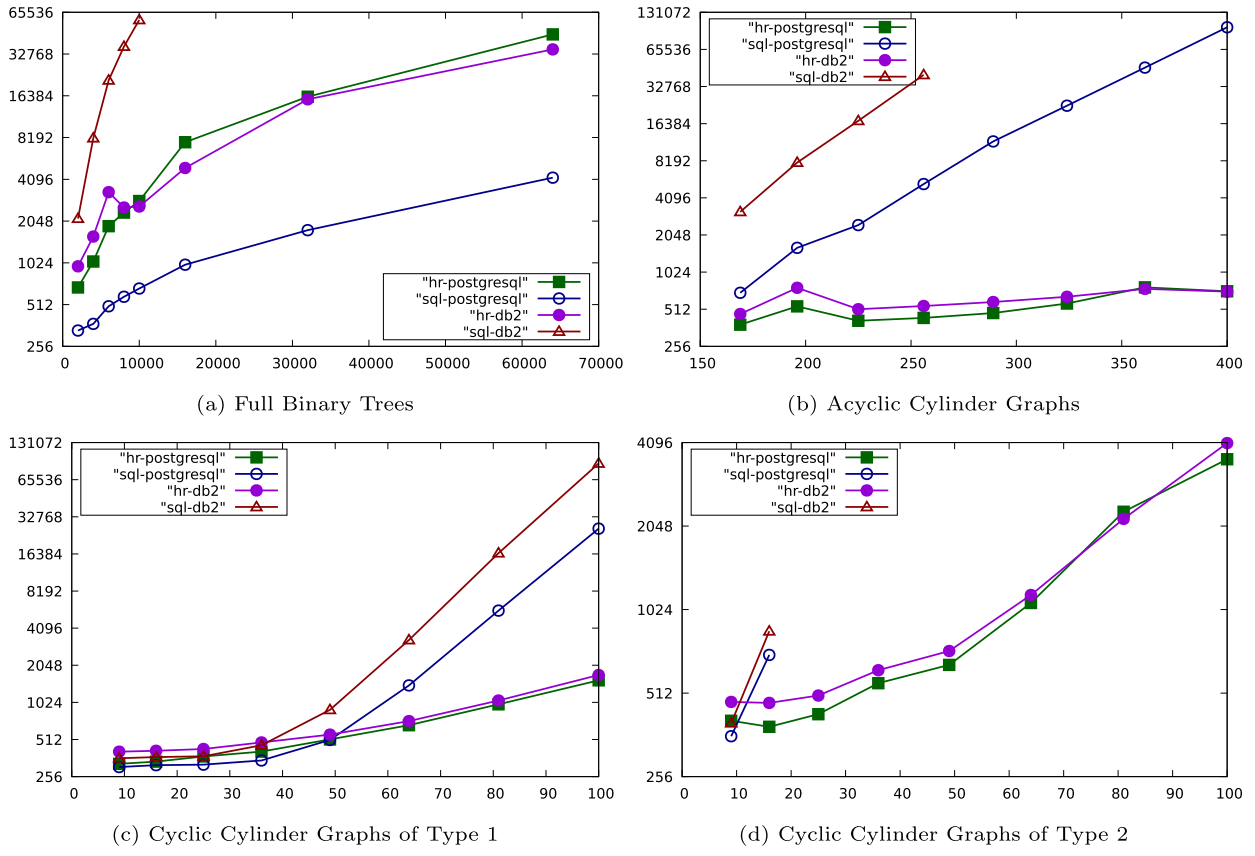
First graph (a) for full binary trees shows that sql-postgresql is the most performant configuration, most likely because the recursion implementation in PostgreSQL is optimized for hierarchical data structures.[6] The worst performant configuration in all graphs is the native DB2 SQL implementation. In turn, HR-SQL behaves much better than native queries for all cylinder graphs and full binary trees for sql-db2, with slight differences for its target RDBMS (where hr-postgresql is a bit better than db2-postgresql). This is explained because current RDBMS's do not discard duplicates during the evaluation of recursive queries as HR-SQL does, and they have to deal with exponentially increasing tables along recursion iterations. Finally, graphs (c) and (d) suggest that as long as the number of cycles in graphs increases, the performance gain of HR-SQL with respect to native implementations also increase.

[4] We did some preliminary experiments fixing the number of nodes and edges, and then generating edges in a random way (avoiding cycles when needed), but the performance results were highly influenced by the topologies and those experiments were not the most appropriate for the comparison.

[5] Another kind of cyclic cylinder graphs can be obtained in a natural way by extending those of type 2 adding cross edges from the nodes of the last row to the first one. In the example, the new edges would be $\{12 \to 1, 13 \to 2, 14 \to 3, 15 \to 0\}$. We have not included them in the paper because the performance experiments do not add any novelty with respect to the proposed ones.

[6] These structures are usual in database applications and have led to specific databases such as LDAP.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485



(a) Full Binary Trees

(b) Acyclic Cylinder Graphs

(c) Cyclic Cylinder Graphs of Type 1

(d) Cyclic Cylinder Graphs of Type 2

Fig. 13. Experiments of reachability in graphs.

7.2. What-if analysis on queries including aggregates over big data

In this section we analyze the performance of HR-SQL when dealing with hypothetical queries using aggregates over big databases. The next example shows a simple way for making such an experiment. We assume a relation `R1(a integer)` with values $\{(1), (2), \ldots, (N)\}$ and use it to define the following relations:

```
R2(a INTEGER) := SELECT * FROM R1;
Rh(a FLOAT) := ASSUME SELECT a + N FROM R1 IN R2 (SELECT avg(a) FROM R2);
R(a FLOAT, b INTEGER) := SELECT * FROM Rh, R2;
```

The relation `R2` also contains values from 1 to N. The relation `Rh` will be calculated by assuming the values $N+1$ to $N+N$ in `R2` and then obtaining the average value under that hypothesis. The final relation `R` is used to show that such an average has been correctly obtained while `R2` remains untouched under the ASSUME clause of `Rh` (the assumption has been locally done).

The native SQL version can be specified as follows: Table `R2` can be temporarily extended (with an `INSERT` instruction) with the tuples of the hypothesis for evaluating `Rh` and then restored (with `DELETE`) to its original state.

```
CREATE TABLE R2(a INTEGER);
CREATE TABLE Rh(a FLOAT);
CREATE TABLE R(a FLOAT, b INTEGER);

-- R2 as a copy of R1
INSERT INTO R2 SELECT ALL * FROM R1;

-- Temporary extension of R2 to simulate the hypothetical query
INSERT INTO R2 (SELECT ALL a + N FROM R1);

-- Rh evalutes the average of the modified R2
INSERT INTO Rh SELECT ALL avg(a) FROM R2;
```

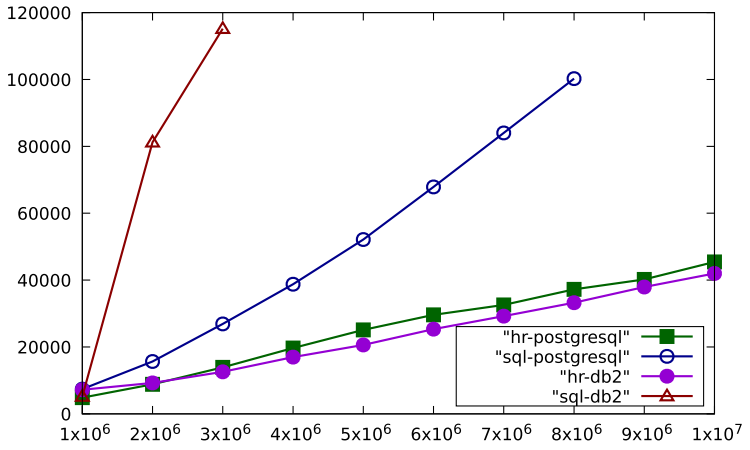


Fig. 14. Experiments of a *What-if* query with aggregates.

```
-- Restore R2 to its original state

  DELETE FROM R2 WHERE a IN (SELECT ALL a + N FROM R1);

-- R collects tuples with the value of Rh and the original R2
INSERT INTO R (SELECT * FROM Rh, R2);
```

Fig. 14 shows the results of the experiment for $N = 10^6$ up to $N = 10^7$. The times for HR-SQL generating code for both PostgreSQL and DB2 (`hr-postgresql` and `hr-db2` in the figure) have a linear growth with respect to `N`, and are much more efficient than the native versions (`sql-postgresql` and `sql-db2`). In fact, the native version for DB2 reaches the 120 s time-out for $N = 4 \cdot 10^6$, and PostgreSQL for $N = 9 \cdot 10^6$, while the HR-SQL versions support values for N well over 10^7. This difference in performance is explained by the cost of deletions in `R2`.

An extended study of performance by analyzing more examples, comparing alternative ways for computing the relations within the system, and also for contrasting the system with other RDBMS's can be found at http://gpd.sip.ucm.es/trac/gpd/raw-attachment/wiki/GpdSystems/HR-SQL2/AppendixB.pdf.

8. Conclusions and future work

The stratified fixpoint semantics for the hypothetical and recursive SQL language, proposed in this paper, has been shown appropriate and effective for building a system for current database systems as the performance analysis concludes.

With respect to recursion, our proposal extends current relational database systems by enabling mutual and non-linear recursion, termination detection, and duplicate elimination. As it is well-known, duplicate elimination requires sorting, which is a costly operation avoided in such recursive SQL's (for which the operator UNION requires the modifier ALL, the modifier DISTINCT in a SELECT clause and the EXCEPT operator are not allowed, and so on). Since we need to discard duplicate tuples for insertion with the EXCEPT operator, as depicted in the algorithm of Section 6.2, a natural improvement, that we conceive as future work, is to apply in-memory indexing of the involved result sets for small search keys. These keys can be identified as the candidate keys derived from explicit functional dependencies (as for instance already allowed in IBM DB2) and primary keys.

With respect to hypothetical queries and relations, our proposal is the first work that fully integrates hypothetical queries and relation definitions (including positive and negative assumptions) with recursion in an SQL relational database. The work in the last published version of [39] has no nested assumptions and rely on an in-memory hypothetical Datalog implementation, and [31] can be understood as a way to efficiently arrange data arrays as a result of a query for decision analysis. Indeed, such "hypothetical" queries can be alternatively expressed with SQL queries, while this is not true in our approach. In addition, we provide support for mutually recursive hypothetical relations, and a novel formalization of the whole system based on an extended stratified fixpoint semantics.

The algorithm in Section 6.2 improves our former work [4] as already suggested there and confirmed by our experiments: extracting out those non-recursive fragments of SQL code from the iterative computation. In addition, by maximizing the number of strata, it is possible that a relation is iteratively built with as many cycles as needed (in the best case, either only a relation is in a stratum, or several relations requiring the same number of cycles are in a stratum). Both improvements have a noticeable impact on performance as the number of uses of the costly operator EXCEPT decreases (recall that this operator is usually implemented with sorting). Although we have proposed a generic algorithm to compute recursive and

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

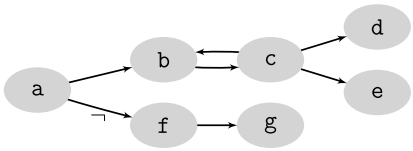


Fig. A.1. Dependency graph example.

hypothetical queries, a natural improvement as future work might include to detect particular cases for which more efficient methods do exist. For instance, already-known linear recursion optimizations [32] can be applied to our work.

Using temporary tables is adequate as they are in-memory data structures and confirmed by the performance results. Only when the available memory exhausts due to large result sets, then the persistent table space is used. However, the use of cursors in a fourth generation language –4GL– (as Oracle's PL/SQL and IBM's SQL PL) for intermixing fetching with SQL queries can be analyzed and might provide performance gains, in particular when coupled with indexing. In addition, we might extend this work by allowing not only materialized relations, but also regular views. This way, the user could choose what relations are not to be materialized. For instance, relations such that its materialization implies a performance or space penalty. For this, table functions (see, e.g., IBM DB2 concepts) can be used as a natural construction to build HR-SQL query results. This way, the 4GL language can be used to implement the desired relation outcome in the table function body.

An assumption over a large table implies creating a new table with the modified semantics. This involves a penalty in both space and performance. However, we can consider as future work an optimization developed to overcome such a situation. RDBMS transactions can be used to locally modify relations, and also creating (intensional) views at the RDBMS-level including modifications with UNION (for IN assumptions) and EXCEPT (NOT IN assumptions) operators.

Notice that the modification of a database relation (table) in the underlying RDBMS can cause inconsistencies since the tables are not automatically recomputed, but on-demand. Though this is the very same behavior of RDBMS's when dealing with materialized views, a future direction in order to fully integrate HR-SQL into an RDBMS is to have the possibility of automatically maintaining the database consistency (e.g., using triggers). This maintenance involves in general the recomputation of the database fixpoint. But, using the dependency graph, it is easy to determine the subset of relations that must be calculated, instead of computing the whole fixpoint for the database. In addition, those relations may not need to be recomputed from scratch. In addition, it is straightforward to modify the algorithm introduced in Section 6.2 to get a lazy evaluation of such relations, performing iterations only when new values are demanded.

Another point worth of future work is to seamlessly integrate our extensions into an RDBMS by profiting from 4GL's and completely integrate query solving and view maintenance into the RDBMS, therefore eliding the need of the HR-SQL front-end. This way, prepared SQL statements (i.e., precompiled statements with parameters) can be used, which should also improve performance with respect to our current dynamic approach. Finally, despite several representative RDBMS's are supported, the system can be easily tweaked to deal with others.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank the anonymous referees for helping us in improving this paper with their suggestions. Also, thanks are given to the institutions supporting our work in the form of projects and grants. In particular, this research has been partially supported by the Spanish project TIN2017-86217-R (CAVI-ART-2) and Madrid regional project S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the EU.

Appendix A. Stratification algorithm

This Appendix shows the stratification algorithm used by the system in detail. The aim of this algorithm is to minimize the number of relations in each strata to improve the efficiency of the fixpoint computation. For instance, for the dependency graph of Fig. A.1, a possible stratification can assign stratum 1 to the relations $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ and stratum 2 to $\{\mathtt{f}, \mathtt{g}\}$. But, intuitively it is easy to see that only `b` and `c` must belong to the same stratum due to the mutual dependency between them.

The system HR-SQL executes the following algorithm to obtain a stratification of a database `db`[7]:

[7] This algorithm uses the library *ugraphs* of SWI-Prolog and the module *scc* implemented by Markus Triska. It is accessible from http://www.logic.at/prolog/scc.pl.

- Compute the *strongly connected components* C from DG_{db}. Negative labels are not relevant initially, but once the components are evaluated, it must be checked if there exists some cycle with a negatively labeled edge. In such a case, db is not stratifiable and the computation stops. For the example of Fig. A.1 the components are {a}, {f}, {g}, {b, c}, {d} and {e}.
- *Collapse each strongly connected component* obtaining a new graph with a node for each component, C, and with an edge from C to C' if and only if C contains a relation R and C' contains a relation R′, such that there is an edge from R to R′ in DG_{db}. In our example, the component {b, c} can be collapsed to the node bc, and the rest to its single element. The new graph has the edges $\{a \to bc, bc \to d, bc \to e, a \to f, f \to g\}$.
- Obtain a *topological sorting* for the resulting graph. In our example we can get $a < f < g < bc < e < d$.
- Uncollapse the nodes of such a sorting for obtaining a topological sorting for the strongly connected components, and enumerate them in ascending order. In our example, we get $\{a\} < \{f\} < \{g\} < \{b, c\} < \{e\} < \{d\}$. Then, the expected stratification $str(a) = 1$; $str(f) = 2$; $str(g) = 3$; $str(b) = str(c) = 4$; $str(e) = 5$; $str(d) = 6$ is obtained.

References

[1] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.
[2] M. Alvarez-Picallo, A. Eyers-Taylor, M. Peyton Jones, C.-H.L. Ong, Fixing incremental computation, in: L. Caires (Ed.), Programming Languages and Systems, Springer International Publishing, 2019, pp. 525–552.
[3] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, J. Sánchez-Hernández, Implementing a fixpoint semantics for a constraint deductive database based on hereditary Harrop formulas, in: Proceedings of the 11th International ACM SIGPLAN Symposium of Principles and Practice of Declarative Programing (PPDP'09), ACM Press, 2009, pp. 117–128.
[4] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, J. Sánchez-Hernández, Formalizing a broader recursion coverage in SQL, in: Symposium on Practical Aspects of Declarative Languages (PADL'13), in: LNCS, vol. 7752, 2013, pp. 93–108.
[5] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, J. Sánchez-Hernández, R-SQL: An SQL Database System with Extended Recursion, Electronic Communications of the EASST, vol. 64, Programming and Computer Languages, 2013.
[6] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, J. Sánchez-Hernández, Incorporating hypothetical views and extended recursion into sql database systems, in: K. Mcmillan, A. Middeldorp, G. Sutcliffe, A. Voronkov (Eds.), LPAR-19, in: EPiC Series, vol. 26, EasyChair, 2014, pp. 9–22.
[7] M. Arenas, L. Bertossi, Hypothetical temporal reasoning in databases, J. Intell. Inf. Syst. 19 (2) (2002) 231–259.
[8] I. Balbin, K. Ramamohanarao, A generalization of the differential approach to recursive query evaluation, J. Log. Program. 4 (3) (1987) 259–262.
[9] A. Balmin, T. Papadimitriou, Y. Papakonstantinou, Hypothetical queries in an OLAP environment, in: Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00, Morgan Kaufmann Publishers Inc., 2000, pp. 220–231.
[10] E. Bertino, B. Catania, R. Gori, Enhancing the expressive power of the U-datalog language, Theory Pract. Log. Program. 1 (1) (2001) 105–122.
[11] A. Bonner, M. Kifer, A logic for programming database transactions, in: J. Chomicki, G. Saake (Eds.), Logics for Databases and Information Systems, in: The Springer International Series in Engineering and Computer Science, vol. 436, Springer US, 1998, pp. 117–166.
[12] A.J. Bonner, Hypothetical datalog: negation and linear recursion, in: The ACM Symposium on the Principles of Database Systems (PODS), 1989, pp. 286–300.
[13] A.J. Bonner, L.T. McCarty, Adding negation-as-failure to intuitionistic logic programming, in: E.L. Lusk, R.A. Overbeek (Eds.), Logic Programming, in: Proceedings of the North American Conference, The MIT Press, 1989, pp. 681–703.
[14] W. Chen, Programming with logical queries, bulk updates and hypothetical reasoning, IEEE Trans. Knowl. Data Eng. 9 (1995) 587–599.
[15] H. Christiansen, T. Andreasen, A practical approach to hypothetical database queries, in: Transactions and Change in Logic Databases, in: LNCS, vol. 1472, Springer, 1998, pp. 340–355.
[16] S. Cohen, J.Y. Gil, E. Zarivach, Datalog programs over infinite databases, revisited, in: Database Programming Languages: 11th International Symposium, DBPL 2007, Vienna, Austria, September 23-24, 2007, Revised Selected Papers, Springer Berlin Heidelber, Berlin, Heidelberg, 2007, pp. 32–47.
[17] S. Dietrich, Understanding Relational Database Query Languages, Prentice Hall, 2001.
[18] S.J. Finkelstein, N. Mattos, I.S. Mumick, H. Pirahesh, Expressing Recursive Queries in SQL, Technical Report, ISO, 1996.
[19] H. Garcia-Molina, J.D. Ullman, J. Widom, Database Systems - The Complete Book, second ed., Pearson Education, 2009.
[20] M. Golfarelli, S. Rizzi, What-if simulation modeling in business intelligence, Int. J. Data Warehous. Min. 5 (4) (2009) 24–43.
[21] T.J. Green, S.S. Huang, B.T. Loo, W. Zhou, Datalog and recursive query processing, Found. Trends Databases 5 (2) (November 2013) 105–195.
[22] T. Griffin, R. Hull, A framework for implementing hypothetical queries, in: SIGMOD Conference, 1997, pp. 231–242.
[23] Z. Hilkevich, Oracle SQL Tricks and Workarounds: Expert Guide to Oracle SQL Excellence, AuthorHouse, Bloomington, IN, USA, 2011.
[24] M.A.W. Houtsma, P.M.G. Apers, Algebraic optimization of recursive queries, Data Knowl. Eng. 7 (1991) 299–325.
[25] W.H. Inmon, Building the Data Warehouse, QED Information Sciences, Inc., 2005.
[26] ISO/IEC, SQL:2011 ISO/IEC 9075(1-4,9-11,13,14):2011 Standard, 2011.
[27] O. Kaser, C.R. Ramakrishnan, S. Pawagi, On the conversion of indirect to direct recursion, ACM Lett. Program. Lang. Syst. 2 (1–4) (1993) 151–164.
[28] R.A. Kowalski, Logic for data description, in: Logic and Data Bases, 1977, pp. 77–103.
[29] I.S. Mumick, H. Pirahesh, Implementation of magic-sets in a relational database system, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, vol. 23, 1994, pp. 103–114.
[30] S. Nieva, F. Sáenz-Pérez, J. Sánchez-Hernández, Formalizing a constraint deductive database language based on hereditary Harrop formulas with negation, in: FLOPS'08, in: LNCS, vol. 4989, Springer-Verlag, 2008, pp. 289–304.
[31] Oracle, Database Data Warehousing Guide, 11g Release 2 (11.2) Part Number E25554-01, 2011.
[32] C. Ordonez, Optimization of linear recursive queries in SQL, IEEE Trans. Knowl. Data Eng. 22 (2) (2010) 264–277.
[33] K. Ramamohanarao, J. Harland, An introduction to deductive database languages and systems, VLDB J. 3 (2) (1994) 107–122.
[34] R. Reiter, Towards a logical reconstruction of relational database theory, in: On Conceptual Modelling (Intervale), 1982, pp. 191–233.
[35] P.Z. Revesz, Safe query languages for constraint databases, ACM Trans. Database Syst. 23 (1) (March 1998) 58–99.
[36] P.Z. Revesz, Introduction to Constraint Databases, Springer, 2002.
[37] F. Sáenz-Pérez, ACIDE: an integrated development environment configurable for LaTeX, PracTeX J. 2007 (3) (August 2007), available at http://acide.sourceforge.net.
[38] F. Sáenz-Pérez, Implementing tabled hypothetical datalog, in: Proceedings of the 25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI'13, 2013, pp. 596–601.
[39] F. Sáenz-Pérez, Datalog educational system, http://des.sourceforge.net/, 2016.
[40] J. Shepherdson, Negation in logic programming, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Kaufmann, 1988, pp. 19–88.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485

[41] M. Stonebraker, K. Keller, Embedding expert knowledge and hypothetical data bases into a data base system, in: The 1980 ACM SIGMOD International Conference on Management of Data, SIGMOD '80, ACM, 1980, pp. 58–66.
[42] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pac. J. Math. 5 (1955) 285–309.
[43] G. Terracina, N. Leone, V. Lio, C. Panetta, Experimenting with recursive queries in database and logic programming systems, Theory Pract. Log. Program. 8 (2) (2008) 129–165.
[44] J. Ullman, Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies), Computer Science Press, 1995.
[45] H.J. Watson, B.H. Wixom, The current state of business intelligence, Computer 40 (9) (2007) 96–99.
[46] C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, R. Zicari, Advanced Database Systems, Morgan Kaufmann Publishers Inc., 1997.
[47] Y. Zhang, H. Chen, H. Sheng, Z. Wu, Applying hypothetical queries to e-commerce systems to support reservation and personal preferences, in: Proceedings of the 11th International Database Engineering and Applications Symposium, IDEAS '07, IEEE Computer Society, 2007, pp. 46–53.
[48] G. Zhou, H. Chen, Y. Zhang, Hypothetical queries on multidimensional dataset, in: S. Wang, L. Yu, F. Wen, S. He, Y. Fang, K.K. Lai (Eds.), BIFE, IEEE Computer Society, 2009, pp. 539–543.

Please cite this article in press as: S. Nieva et al., HR-SQL: Extending SQL with hypothetical reasoning and improved recursion for current database systems, Inf. Comput. (2019), https://doi.org/10.1016/j.ic.2019.104485