Tiresias: The Database Oracle for How-To Queries*

Alexandra Meliou Dan Suciu

Computer Science & Engineering
University of Washington
Seattle, WA, USA
{ameli,suciu}@cs.washington.edu

ABSTRACT

How-To queries answer fundamental data analysis questions of the form: "How should the input change in order to achieve the desired output". As a *Reverse Data Management* problem, the evaluation of how-to queries is harder than their "forward" counterpart: hypothetical, or *what-if* queries.

In this paper, we present *Tiresias*, the first system that provides support for how-to queries, allowing the definition and integrated evaluation of a large set of constrained optimization problems, specifically Mixed Integer Programming problems, on top of a relational database system. Tiresias generates the problem variables, constraints and objectives by issuing standard SQL statements, allowing for its integration with any RDBMS. The contributions of this work are the following: (a) we define how-to queries using possible world semantics, and propose the specification language *TiQL* (for Tiresias Query Language) based on simple extensions to standard Datalog. (b) We define translation rules that generate a Mixed Integer Program (MIP) from TiQL specifications, which can be solved using existing tools. (c) Tiresias implements powerful "data-aware" optimizations that are beyond the capabilities of modern MIP solvers, dramatically improving the system performance. (d) Finally, an extensive performance evaluation on the TPC-H dataset demonstrates the effectiveness of these optimizations, particularly highlighting the ability to apply divide-and-conquer methods to break MIP problems into smaller instances.

1. INTRODUCTION

A *How-To* query [21] computes hypothetical updates to the database that achieve a desired effect on one or several indicators, while satisfying some global constraints. Key Performance Indicators (KPI), or simply indicators, is an industry term for a measure that evaluates the company's performance according to some metric [8]. For example, a shipping company fills in orders by contracting with several suppliers. One KPI is the total quantity per order and supplier: the smaller the indicator, the lesser the company's exposure to order delays due to delivery delays from the suppliers. The KPIs can be computed using standard SQL queries on the relational database. However, company planners are constantly looking for ways to improve these indicators. In *What-if* queries [18, 6] the user describes hypothetical changes to the database, and the system computes the effect on the KPIs. This scenario requires the decision maker to specify the hypothetical change, and the query will compute the effect on the indicator. *How-to* queries are the opposite: the decision maker specifies the desired effect on the indicators, and the system proposes some hypothetical updates to the database that achieve that effect. How-to queries are important in business modeling and strategic planning, and are computationally very expensive. For example, *how do I reduce the total quantity per order and supplier to be at most 50?*: to answer this query, the system needs to propose major updates to the database of outstanding orders, and present it to the user.

How-to queries are a special case of constrained optimization, in particular Linear Programming, and Integer Programming [11]. Several mature LP/IP tools exists, and are used extensively in many applications. However, mapping a *how-to* query to a linear or integer program is a non-trivial task. The program needs to model the data in terms of integer and/or real variables, and the constraints (both database constraints and constraints on the KPI's) as inequalities. There exists a semantic gap between the relational data model, where the data is stored, and the linear algebra model of the LP tools. For that reason, strategic planning in enterprises today is done outside of, and separate from the operational databases that supports that planning.

In this paper, we present TIRESIAS, the first how-to query engine, which integrates relational database systems with a linear programming engine. In TIRESIAS, users write a declarative query in the TIRESIAS query language (TiQL), which is a datalog-based language for how-to queries. The key concept in TiQL is that of *hypothetical tables*, which form a hypothetical database, HDB. The rules in TiQL are like datalog rules, but have a non-deterministic semantics, and express the actions the system has to consider while answering the how-to query, as well as the constraints that the system needs to meet. Possible actions expressible in TiQL include modifying quantity values, removing tuples, creating new tuples, computing aggregates. Constraints can represent business rules or desired outcomes on KPI's. TiQL's non-deterministic semantics consists of the collection of all possible worlds on the HDB that satisfy all constraints. TIRESIAS chooses one possible world that optimizes a specified objective function. While users write TiQL programs, they are only exposed to the relational data model and to familiar query constructs, such as selections, joins, aggregates, etc. However, TiQL was designed such that everything it can express can be mapped into an linear program, or, more precisely,

*This work was partially funded by NSF IIS-0911036, IIS-0915054, and IIS-0713576.

SIGMOD âĂŹ12, May 20–24, 2012, Scottsdale, Arizona, USA.

into a *mixed integer program*, MIP, which has both real and integer variables. We describe TiQL in Sect. 3.

The translation of TiQL into MIP is quite non-trivial, and is an important technical contribution of this paper. At a high level, the translation proceeds by mapping each tuple in a hypothetical relation to one or several integer variables, and mapping each unknown attribute in a tuple of a hypothetical relation to a real variable. The translation needs to take into account the *lineage* (provenance) of each tuple, and first represent it as a Boolean expression over the Boolean variables that encode the non-deterministic choices made by TiQL, then covert it into integer variables and constraints. Similarly, it needs to trace all unknown real variables, and compute aggregates correspondingly. Dealing with duplicate elimination and with key constraints further add to the complexity of the translation. To overcome these challenges we use provenance semi-rings [16], and the recently introduced semi-modules for aggregation provenance [4]. The number of integer and real variables created by the translation is large, and needs to be managed inside the relational database system. We describe the translation in Sect. 4.

As expected, even robust MIP solvers cannot scale to typical database sizes. Another key contribution of this paper is a suite of optimizations that reduce the MIP problem sufficiently in order to be within reach of today's standard MIP solvers; these optimizations enabled us to scale TIRESIAS up to 1M tuples. The most powerful optimization is a technique that splits the input problem into several independent problems. Partitioning splits one TiQL query into several, relatively small MIP problems, which can be solved independently by the MIP solver. This results in huge performance improvements, allowing TIRESIAS to scale to up to 1M tuples. We also implemented other optimizations, like variable or matrix elimination: while some of these are also done by the MIP solver, we found that by performing them early (at the TiQL query level as opposed to the MIP level), each such optimization results in a ten-fold performance increase. We describe optimizations in Sect. 5.

Finally, we report experimental evaluation of TIRESIAS on TPC-H data in Sect. 6.

Tiresias was a mythical prophet from Thebes, in ancient Greece. He was so wise, that the gods blinded him for accessing and revealing their secrets. The TIRESIAS system is an oracle on top of a database, allowing users to discover how to make major updates to the database in order to improve key performance indicators.

The main contributions of this work are as follows:

- We describe the TIRESIAS system architecture for formulating and evaluating *how-to* queries over relational databases, by using an MIP solver (Sect. 2).
- We introduce a new, simple declarative language, TiQL, which can express complex *how-to* queries, in a datalog-like notation (Sect. 3).
- We describe a method for translating TiQL programs in MIP programs (Sect. 4).
- We describe several optimization techniques for reducing the size of the resulting MIP program, and improve overall execution time Sect. 5.
- Finally, we perform extensive performance evaluation of the TIRESIAS system over the TPC-H dataset (Sect. 6).

2. ARCHITECTURE OF TIRESIAS

The architecture of TIRESIAS is shown in Fig. 1. Users write *how-to* queries in TiQL(Sect. 3). This query is parsed, then translated into a Mixed Integer Program (MIP). The translation is non-trivial, and we explain its logic in Sect. 4. From the system's perspective, the translation has three parts. First, generate the set of *Core tables*, whose tuples are used as templates for the variables and inequalities in MIP (Sect. 4.2). Second, generate partitioning information: after a static analysis of the TiQL program, the system decides how to partition the MIP into smaller fragments that can be handled by the solver (Sect. 5.1). Both the core tables and the partition information are stored in the relational database. Third, the MIP constructor reads the core tables and the partition information, and generates a separate problem file for each MIP subproblem (Sect. 5.2). The MIP solver is a black-box module that processes subproblems one at a time, and produces a solution for each.

Finally, the results are read by a separate module (Solution Processor), which presents the solution to the user and interacts with her. This module is not presented in the paper.

3. TiQL

After a brief background on datalog, we illustrate TiQL by examples, then define it formally.

3.1 Background on Datalog

We denote a relational schema $\mathbf{R} = (R_1, \ldots, R_m)$, where R_i is a relation name (aka predicate name). We assume that each relation name has at least one key; if no explicit key is given, then the set of all its attributes always form a key. A database instance is denoted I, and consists of m relations, one for each relation symbol, $I = (R_1^I, \ldots, R_m^I)$.

We briefly review non-recursive datalog with negation [22, 23]. In datalog, predicate names are partitioned into EDBs, and IDBs (extensional and intensional database). A single datalog rule is:

$$\mathtt{P}(\bar{x}) \ \ \texttt{:-}\ \mathtt{body}$$

where $\mathtt{P}(\bar{x})$ is called the head of the rule, `P` is an IDB predicate, $\bar{x}$ are called head variables, and `body` is a conjunction of positive or negated predicates, or arithmetic comparisons (e.g. `x < y`). We will allow only EDB's to occur under negation. The rule must be *safe*, meaning that every variable must occur in a positive relational atom on the body. Using a simple extension, see e.g. [12], we also allow aggregate operators in the rule's head, for example the datalog rule

```
SuppSum(city,sum(qnt))  :- Supp(sk,city)
                         & LineItem(ok,pk,sk,qnt)
```

computes for each city the total quantity on all line items with suppliers in that city. We only consider `sum` and `count` in this paper. A *datalog program*, Q, is a list of rules. In this work, we consider only non-recursive datalog programs, which means that the rules can be *stratified*, i.e. the IDBs can be ordered $(P_1, \ldots, P_n)$ such that any rule having P_k in the head may have only the IDBs $P_1, \ldots, P_{k-1}$ in the body (in addition to any EDB predicates). Evaluation of a stratified datalog program proceeds in a sequence of steps, one for each stratum. Denote Q_k the set of rules in stratum k, i.e. that have the predicate P_k in the head. Given an EDB instance I, denote $J_k = (R_1^I, \ldots, R_m^I, P_1^J, \ldots, P_k^J)$: that is, J_k consists of the EDBs and the IDBs up to stratum k. Then the program computes the sequence $J_1 = I, J_2, \ldots, J_n$, where $P_k^J = Q_k(J_{k-1})$, $k = 1, n$. The last instance, J_n, consists of the entire EDB and IDB instance.

3.2 TiQL by Examples

A shipping company keeps records of open order requests in a TPC-H inspired schema [2]. Each order consists of several line items, stored in a relation `LineItem(ok,pk,sk,quant)` with key

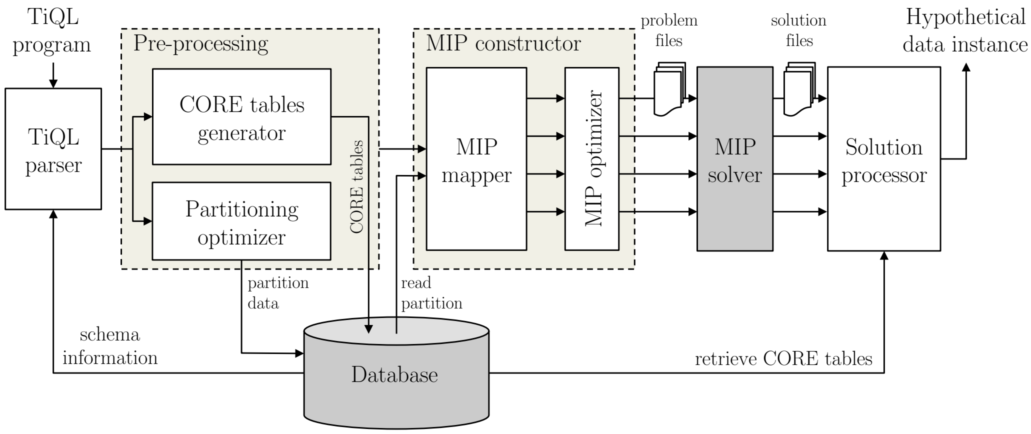

Figure 1: The TIRESIAS system architecture. Both the database and the MIP solver as treated as black box components, so TIRESIAS can be integrated with different DBMSs and MIP tools.

`(ok,pk,sk)`: a record means that order `ok` contains a line item where part `pk` will be delivered from supplier `sk`. Due to a change in corporate policies, the company decides to limit the total quantity per order and supplier to at most 50. The total quantity per order and supplier is a very simple example of a KPI, and can be computed as:

```
OrderSum(ok,sk,sum(qnt))   :- LineItem(ok,pk,sk,qnt)
```

The problem is: *how* should we modify the database such that all values `c` in `OrderSum(ok,sk,c)` are `<= 50`?

Our first TiQL example in Fig. 2 is very simple (we will show a more realistic program in a moment): it keeps the same line items per order, but decreases their quantities, to ensure `[c <= 50]`. The main output is a *hypothetical table*, called `HLineItem(ok,pk,sk,q?)`, which is simply a copy of `LineItem` but with updated quantities. Note that the quantity attribute has a trailing question mark, `q?`; such an attribute is called *unknown*, and it means that it needs to be computed by TIRESIAS. We explain now the rules in Fig. 2. The first rule copies `LineItem` to `HLineItem`, choosing `q?` nondeterministically; next, a constraint asserts that quantities can only decrease `[q? <= qnt]`; finally, the program computes the indicator `HOrderSum(ok,sk,c?)` and asserts that `[c? <= 50]`. Intuitively, this program makes a nondeterministic choice for the new quantities `q?`, while satisfying all constraints. There are many possible solutions: the *objective function* on the last line gives a criteria for choosing one solution, namely a solution that minimizes the sum of all quantity differences.

A variant on this program is shown in Fig. 3, and is also very simple: instead of decreasing the quantities, this programs keeps the quantities unchanged but drops `LineItems` altogether, until `[c <= 50]`. The TiQL rule

```
HLineItem(ok,pk,sk,qnt) :< LineItem(ok,pk,sk,qnt)
```

is called a *reduction rule* and it means that `HLineItem` is a nondeterministically chosen subset of `LineItem`.

Finally, we show a much more realistic TiQL program in Fig. 4. Here we will ensure that the database changes such that each order keeps the same total quantity. The previous two programs reduced the total quantity per order, which may not be acceptable in practice. To satisfy the KPI, the program changes the suppliers in some items, while keeping everything else unchanged. The hypothetical table `HChooseS(ok,pk,sk,sk')` selects a new supplier `sk'` to replace `sk`. The rule computing `HChooseS` checks that the new supplier `sk'` supplies the product `pk`, so that we can fill in the order from the new supplier. Notice that `HChooseS` has two different keys: the first key `(ok,pk,sk)` says that only one new supplier `sk'` is chosen for each line item; the second key ensures that `(ok,pk,sk')` will be a key for `HLineItem` (otherwise it would not be a valid hypothetical change for `LineItem`). Finally, the objective function requires that the total number of replacements of a supplier `sk` with another supplier `sk'` be minimized: this is equivalent to maximizing the number of tuples that keep the same supplier, so the objective is to maximize the number of tuples in `HChooseS` where `sk=sk'`.

```
HTABLES:
  HLineItem(ok,pk,sk,q?)        KEY:(ok,pk,sk)
  HS(ok,pk,sk,qnt,q?)           KEY:(ok,pk,sk)
  HOrderSum(ok,sk,c?)           KEY:(ok,sk)
RULES:
  HLineItem(ok,pk,sk,q?)      :- LineItem(ok,pk,sk,qnt)
  HS(ok,pk,sk,qnt,q?)         :- HLineItem(ok,pk,sk,q?)
                               & LineItem(ok,pk,sk,qnt)
  [q? <= qnt]                 <- HLineItem(ok,pk,sk,q?)
                               & LineItem(ok,pk,sk,qnt)
  HOrderSum(ok,sk,sum(q?))    :- HLineItem(ok,pk,sk,q?)
  [c? <= 50]                  <- HOrderSum(ok,sk, c?)
MINIMIZE(sum(qnt-q?))   :-  HS(ok,pk,sk,qnt,q?)
```

Figure 2: Example query Q_1: A TiQL program that decreases quantities in order to achieve a desired KPI.

In summary, a *how-to* query is written in TiQL using a set of rules that define a collection of *hypothetical tables*, which together form the *Hypothetical Database, HDB*. The rules that populate these tables have a non-deterministic semantics, and may involve inserting or deleting tuples, updating attribute values, or any combination thereof. All these hypothetical changes are described declaratively. Constraints are specified on attributes, or on aggregates. In addition, the user specifies an objective function that the system needs to optimize (minimize or maximize).

3.3 Formal Definition of TiQL

We now define formally TiQL's syntax and semantics. A TiQL program, `T`, has three parts: the *HTable declarations*; the *rules*; and the *objective function*. We denote $R_1, \ldots, R_m$ the EDB predicates.

```
HTABLES:
  HLineItem(ok,pk,sk,q)          KEY:(ok,pk,sk)
  HOrderSum(ok,sk,q?)            KEY:(ok,pk)
RULES:
  HLineItem(ok,pk,sk,qnt)      :< LineItem(ok,pk,sk,qnt)
  HOrderSum(ok,sk,sum(qnt))    :- HLineItem(ok,pk,sk,qnt)
  [c? <= 50]                   <- HOrderSum(ok,sk,c?)
MAXIMIZE(count(*)) :- HLineItem(ok,pk,sk,qnt)
```

Figure 3: Example query Q_2: A TiQL program that deletes line items in order to achieve a desired KPI.

```
HTABLES:
  HChooseS(ok,pk,sk,sk')         KEY:(ok,pk,sk),(ok,pk,sk')
  HLineItem(ok,pk,sk,q)          KEY:(ok,pk,sk)
  HOrderSum(ok,sk,q?)            KEY:(ok,pk)
RULES:
  HChooseS(ok,pk,sk,sk')       :- PartSupp(pk,sk')
                                & LineItem(ok,pk,sk,qnt)
  HLineItem(ok,pk,sk',qnt)     :- HChooseS(ok,pk,sk,sk')
                                & LineItem(ok,pk,sk,qnt)
  HOrderSum(ok,sk,sum(qnt))    :- HLineItem(ok,pk,sk,qnt)
  [c? <= 50]                   <- HOrderSum(ok,sk,c?)
MAXIMIZE(count(*)) :-  HChooseS(ok,pk,sk,sk)
```

Figure 4: Example query Q_3: A TiQL program that chooses new suppliers in order to achieve a desired KPI.

The HTable declaration lists the HDB predicates $\mathtt{HP}_1, \ldots, \mathtt{HP}_n$, and for each $\mathtt{HP}_k$ gives its attributes and its keys. The first key is called the *primary key*. Each attribute is either *known* or *unknown*: the latter are identified by a trailing `?`, and may not be used in a key. By convention, all attributes of an EDB relation are known.

A rule has one of the following forms:

$\mathtt{HP}(\bar{x})$	`:-`	`body`	`(Deduction Rule)`
$\mathtt{HP}(\bar{x})$	`:<`	`body`	`(Reduction Rule)`
`[arithm-pred]`	`<-`	`body`	`(Constraint Rule)`

The rules are similar to datalog rules (Sect. 3.1), with the following differences. The rule's head is either some $\mathtt{HP}_k$, or an arithmetic predicate of the form `[var1 <= var2]` or `[var <= const]` or `[const <= var]` etc, where `var1,var2,var` are variables, and `const` is a constant. To each constraint rule we associate a fresh HDB symbol `HP`, whose attributes are all variables occurring in the rule, and with no explicit key[1]. For example, for the constraint `[q? <= qnt] <- HS(ok,pk,sk,qnt,q?)` we create the new HDB symbol `HC(ok,pk,sk,qnt,q?)`: intuitively, this predicate is first populated, then the constraint checked in each row.

We require the program to be *stratified*, in the sense that a rule having $\mathtt{HP}_k$ in the head may only have the HDB's $\mathtt{HP}_i$ with $i < k$ in the body; HDB predicate may not be negated in the body. We denote $\mathtt{T}_k$ the set of rules in stratum k, i.e. where the head predicate is $\mathtt{HP}_k$. All rules in $\mathtt{T}_k$ must be of the same type, either all are deduction rules, or all are reduction rules, or $\mathtt{T}_k$ consists of only one constraint rule; consequently, we call $\mathtt{HP}_k$ a *deductive*, or a *reductive*, or a *constraint* predicate; `HLineItem` is deductive in Fig. 2, and is reductive in Fig. 3, while `HC` introduced above is a constraint predicate.

[1]Thus, the key is the set of all attributes.

In every rule, variables are labeled as either known or unknown, the latter having a trailing `?`. A known variable may only occur in positions of known attributes, and an unknown variable may only occur in positions of an unknown attributes. By definition, an aggregate is unknown, hence the attribute where it occurs must be unknown. Intuitively, known variables are always bound to some values from the EDB, while unknown variables will have some non-deterministically chosen values. In every rule, the known variables must be *safe*: every known variable must occur in a positive relational predicate in the body. Let $\mathtt{HP}_k$ be the predicate in the rule's head. We call an attribute in the head *safe* if it contains a safe variable (known or unknown); otherwise we call the attribute *unsafe*; intuitively, the value of an unsafe attribute must be chosen non-deterministically, while that of a safe attribute comes from the rule's body. For example, consider the rules for `HLineItem` and for `HS` of query Q_1 in Fig. 2. In both predicates the attribute `q?` is unknown. However, in the first rule the attribute is unsafe, because `q?` does not occur in the body, while in the second it is safe. What that means is that the first rule must choose `q?` non-deterministically; the second rule will simply copy it from the body. If multiple rules have $\mathtt{HP}_k$ in the head position, then we require an attribute to have the same status, safe or unsafe, in all rules, and we denote S_k the set of safe attributes of $\mathtt{HP}_k$.

Finally, the objective function is a deduction rule where the head predicate is either `MINIMIZE` or `MAXIMIZE`, with a single attribute. By convention, we consider the last HDB predicate, $\mathtt{HP}_n$, to be the objective function, i.e. either `MINIMIZE` or `MAXIMIZE`; its key is the empty set of attributes, `()`, in other words, this relation may have a single value.

We now define the semantics of a TiQL program `T`. Let $I = (\mathtt{R}_1^I, \ldots, \mathtt{R}_m^I)$ be an EDB instance, and $J = (\mathtt{HP}_1^J, \ldots, \mathtt{HP}_n^J)$ be an HDB instance. The semantics is a set of *possible worlds* J. Denote $J_k = (\mathtt{R}_1^I, \ldots, \mathtt{R}_m^I, \mathtt{HP}_1^J, \ldots, \mathtt{HP}_k^J)$, for $k = 1, \ldots, n$. We compute the sequence $J_0 (= I), J_1, J_2, \ldots, J_n$, where each J_k is obtained, non-deterministically, from J_{k-1} by computing the new relation, $\mathtt{HP}_k^J$. Drop all unsafe attributes from the head predicates of $\mathtt{T}_k$, denoting $\mathtt{T}_k^s$ the resulting rules; all rules in $\mathtt{T}_k^s$ have the same head predicate, and its attributes are S_k. Thus, $\mathtt{T}_k^s(J_{k-1})$ means "the result of applying the datalog rules $\mathtt{T}_k^s$ to the instance J_{k-1}". We do the following, for $k = 1, \ldots, n$:

- If $\mathtt{HP}_k$ is a reduction predicate, then $\mathtt{HP}_k^J$ is chosen nondeterministically such that (a) it satisfies the key constraints, and (b) $\Pi_{S_k}(\mathtt{HP}_k^J) \subseteq \mathtt{T}_k^s(J_{k-1})$. In other words, compute the datalog rules $\mathtt{T}_k^s$, remove tuples non-deterministically to satisfy the key constraints, and for each remaining tuple fill in non-deterministically some values in the unsafe attributes. In example Q_2 of Fig. 3, the rule `HLineItem(ok,pk,sk,qnt) :<LineItem(ok,pk,sk,q)` is evaluated by removing non-deterministically tuples from `LineItem(ok,pk,sk,q)` (possibly all tuples).
- If $\mathtt{HP}_k$ is a deduction predicate, then $\mathtt{HP}_k^J$ is chosen nondeterministically such that it satisfies (a) and (b) above, and furthermore: (c) if A is the primary key of $\mathtt{HP}_k$, then $\Pi_A(\mathtt{HP}_k^J) = \Pi_A(\mathtt{T}_k(J_{k-1}))$. In other words, as we remove tuples non-deterministically we must ensure that every value of the primary key is kept in the table. In example Q_3 (Fig. 4), the rule `HChooseS(ok,pk,sk,sk') :- ...` is evaluated by choosing a subset such that every value `(ok,pk,sk)` is kept in the output, while also ensuring that all values `(ok,pk,sk')` are unique.

- If $\texttt{HP}_k$ is a constraint predicate, then we simply compute $\texttt{HP}_k^J = \texttt{T}_k(J_{k-1})$.

Finally, once we finish computing an instance J we check all constraints, by iterating over all tuples of a constraint predicate $\texttt{HP}_k$ and checking if it satisfies the corresponding `[arithm-pred]`. If yes, then we call J a *possible world*; if not, then we reject J.

Recall that $\texttt{HP}_n$ is the objective function, and that $\texttt{HP}_n^J$ has a single value: we denote the latter with $\texttt{val}(J)$.

DEFINITION 1. *An instance J returned by the non-deterministic algorithm above is called a* possible world. *$\mathcal{W}_{\texttt{T}}$ denotes the set of possible worlds. The* answer *to the* TiQL *program* T *is a possible world J for which* $\texttt{val}(J)$ *is minimized (or, maximized, respectively).*

We illustrate some (fragments of) possible worlds for `HLineItem` for the three example queries, and the values of their objective function in Fig. 5.

3.4 Discussion

Several formalisms exist whose semantics is based on possible worlds. For example *disjunctive datalog* (DL) is a powerful extension of datalog used for knowledge representation and reasoning; DLV is an advanced implementation of disjunctive datalog [19]. For example, key constraints can be represented in DL. Another line of research is on *incomplete databases* [17]; a recent system, ISQL [6], has a key repair construct as a primitive. For example, consider the following TiQL program:

```
HTABLE S(x,y) KEY (x)
RULE S(x,y) :- R(x,y)
```

which selects a subset of R by enforcing the first attribute to be a key. This program can be expressed as the following "shifted" normal datalog program:

```
S(x,y)      :- R(x,y), not NS(x,y)
NS(x,y)     :- R(x,y), S(x,z), y != z
```

One can check that the stable models of this program (called *answer sets* in [19]) are precisely the possible worlds of the TiQL program. Similarly, it can be expressed in ISQL [6] as[2]:

```
S(x,y) :- R(x,y) REPAIR BY KEY x
```

Their expressive power is not the same, however. DLV expresses all problems in Σ_2^P, while TiQL is in NP (as follows from the next section). The evaluation problem in ISQL is also NP-hard, but TiQL is strictly more expressive, for example ISQL doesn't seem to be able to express two key constraints as in:

```
HTABLE S(x,y) KEY (x) (y)
RULE S(x,y) :- R(x,y)
```

Here S computes a 1-1 matching between the x values and the y values that is total on the x values.

In addition, TiQL can represent unknown values, which are essential in *how-to* queries, because of their need to modify numerical quantities; neither DL nor ISQL do that.

The most important difference between TiQL and previous formalisms, however, is the fact that TiQL returns one possible world, while all other formalisms compute the certain answers, or possible answers (called *cautious reasoning* or *brave reasoning* in DL [19]). Thus, in TiQL we want to find *one* hypothetical database, in contrast to finding which tuple belongs to *all*. This semantics is what is needed for *how-to* queries. In order to compute such a possible world, TiQL translates the problem into a Mixed Integer Program, as we describe next.

[2]We adapted the ISQL syntax to a datalog notation.

4. MAPPING TiQL TO MIP

TIRESIAS evaluates a TiQL program by converting it into one, or several, Mixed Integer Programs (MIP), then invoking a MIP solver. This translation is non-trivial, because TiQL is a declarative language, with set semantics, while a MIP consists of linear equations over real and integer variables, and there is a large semantic gap between the two formalisms. We describe here the translation at the conceptual level, then present several optimizations in the next section. The translation proceeds in three steps. First, it creates the *core tables*, and several integer and real variables for tuples in the core tables. Second, it computes provenance expressions using semi-rings and semi-modules and creates combined constraints (involving both Boolean and numerical variables). Third, it linearizes all combined constraints to obtain only linear constraints. Before presenting the translation, we give a brief overview of semi-rings, semi-modules, and illustrate the main idea in using these to produce the linear program.

4.1 Background: Semi-Rings and Semi-Modules

Semi-rings for provenance expressions were introduced in [16]. Recently, Amsterdamer et al. [4] extended them to semi-modules to express the provenance of aggregate operators. A semi-module consists of a commutative semi-ring, whose elements are called *scalars*, a commutative monoid whose elements are called *vectors* and a *multiplication-by-scalars* operation that takes a scalar x and a vector u and returns a vector $x \otimes u$. We use two semi-rings in our translation: the Boolean semi-ring $(\mathbb{B}, \vee, \wedge, 0, 1)$, and the natural numbers semi-ring $(\mathbb{N}, +, \cdot, 0, 1)$: we need the former to capture the provenance of tuples in the HDB (since TiQL has set semantics) and we need the latter to express aggregate provenance. The monoid we consider is[3] $\texttt{SUM} = (\mathbb{R}_+, +, 0)$. It becomes an $\mathbb{N}$-semi-module by defining $x \otimes u = x \cdot u$ (standard multiplication).

Notice that SUM cannot be extended to a $\mathbb{B}$ semi-module (this was proven formally in [4]); if we tried, then the following distributivity law of semi-module fails: $x \otimes u = (x \vee x) \otimes u \neq x \otimes u + x \otimes u$. In database terms, one cannot do duplicate elimination before aggregation. Therefore, in TiQL we need both semi-rings $\mathbb{B}$ and $\mathbb{N}$: the first to compute the tuple provenance under set semantics, and the second to compute the provenance of aggregate expressions.

Assume for now that each tuple t_j in the database instance I is annotated with a Boolean variable X_j, and with a natural number x_j (later we will use annotations for only some tuples and values): the two variables have the same value, $X_j = x_j = 0/1$, and mean that the tuple is absent/present. Furthermore, each attribute A of each the t_j is annotated with a real variable $v_{j,A}$. Let Q be a query given as a datalog rule, and let $t \in Q(I)$ be an output tuple. We need the following concepts from the theory of provenance: The *set provenance expression* for the output tuple t is an expression in the Boolean semiring $\mathbb{B}$ over the variables X_j; its value is 0/1, signifying whether t is present in the output; we denote set provenance with Φ_t. The *bag provenance expression* of t is an expression in the semiring $\mathbb{N}$ over the variables x_i; its value is a number representing the multiplicity of t in the output; we denote the bag provenance φ_t. Finally, for each aggregate expression `sum(A)` or `count(*)` in the head of the query Q, the *aggregation provenance* of that value is an expression in the semi-module, using variables x_i and $u_{i,B}$ for

[3]We currently restrict the unknown values to be non-negative numbers.

Database:

LineItem

ok	*pk*	*sk*	*quant*
1	P15	S10	22
1	P24	S10	30
1	P22	S55	24
2	P15	S43	45
3	P24	S50	14
3	P22	S43	11

PartSupp

pk	*sk*
P15	S10
P15	S13
P15	S43
P24	S10
P24	S50
P24	S55
P22	S55
P22	S43

(a)

$[Q_1]$ *Possible worlds:*

HLineItem-1 [13]

ok	*pk*	*sk*	*quant*
1	P15	S10	22
1	P24	S10	20
1	P22	S55	24
2	P15	S43	42
3	P24	S50	14
3	P22	S43	11

HLineItem-2 [2]

ok	*pk*	*sk*	*quant*
1	P15	S10	20
1	P24	S10	30
1	P22	S55	24
2	P15	S43	45
3	P24	S50	14
3	P22	S43	11

HLineItem-3 [2]

ok	*pk*	*sk*	*quant*
1	P15	S10	21
1	P24	S10	29
1	P22	S55	24
2	P15	S43	45
3	P24	S50	14
3	P22	S43	11

HLineItem-4 [7]

ok	*pk*	*sk*	*quant*
1	P15	S10	22
1	P24	S10	28
1	P22	S55	24
2	P15	S43	40
3	P24	S50	14
3	P22	S43	11

...

$[Q_2]$ *Possible worlds:*

HLineItem-1 [4]

ok	*pk*	*sk*	*quant*
1	P15	S10	22
1	P22	S55	24
2	P15	S43	45
3	P22	S43	11

HLineItem-2 [5]

ok	*pk*	*sk*	*quant*
1	P24	S10	30
1	P22	S55	24
2	P15	S43	45
3	P24	S50	14
3	P22	S43	11

HLineItem-3 [3]

ok	*pk*	*sk*	*quant*
1	P24	S10	30
3	P24	S50	14
3	P22	S43	11

HLineItem-4 [2]

ok	*pk*	*sk*	*quant*
1	P15	S10	22
1	P22	S55	24
2	P15	S43	45
3	P24	S50	14

...

$[Q_3]$ *Possible worlds:*

HLineItem-1 [4]

ok	*pk*	*sk*	*quant*
1	P15	S13	22
1	P24	S10	30
1	P22	S55	24
2	P15	S43	45
3	P24	S50	14
3	P22	S55	11

HLineItem-2 [4]

ok	*pk*	*sk*	*quant*
1	P15	S10	22
1	P24	S55	30
1	P22	S43	24
2	P15	S43	45
3	P24	S50	14
3	P22	S43	11

HLineItem-3 [5]

ok	*pk*	*sk*	*quant*
1	P15	S10	22
1	P24	S50	30
1	P22	S55	24
2	P15	S43	45
3	P24	S50	14
3	P22	S43	11

HLineItem-4 [4]

ok	*pk*	*sk*	*quant*
1	P15	S10	22
1	P24	S55	30
1	P22	S43	24
2	P15	S43	45
3	P24	S50	14
3	P22	S43	11

...

(b)

Figure 5: An example EDB instance and some possible worlds for example queries Q_1, Q_2, and Q_3. Each possible world is annotated on the top right with the evaluation of the corresponding objective function.

some attribute B; it denotes the value of the aggregate; we denote $\alpha_{t,A}$ for the aggregation provenance. We omit the formal definitions of $\Phi_t, \varphi_t, \alpha_{t,A}$ (they can be found in [16, 4]) and illustrate with an example instead.

Consider the following database instance:

R

A	B	C	
a_1	b_1	c_1	x_1
a_1	b_2	c_1	x_2
a_1	b_3	c_2	x_3

S

C	D	
c_1	$10\ (= v_{1,D})$	y_1
c_2	$20\ (= v_{2,D})$	y_2

and the query:

```
Q(x,sum(u),count(*)) :-R(x,y,z) & S(z,u)
```

Then the result has a single tuple, $t = (a_1, 10+10+20, 3) = (a_1, 40, 3)$, and we denote its attributes with A, D, E; we have:

$$\begin{aligned}
\Phi_t &= X_1Y_1 \vee X_2Y_1 \vee X_3Y_3 \\
\varphi_t &= x_1 \cdot y_1 + x_2 \cdot y_1 + x_3 \cdot y_3 \\
\alpha_{t,D} &= x_1y_1 \otimes v_{1,D} + x_2y_1 \otimes v_{1,D} + x_3y_2 \otimes v_{2,D} \\
\alpha_{t,E} &= x_1y_1 \otimes 1 + x_2y_1 \otimes 1 + x_3y_2 \otimes 1
\end{aligned}$$

($\otimes$ is regular multiplication; we prefer to use $\otimes$ to indicate that it is the semi-module multiplication by scalars.) Φ_t is the set-provenance, φ_t is the bag-provenance, and $\alpha_{t,D}$, $\alpha_{t,E}$ are the aggregation provenances. Here X_i, Y_j represent the Boolean counterparts of the integer variables x_i, y_j. If all Boolean variables are set to 1, then the set provenance is $\Phi_t = 1$; the bag-provenance is $\varphi_t = 3$ (it computes the multiplicity of the tuple); and the aggregation provenances are $\alpha_{t,D} = 40$, $\alpha_{t,E} = 3$. The three formulas keep track, concisely, of how the output would change if we modify the input (replace it with a different possible world); for example, if we set $x_2 = 0$ (remove the second R-tuple) and set $v_{1,D} = 5$, then $\alpha_{t,D} = 5+0+20 = 25$ and $\alpha_{t,E} = 1+0+1 = 2$. Note that φ_t is not a linear expression, since it contains products of variables $x_1 \cdot y_1$: this is why we cannot use the semiring $\mathbb{N}$ to translate TiQL programs to MIP. Instead, we use $\mathbb{B}$, and will show how the set provenance Φ_t can be *linearized*. The aggregation provenance expressions $\alpha_{t,D}$ and $\alpha_{t,E}$ are not linear either. To linearize them, we first rewrite the query by separating the join from the aggregations:

```
T(x,y,z,u) :-R(x,y,z) & S(z,u)
Q(x,sum(u),count(*)) :-T(x,y,z,u)
```

The intermediate table `T` is:

T

A	B	C	D	
a_1	b_1	c_1	10	Z_1
a_1	b_2	c_1	10	Z_2
a_1	b_3	c_2	20	Z_3

The Boolean provenance variables Z_1, Z_2, Z_3 are:

$$Z_1 = X_1Y_1 \qquad Z_2 = X_2Y_1 \qquad Z_3 = X_3Y_2$$

Now the aggregation provenance expressions become:

$$\begin{aligned}
\alpha_{t,D} &= z_1 \otimes v_{1,D} + z_2 \otimes v_{1,D} + z_3 \otimes v_{2,D} \qquad (1) \\
\alpha_{t,E} &= z_1 \otimes 1 + z_2 \otimes 1 + z_3 \otimes 1
\end{aligned}$$

Here z_i represent the integer counterparts of Z_i. We will show how expressions like these can be linearized.

4.2 Core Tables

The first step of the translation is to compute core tables. For each HDB predicate $\mathtt{HP}_k$ we construct a *core table* $\mathtt{CORE\_HP}_k$, $k = 1, \ldots, n$. Its attributes are all the known attributes of $\mathtt{HP}_k$ (i.e. those without `?`) and no key constraints. We create a dalalog program to compute the core tables, as follows: For each TiQL rule whose head predicate is $\mathtt{HP}_k$ we create a datalog rule where the head predicate is $\mathtt{CORE\_HP}_k$, and where every HDB predicate in the body is replaced with the corresponding core table. For example, the core tables for Fig. 4 are computed as follows:

```
CORE_HChooseS(ok,pk,sk,sk')
                          :- PartSupp(pk,sk')
                           & LineItem(ok,pk,sk,qnt)
CORE_HLineItem(ok,pk,sk',qnt)
                          :- CORE_HChooseS(ok,pk,sk,sk')
                           & LineItem(ok,pk,sk,qnt)
CORE_HOrderSum(ok,sk)     :- CORE_HLineItem(ok,pk,sk,qnt)
CORE_HC(ok,sk)            :- HOrderSum(ok,sk,c?)
```

Note that the unknown attribute `q?` has been dropped from table `CORE_HOrderSum(ok,sk)`. The last core predicate, `CORE_HC`, is for the constraint predicate `HC`, which we introduced for the constraint `[c? <= 50] <-..`. These queries are translated into SQL in a straightforward way, and executed in the RDBMS; their results are stored in new tables in the database. Every tuple in any possible world $J \in \mathcal{W}_{\mathtt{T}}$ is a core tuple (after projecting out the unknown attributes), but in general, the converse does not hold. Constraints like key constraints or TiQL constraint rules may make it impossible for some core table tuples to belong to any possible world.

4.3 Provenance

The next step is to compute provenance expressions for all core tuples and their unknown attribute values, and to generate a set of constraints. These constraints contain Boolean constraints, linear (in)equality constraints, and (in)equality constraints over `SUM` semimodule expressions. We call them *combined constraints*, or CC; the next step will be to linearize them.

We start by creating the following variables:

- For each tuple t_i in a core relation, create two variables: a Boolean X_i and a binary integer variable x_i. Their meaning is: if $X_i = x_i = 1$ then t_i is present from the possible world, if $X_i = x_i = 0$ then t_i is absent. During linearization we will eliminate the Boolean variable X_i, but for now we will use both. The system adds the following *key* CC's (both are linear constraints):

$$0 \leq x_i \leq 1 \qquad \forall k_j : \sum_{i, key(x_i)=k_j} x_i \leq 1$$

  In the last line, k_j ranges over key values in the core tables: the constraint says that at most one tuple with a given key may be selected. Let $x^{\mathtt{HChooseS}}_{(ok,pk,sk,sk')}$ denote the binary integer variable associated to a tuple `HChooseS(ok,ps,sk,sk')` in Fig. 4. Then we have the following two constraints:

$$\forall ok, pk, sk : \sum_{sk'} x^{\mathtt{HChooseS}}_{ok,pk,sk,sk'} \leq 1$$
$$\forall ok, pk, sk' : \sum_{sk} x^{\mathtt{HChooseS}}_{ok,pk,sk,sk'} \leq 1$$

- For each tuple t_i in a core relation and each unknown attribute A of the corresponding HDB create a real valued variable $u_{i,A}$. The system adds the constraint $u_{i,A} \geq 0$. TIRESIAS currently does not support negative unknown values.

Next, the system generates the following provenance expressions for each HDB predicate $\mathtt{HP}_k$, and each tuple t_i in $\mathtt{CORE\_HP}_k$. Below we assume that the provenance annotation of EDB tuples is 1 (i.e. the tuple is always present), and the value provenance of any known attribute is a constant.

Boolean Provenance Compute the set-provenance expression Φ_i for t_i (Sect. 4.1).

Value Provenance For each safe, unknown attribute A of $\mathtt{HP}_k$ create a value-provenance expression $\alpha_{i,A}$, as follows. If A is copied from some other attribute B of some core tuple t_j, then $\alpha_{i,A} = u_{j,B}$; if A is an aggregate `sum(B)` or `count(*)`, then $\alpha_{i,A}$ is the aggregation provenance (Sect. 4.1).

Finally, we add the following CC's:

Value CC For each safe, unknown attribute A of $\mathtt{HP}_k$, add the following CC:

$$u_{i,A} = \alpha_{t,A}$$

It simply says how the value in that attribute position was obtained. For example, referring to Fig. 2, we have the following CCs, for all values ok, pk, sk:

$$u^{\mathtt{HS}}_{(ok,pk,sk),\mathtt{q}} = u^{\mathtt{HLineItem}}_{(ok,pk,sk),\mathtt{q}}$$
$$u^{\mathtt{HOrderSum}}_{(ok,pk,sk),\mathtt{s}} = \alpha_{(ok,pk,sk)}$$

The first says that `q?` is copied from `HLineItem` to `HS`; the second says that `s?` in `HOrderSum` is an aggregate (here $\alpha_{(ok,pk,sk)}$ is the aggregation provenance expression for `sum(q?)`, derived from the rule in Fig. 2).

Boolean CC For each reduction and deduction predicate $\mathtt{HP}_k$ add the CC:

$$X_i \Rightarrow \Phi_{t_i} \tag{2}$$

It says that tuple t_i can only exist if its provenance expression is true. Notice that t_i does not need to exist if Φ_{t_i} is true, because $\mathtt{HP}_k$ is defined by a reduction rule, thus any subset of tuples returned by the query may be chosen for inclusion in $\mathtt{HP}_k$.

For deduction predicates only, add the following CC, for every value pk of the primary key:

$$\bigvee_{pkey(t_i)=pk} \Phi_{t_i} \Rightarrow \bigvee_{pkey(t_i)=pk} X_i \tag{3}$$

It says that every value of the primary key must be included. Referring again to Q_3 (Fig. 4), we have:

$$X^{\mathtt{HChooseS}}_{ok,pk,sk,sk'} \Rightarrow 1$$
$$1 \Rightarrow \bigvee_{sk'} X^{\mathtt{HChooseS}}_{ok,pk,sk,sk'}$$

The first CC is vacuous, because `HChooseS` is defined directly in terms of EDB tables, and therefore the provenance expression for every tuple in `HChooseS` is 1: the CC says that we can choose any set of tuples. The second rule says that we must choose at least one tuple (ok, pk, sk, sk') for every value of (ok, pk, sk).

Constraint Rule CC If $\mathtt{HP}_k$ is a constraint HDB predicate, then let the corresponding contraint rule be `[arithm-pred] <- body`. The `[arithm-pred]` is an inequality predicate: assume it is `A <= B`, where we used `A,B` to refer to attributes in $\mathtt{HP}_k$ or are constants. Then add the following CC:

$$\Phi_{t_i} \Rightarrow (u_{i,A} \leq u_{i,B}) \tag{4}$$

Here $u_{i,A}, u_{i,B}$ are either variables, or constants. For example, consider the constraint rule `[c? <= 50] <-..` in Q_3 of Fig. 4, to which we associated the HDB predicate `HC(ok,sk,c?)`. Then we create the CC:

$$\Phi_{(ok,sk)} \Rightarrow (v^{\mathtt{HC}}_{(ok,sk)} \leq 50)$$

Here $\Phi_{(ok,sk)}$ is the set-provenance formula for a tuple in `HC`, and is equal to $X^{\mathtt{HLineItem}}_{(ok,pk,sk)}$ (see the rule defining the constraint `[c? <= 50]` in Fig. 4).

4.4 Linearization

The last step is to convert the set of Combined Constraints introduced in the previous section into an equivalent set of Linear Inequality Constraints. Our task is to replace the constraints Eq. 2, Eq. 3, Eq. 4 with linear constrains, and also to linearize the provenance expression for aggregation, illustrated in Eq. 1. We proceed in several steps.

First, we make the following transformation on all Boolean formulas. We create a fresh Boolean variable for each subformula, and add new equality constraints such that each Boolean formula involves either only $\wedge$, or only $\vee$. For example, suppose $\Phi = (X_1 \wedge Y_1) \vee (X_2 \wedge Y_1) \vee (X_3 \wedge Y_2)$ is a formula occurring in one of the CC's. Then we replace it with a new Boolean variable U, and add the constraints:

$$\begin{aligned} &U = Z_1 \vee Z_2 \vee Z_3 \\ &Z_1 = X_1 \wedge Y_1 \;\; Z_2 = X_2 \wedge Y_1 \;\; Z_3 = X_3 \wedge Y_2 \end{aligned}$$

As a consequence, the CC's Eq. 2 and Eq. 3 will now be of the form $X \Rightarrow Y$, where X and Y are Boolean variables. The CC Eq. 4 has the form $X \Rightarrow (u \leq v)$ where u, v are real variables, or constant expressions.

Second, we replace each Boolean variable X_i with its binary integer counterpart x_i. Every implication $X \Rightarrow Y$ becomes a simple linear inequality constraint $x \leq y$. Equality constraints are handled as follows.

Disjunction: If $Y = X_1 \vee X_2 \vee \ldots \vee X_n$

Then we create the following linear constraints:

$$\begin{aligned} &(a)\;\; \forall i,\; y \geq x_i \\ &(b)\;\; y \leq \sum_i x_i \end{aligned}$$

Conjunction: If $Y = X_1 \wedge X_2 \wedge \ldots \wedge X_n$

Then we create the following linear constraints:

$$\begin{aligned} &(a)\;\; \forall i,\; y \leq x_i \\ &(b)\;\; y \geq \sum_i x_i - (n-1) \end{aligned}$$

Each CC of the form $X \Rightarrow (u \leq v)$ is replaced with $x \otimes u \leq x \otimes v$.

Finally, we show how to linearize provenance expressions for aggregations. For that we adapt a method from [11]. Given an expression:

$$\alpha = b_1 \otimes u_1 + b_2 \otimes u_2 + \ldots + b_k \otimes u_k$$

We create a real variable v_j for each term $b_j \otimes u_j$, and assume a number M that is an upper bound on all u_j's. We create the following linear constraints for v_j:

$$\begin{aligned} &v_j \leq u_j \\ &v_j \leq b_j M \\ &v_j \geq u_j - (1 - b_j)M \end{aligned}$$

Then α can be written as the sum of all v_j: $\alpha = \sum_{j=1}^{k} v_j$

5. OPTIMIZATIONS

In the previous section, we showed how to use the provenance expressions of the HDB tuples in order to create linear constraints. These comprise a complete mixed integer program that can be solved with dedicated solvers. Even though these modern tools have evolved through many years of research and have become increasingly more efficient, constrained optimization remains in general a hard problem, and even modern solvers will "choke" when data grows very large. However, TIRESIAS can exploit DB specific information in order to improve the system performance. We will describe in this section two separate optimizer modules, both of which result in significant performance improvements.

5.1 Pre-processing Optimizer

Optimization problems are often comprised by independent components that do not share variables. Assume the following simple linear program:

$$\begin{aligned} &max \sum_i x_i \\ s.t:\;\; &x_1 + x_2 \leq 50 \\ &x_3 + x_4 \leq 50 \\ &x_i \geq 0 \end{aligned}$$

Variables x_1 and x_2 do not share any constraints with variables x_3 and x_4, and due to the linearity of the objective function, the above LP is equivalent to the two below:

$$\begin{aligned} &max \sum_i x_i \\ s.t:\;\; &x_1 + x_2 \leq 50 \\ &x_i \geq 0 \end{aligned} \qquad\qquad \begin{aligned} &max \sum_i x_i \\ s.t:\;\; &x_3 + x_4 \leq 50 \\ &x_i \geq 0 \end{aligned}$$

The two problems can be solved independently, and their solutions can be combined to form the solution for the first problem. The pre-processing optimizer takes advantage of this divide-and-conquer technique to split very large problems into several smaller ones.

5.1.1 Partitioning

TIRESIAS's first optimizer module analyzes the TiQL program directly to identify possible horizontal partitionings of the data that can be treated independently in the optimization problem. It does so by identifying consistent *partitioning attributes* across the EDBs and HDBs of the TiQL program. If a set of attributes is chosen as a partitioning set for a relation, then the relation is horizontally partitioned according to the different values of the attribute set. Note that the partitioning is not physically enforced, but rather a list of the distinct partitioning values is stored so the MIP mapper can later retrieve one at a time. For example, the `LineItem` table of Fig. 5 can be divided in 3 partitions on attribute `ok`: `ok=1`, `ok=2`, and `ok=3`. Out of the possible options, the optimizer will select the one with the finest granularity, i.e. the one that leads to more partitions. For example, between the options to partition on `ok` or `(ok,sk)`, the optimizer will select the second.

We present the pseudocode of our partitioning algorithm in algorithm 1. The algorithm keeps updating the partitioning of each relation while iterating through the TiQL program, until a stable solution is reached. The runtime of the algorithm is not relevant to data complexity, as the input is a TiQL program and not a data instance. For every TiQL rule, the algorithm ensures that the following conditions hold:

- The head HDB and the body subgoals should have the same partitioning set: Let `A(x,y) :- B(x,y,z)` be a TiQL rule. If `B` is partitioned on `x` and `A` is partitioned on `(x,y)`, we set the partitioning of both to `x`. If `B` is partitioned on `x` but `A` is partitioned on `y` (due to other rules), then there is no valid partitioning and the algorithm will return false.
- An aggregated attribute cannot be part of a partitioning, because then the calculated value would be incorrect. For example, if `HLineItem` from Fig. 3 was partitioned on `qnt`,

```
Algorithm: Finds a maximal partitioning
Input: a TiQL program
Output: false is partitioning is impossible
Set the partitioning of each relation to be the union of its keys
Remove any aggregates from the partitioning
//Repeat until there is no more change
while something changed do
    forall TiQL rules do
        S = set of all subgoals and head HDB
        P = S[0].partitioning
        for i=1 to S.length do
            //S(P)= attributes of S[i] that correspond
            to P
            S[i].partitioning = S[i].partitioning ∩ S(P)
            if S.partitioning = ∅ AND S[i] is HDB then
                return false
            if S[i] is HDB then
                P = S[i].partitioning
            if S.partitioning changed then
                i = 0
send partitioning to DB
return true
```

Algorithm 1 Finds a set of partitioning attributes if one exists, based on the TiQL rules.

then the sum in `HOrderSum` would be incorrect within a partition.

Queries $Q1$ and Q_2 from the examples in Sect. 3.2 can be partitioned on `(ok,sk)`. That simply means that two tuples with different `(ok,sk)` values will never appear in the same constraint, and thus can be separated into different problems. The algorithm only removes attributes from a partitioning when those are not shared among all the relations in a rule. Therefore, if a partitioning set becomes empty, the partitioning fails. The algorithm executes on example query Q_3 roughly as follows: `HLineItem` can be partitioned on `ok,sk'` due to the 3rd rule, and the partitioning is propagated to `HChooseS` due to the 2nd rule. From the set `(ok,sk')`, `HChooseS` shares attribute `ok` with `LineItem`, and `sk'` with `PartSupp`, but no attribute is shared with both. If these were HDBs, the partitioning would fail, but EDBs are not required to be partitioned: their variables are deterministic, so it is ok if they are repeated across subproblems.

Partition grouping. Our experiments in Sect. 6 demonstrate that partitioning results in radical improvements in query execution time, and TIRESIAS is even shown to handle MIP problems that modern solvers to handle. Large data sizes quickly render the solvers useless, but if a partitioning does exist, size restrictions are entirely eliminated.

However, "extreme" partitioning can have disadvantages. Splitting large problems into the smallest subdivisions makes the system I/O bound (writing and reading the MIP problem and solution files). This effect can be alleviated by *partition grouping*: instead of generating a MIP problem for each partition, we can group multiple together. We study the effect of this parameter in the experimental section.

5.2 MIP Optimizer

The MIP mapper within the MIP constructor component applies the mapping rules presented in Sect. 4 to generate an MIP problem from the set of TiQL statements. This initial model is then processed by the MIP optimizer, which can reduce the problem size. Modern solver to an extend already perform some of these optimizations, but it is much more efficient for TIRESIAS to handle them, as is verified by our experimental results.

Elimination of Key Related Constraints. This optimization checks whether a key CC can be eliminated. The optimizer analyzes the TiQL program and the present functional dependencies, and determines whether key constraints are automatically satisfied and therefore can be dropped from the MIP model. In example Q_2, the following constraint is always satisfied and can be removed:

$$\forall ok, pk, sk : \sum_{qnt} x^{\texttt{HLineItem}}_{ok,pk,sk,qnt} \leq 1$$

Variable Elimination. Variable elimination identifies variables that are not necessary because they always evaluate to a specific value, or are just equal to another attribute. Even though modern solvers apply this optimization as well, they are at a disadvantage: they need to analyze the entire, possibly very large, MIP problem. In contrast, TIRESIAS can easily determine unnecessary variables from the provenance, which is significantly faster.

Matrix Elimination. Assume the TiQL rule `A(x) :- B(x,y)`. Then, the provenance of each head tuple a_i is $a_i = b_{i1} \vee b_{i2} \vee \ldots \vee b_{ik_i}$. This lineage expression will generate the constraints $\forall i, \forall j \in [1, k_i], a_i \geq b_{ij}$. To represent these provenance constraints, Tiresias uses adjacency matrices that maintain the provenance information. Let P be the $n \times m$ "adjacency" matrix over relations `A` and `B`, with sizes n and m respectively. $P_{ij} = 1$ iff b_j is in the provenance of a_i. Then the provenance constraints are written compactly as $a_i \geq P_{ij} b_j$. This is a standard format for modeling MIP constraints, supported by the MathProg modeling language [20].

The adjacency matrix grows quadratically to the data size, and it can become a bottleneck in the MIP processing. However, if the TiQL rule imposes an 1-1 relationship between head and body subgoal, the adjacency is a simple identity matrix (possibly permuted), and therefore unnecessary. The MIP optimizer can easily detect whether this is the case by checking FDs and key constraints. The optimizer also identifies (with the same tools) whether the relationship is many-to-one (due to joins in the body), in which case the adjacency matrix can be replaced by a simple vector.

6. EXPERIMENTAL EVALUATION

The first version of TIRESIAS has been implemented in Java, and interfaces with PostgreSQL and the GNU Linear Programming Kit (GLPK) [1] as the external MIP solver. It takes user input in TiQL and constructs the corresponding MIP problem files, which are then handled by the GLPK solver. The solver produces solution output files which are then parsed and linked to the CORE tables that reside in Postgres to produce the HDB instances. Our current version is running on single double-core machine; most of TIRESIAS's components are fully parallelizable, but that is part of future work. We evaluated our system on TPC-H generated data [2]. The TPC-H benchmark makes it easy to vary the datasize in a meaningful way, which was essential in the evaluation of all our parameters. We experimented with the 3 example queries presented in Sect. 3. We modified Q_1 and Q_2 to make them more challenging for performance, by leaving only `ok` as the group by attribute in the HDB `HOrderSum`: now partitioning is possible on `ok` instead of `(ok,sk')`, resulting in bigger problem sizes. Most of the presented graphs analyze the results of Q_1, and at the end of the section we compare its performance with Q_2 and Q_3. We valuate the performance of different parts of the system, by measuring the following runtimes: TiQL to MIP transformation measures the total time of parsing and processing the TiQL input up to the generation of all MIP problem files. The MIP solver time measures the time the

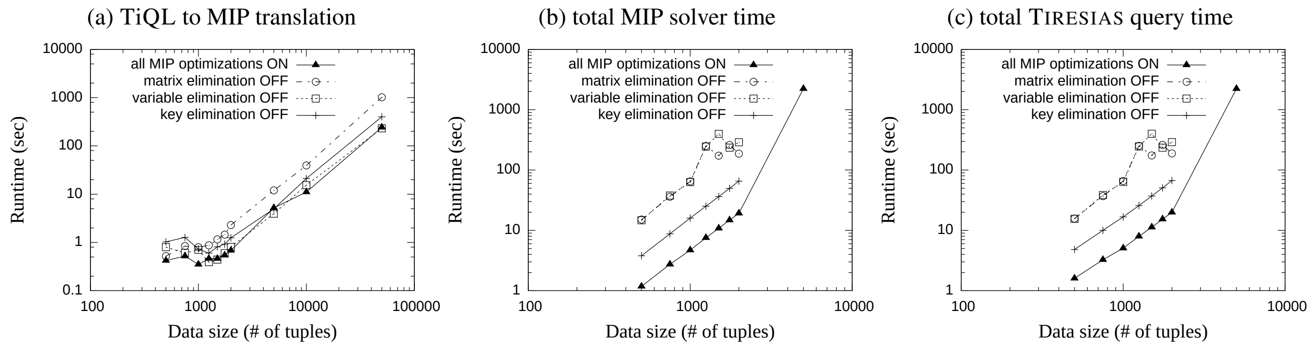

Figure 6: Evaluation of the options of the MIP optimizer over example query Q_1. MIP optimization does not add significant overhead to the TiQL to MIP translation time, but each optimization individually results in significant gains in the MIP solver execution times, and consequently the total TIRESIAS runtime. The MIP solver crashes on the 5k instance when any optimization is turned off, hence the absence of those points.

solver takes to solve all problem files generated by one query input, unless it is given as time per partition, in which case we measure the time the solver takes to process a single subproblem. The MIP construction time measures the time the MIP constructor takes to generate a single problem file, and finally, the total Tiresias query time is the execution time of a TiQL query from start to finish. This is equal to the sum of the TiQL to MIP transformation plus the total MIP solver time. All runtimes are measured in seconds.

6.1 MIP Optimizer

The MIP optimizer reduces the size of the produced problem files with optimizations that are to some extend already incorporated in modern solvers (e.g. elimination of variables and constraints). Our first set of experiments explores whether there is a benefit in performing these optimizations outside the MIP solver. For this part, we have disabled the pre-processing optimizer, so TIRESIAS always produces a single problem file. Fig. 6a and Fig. 6b present the execution times of the TiQL to MIP translation, and the MIP solver respectively, while Fig. 6c is their sum (total runtime). We measure the performance of query Q_1 when all MIP optimizations are activated in the MIP optimizer (all ON), for different sizes of the LineItem table, ranging from 500 to 5k tuples. We then deactivate each optimization in turn, and examine whether the MIP solver can achieve better performance. The answer is decisively no: removing any one of the 3 optimizations increases the execution time of the MIP solver by an order of magnitude in most cases, which is also reflected in the total runtime. The TiQL to MIP translation is also faster with the optimizations turned on.

There are two main reasons for these observations. First, the TiQL to MIP translation time is improved, because the optimizer produces smaller problem files that lead to noticeable gains in I/O cost. The second and most important reason is that TIRESIAS has a significant advantage: it can use DB specific information like functional dependencies to determine redundancies, whereas the MIP solver needs to analyze the whole program in order to achieve the same thing. Consequently, the MIP solver is much less efficient and often unsuccessful.

6.2 Partitioning

Our next experiments study the overall system performance and the pre-processing optimizer module. We ask the questions: how well does TIRESIAS scale in terms of runtime performance, and how does grouping partitions affect performance. We measure execution times for query Q_1 for various specifications of partition grouping size and dataset size. The results are presented in Fig. 7. Larger group sizes means that multiple partitions are grouped into a single subproblem. This results in larger MIP problem sizes, causing the MIP solver to dominate the total query runtime. Smaller groups result in smaller but more problem files, causing the system to become I/O bound. The graphs indicate that the choice of group size results in significant differences in runtime, and the ideal group size becomes larger, the larger the dataset. In its current implementation, TIRESIAS does not pick the group size automatically, but this will be an essential step in future work.

Figure 7 demonstrates significant improvements compared to the results of Fig. 6, where partitioning was deactivated. As seen in Fig. 7f, TIRESIAS can create and solve a constrained optimization problem over 1M tuples in about 2.5 hours. Figure 8 offers additional insight on the system scalability. The bars display the average time taken by the MIP constructor and the MIP solver to work on a single subproblem, when the group size is fixed to 150 partitions. We observe that the average time the solver spends on one problem remains fairly constant (around 1sec) for datasets of 5k to 1M tuples. This is intuitive as the average size of the generated subproblems is expected to be the same across the different datasets, as long as the underlying statistical properties of the data do not change. Of course the number of files generated will grow proportionally to the data size, so overall, the MIP solver runtime will scale linearly.

The per-problem runtime of the MIP constructor time on the other hand does not remain constant, but increases with the data size. This is because the MIP constructor retrieves all the data needed to generate a subproblem by issuing several database queries (8 in the case of Q_1), which become slower as the dataset grows larger. This still means though that the MIP construction time grows at a constant pace in relation to how regular database queries scale. This is an important observation, as it means that the performance of TIRESIAS is on par with DBMS performance. [4]

Finally, the line graphs of Fig. 8 refer to the logscale y-axis on the right, and represent the total TIRESIAS execution time for the same settings, along with the average elapsed time until the first results become available, which is about 3min for the largest dataset.

6.3 Query Comparisons

Our final set of experiments compares the runtimes of our 3 example queries Q_1, Q_2, and Q_3 across different data sizes, and for

[4] This hold in the case of problems that can be partitioned.

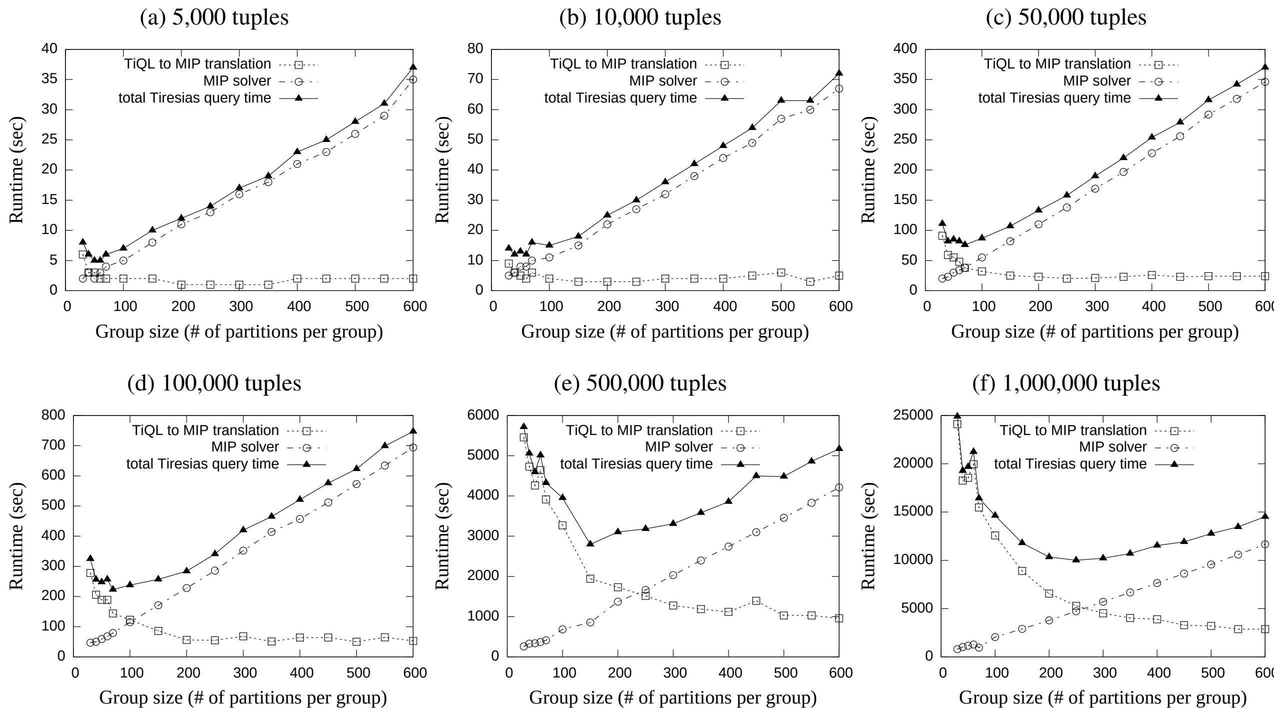

Figure 7: Evaluation of the partitioning optimization. TIRESIAS is shown to scale to optimization problems of very large sizes. The effect of group size in performance is significant, which pushes for a smart algorithm for its selection.

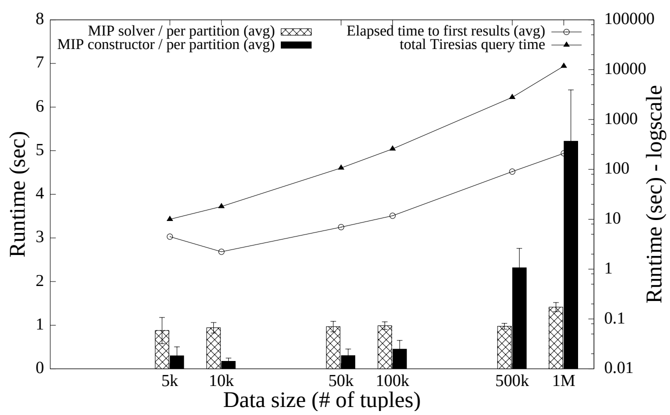

Figure 8: The bars demonstrate average execution time per partition for TIRESIAS and the GLPK solver, when we set group size equal to 150. These runtimes map to the left y-axis. The right y-axis is in log scale and demonstrates the total execution time of the how-to query, and the projected elapsed time to first results.

two settings of the group size parameter. The results are presented in Fig. 9. The results do not include any big surprises. Query Q_2 has the fastest TiQL to MIP runtime because it has fewer variables and constraints, so generating the problem files can be done faster than Q_1. Q_3 is the most complicated of the 3, as it includes a join with the `PartSupp` table, which makes the generated MIP problems much bigger. Note that the data sizes are indicated in terms of the number of tuples in the LineItem relation, but since Q_3 involves a second EDB, the number of actual tuples that are involved is much larger. The most important take-away from these observations is that even more complicated queries, with a lot of integer variables can be handled by the system.

7. RELATED WORK

Provenance work. The lineage of tuples in query answers was studied in [13], in the context of data warehouses. Provenance semirings were introduced as a formal foundation for lineage in [16], and extended to semi-modules for aggregation provenance in [4]; we use that formalism in our translation from TiQL to MIP. Lineage expressions are routinely used in probabilistic databases [3, 14, 5].

Incomplete Databases and Stable Models. The highly influential work on incomplete databases [17] introduced the notion of possible worlds and the concept of c-tables, which are essentially relations where tuples are annotated with lineage expressions. Recent work [6] introduced ISQL, an extension of SQL for writing *what-if* queries. Stable models are a principled way to define semantics for datalog with negation; when extended to disjunctive datalog (which extends datalog with negation) stable models are called answer sets [19]. Both incomplete databases and stable model semantics ask two kinds of questions: is a tuple present in *all* possible worlds, or is a tuple present in *some* possible worlds, called the possible answers, and the certain answers respectively. TiQL borrows the idea of possible worlds from incomplete databases and from stable models, but it's semantics is different, in that it chooses one particular possible world.

Other RDM problems. How-to queries are related to *Reverse Query Processing* [9] and *Reverse Data Management* (RDM) [21], which consider the "reverse" direction of data transformations given a desired output, find corresponding changes to the input. Other examples of tasks that fall under this paradigm are view updates [10], data generation [7], data exchange [15]. RDM problems are in gen-

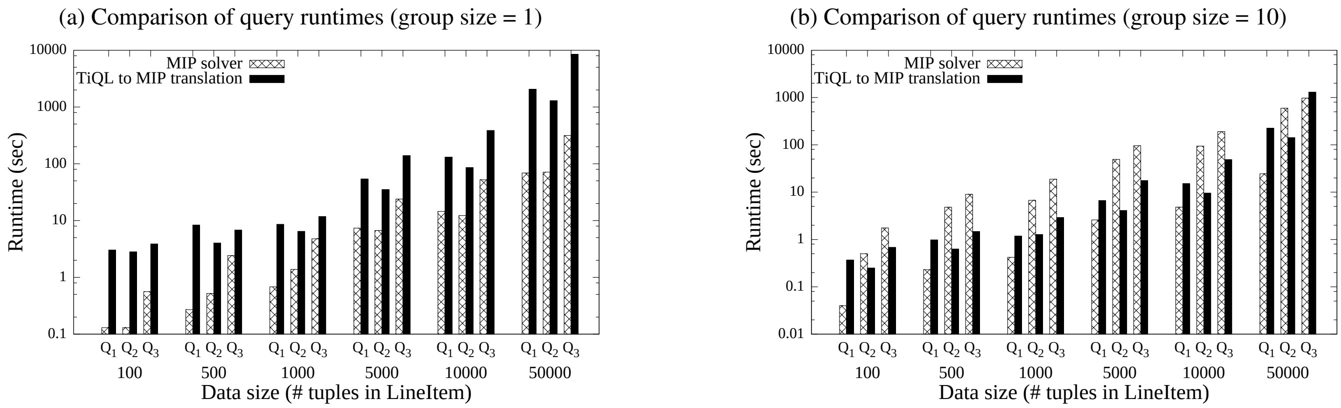

Figure 9: Performance comparison of the example queries over datasets of varying size. Queries Q_2 and Q_3 contain only integer variables, and are generally harder for the solvers to handle. Due to the partitioning optimization, TIRESIAS achieves great performance on all 3 queries. Without partitioning, MIP solvers need multiple hours to solve an instance of Q_3 on the smallest dataset.

eral harder than their "forward" counterparts, because the inverse of a function is not always a function, leading to significant challenges in their definition and implementation.

Synthetic data generation and reverse query processing. The goal of synthetic data generation is to generate a databases with some given statistics. It is important in database performance testing. One approach to synthetic data generation is *reverse query processing* [9], which essentially computes the inverse of each query operator. The approach is much more operational than TiQL, which treats the query as a whole. A declarative approach to data generation has been described recently [7]. That approach also translates the problem into a Linear Program and uses an LP solver to compute a solution. Synthetic data generation is different from *how-to* queries, because there is no input database: the input consists only of a set of numbers, the desired statistics.

8. CONCLUSIONS

We have described TIRESIAS, a system for computing *how-to* queries on relational databases. How-to queries are important in strategic enterprise planning today, because they allow users to explore what hypothetical change to make to the database in order to improve Key Performance Indicators for the enterprise. Queries are written in a declarative language, TiQL, which is based on datalog syntax. Query evaluation consists of translating the query into a Mixed Integer Program, then using a standard MIP solver. The translation is non-trivial, and is based on provenance semi-rings, and on semi-modules for aggregate provenance. A naive translation does not scale to large databases: we describe several optimizations that allowed TIRESIAS to scale to one million tuples. Future work includes parallelization, and an extension of the user interface with the hypothetical tables.

9. REFERENCES

[1] GNU Linear Programing Kit. http://www.gnu.org/s/glpk.

[2] TPC-H benchmark specification. http://www.tpc.org/tpch.

[3] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154, 2006.

[4] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011.

[5] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, pages 983–992, Washington, DC, USA, 2008. IEEE Computer Society.

[6] L. Antova, C. Koch, and D. Olteanu. From complete to incomplete information and back. In *SIGMOD Conference*, pages 713–724, 2007.

[7] A. Arasu, R. Kaushik, and J. Li. Data generation using declarative constraints. In *SIGMOD Conference*, pages 685–696, 2011.

[8] D. Barone, L. Jiang, D. Amyot, and J. Mylopoulos. Composite indicators for business intelligence. In *ER*, pages 448–458, 2011.

[9] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *ICDE*, pages 506–515, 2007.

[10] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *PODS*, pages 338–347, 2006.

[11] D. Chen, R. Batson, and Y. Dang. *Applied Integer Programming: Modeling and Solution*. John Wiley & Sons, 2011.

[12] S. Cohen, W. Nutt, and Y. Sagiv. Containment of aggregate queries. In *ICDT*, pages 111–125, 2003.

[13] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[14] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDBJ*, 16(4):523–544, 2007.

[15] R. Fagin and A. Nash. The structure of inverses in schema mappings. *J. ACM*, 57(6):31, 2010.

[16] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[17] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[18] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *SIGMOD Conference*, pages 687–700, 2008.

[19] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[20] A. Makhorin. Modeling language GNU MathProg. *Structure*, (January), 2010.

[21] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. In *VLDB*, 2011.

[22] J. D. Ullman. *Principles of Database and Knowledgebase Systems I*. Computer Science Press, Rockville, MD 20850, 1989.

[23] J. D. Ullman. *Principles of Database and Knowledgebase Systems II: The New Technologies*. Computer Science Press, Rockvill, MD 20850, 1989.