The Power of How-To Queries

Alexandra Meliou
University of Massachusetts, Amherst
ameli@cs.umass.edu

1 The Case for How-To Queries

Informally, a *data transformation* consists of a function from an input data source to a target output. The natural evolution of data follows the directionality of the transformations, i.e., from source to target. Database research mainly focuses on forward-moving data flows: Source data is subjected to transformations and evolves through queries, aggregations, and view definitions to form a new target instance, possibly with a different schema. This *forward paradigm* underpins most data management tasks today, such as querying, data integration, data mining, etc. In contrast, in *reverse data management* [11], one needs to act on the input data in order to achieve a desired effect in the output. Some data management tasks already fall under this paradigm. Examples include updating through views [7], data generation [5], causality computation [10], data cleaning [2]. All these problems share a common premise: They essentially reverse-engineer a transformation in order to achieve a desired target instance, or target properties.

In this talk, I will discuss a new database operator that falls under the reverse processing paradigm: *How-to queries* [11]. A how-to query computes hypothetical updates to the database that achieve a desired effect on one or several indicators, while satisfying some global constraints. For example, in the business domain, Key Performance Indicators (KPI) are measures of a company's performance according to some metric [4]. Consider a shipping company that fills orders by contracting with several suppliers. One KPI is the total number of orders per supplier: the smaller the indicator, the less 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 [9, 3] 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 inverse: 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, to answer the query *"How do I diversify my inventory without incurring additional costs?"* the system must propose major updates to the database of outstanding orders, and present it to the user.

EXAMPLE 1 (PORTFOLIO ANALYSIS). *An analyst at a brokerage company wants to investigate strategies to achieve better returns on customer portfolios, based on the company's recommendations during the last three years. He wants to receive a list of possible modifications to the company's stock recommendations that would achieve the desired output in the customer's portfolios (e.g., 10% return). Out of the possible scenarios, the analyst wants to give preference to those closest to the company's current strategy, as they would require fewer trades.*

Example 1 is a modification of an example from [3]. Note the fundamental difference between this example and the one geared towards hypothetical queries: In contrast to [3], where the analyst manually selects relevant hypotheses for testing, in the how-to setting, the relevant scenarios are automatically selected based on the specifications on the target data. Along with the target restrictions on the portfolio returns, *optimization criteria* can also restrict the solution space (e.g., minimize the number of trades). These criteria define a distance metric from the initial input, and select the solution closest to it.

EXAMPLE 2 (INVENTORY DIVERSIFICATION). *A manufacturing company orders parts from multiple suppliers around the world. To reduce its dependence on any single country, the company wants to limit the number of parts ordered from any particular country, to no more than 10% of the total orders. The company can revise the current inventory by reassigning some orders to different suppliers. Ideally, it would like to achieve this with the minimum number of changes.*

2 Evaluation of How-To Queries

Traditional database systems cannot model how-to queries. How-to queries are a special case of constrained optimization, and in particular, linear programming (LP), and integer programming (IP) [6]. Several mature LP/IP tools exists, and are used extensively. However, mapping a *how-to* query to a linear or integer program is non-trivial. 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 KPIs) 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 support such planning.

In this talk, I will discuss and demonstrate TIRESIAS [12], 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. A user can specify through TiQL possible actions, required constraints, and desired objectives. 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, into a *mixed integer program* (MIP), which has both real and integer variables.

The translation of TiQL into MIP is complex, and is an important technical contribution of this system. 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, and 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, TIRESIAS uses provenance semi-rings [8], and the recently introduced semi-modules for aggregation provenance [1]. The number of integer and real variables created by the translation is large, and needs to be managed inside the relational database system.

Even robust MIP solvers cannot scale to typical database sizes. Another key contribution of this work 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 TIRESIAS to scale up to millions of 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. I will also discuss other optimizations, such as variable and matrix elimination: While some of these are also done by the MIP solver, I 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.

TIRESIAS offers an example of the power reverse data management can yield for database systems, and identifies new research questions for exploration.

References

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

[2] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pp. 68–79, 1999.

[3] A. Balmin, T. Papadimitriou, and Y. Papakonstantinou. Hypothetical queries in an OLAP environment. In *VLDB*, pp. 220–231, 2000.

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

[5] N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB*, pp. 1097–1107, 2005.

[6] D. Chen, R. Batson, and Y. Dang. *Applied Integer Programming: Modeling and Solution*. J.W. & Sons, 2011.

[7] S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. In *PODS*, pp. 317–331, 1983.

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

[9] L. V. S. Lakshmanan, A. Russakovsky, and V. Sashikanth. What-if olap queries with changing dimensions. In *ICDE*, pp. 1334–1336, 2008.

[10] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 4(1):34–45, 2010.

[11] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *PVLDB*, 4(12):1490–1493, 2011.

[12] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *SIGMOD*, pp. 337–348, 2012.