Extending Datalog with Analytics in LogicBlox

Molham Aref[1], Benny Kimelfeld[⋆,1,2], Emir Pasalic[1], and Nikolaos Vasiloglou[1]

[1] LogicBlox, Inc.
[2] Technion, Israel

1 Introduction

LogicBlox is a database product designed for enterprise software development, combining transactions and analytics. The underying data model is a relational database, and the query language, LogiQL, is an extension of Datalog [13]. As such, LogiQL features a simple and unified syntax for traditional relational manipulation as well as deeper analytics. Moreover, its declarative nature allows for substantial static analysis for optimizing evaluation schemes, parallelization, and incremental maintenance, and it allows for sophisticated transactional management [11]. In this paper, we describe various extensions of Datalog for supporting prescriptive and predictive analytics. These extensions come in the form of mathematical optimization (mixed integer programming), machine-learning capabilities, statistical relational models, and probabilistic programming. Some of these extensions are currently implemented in LogicQL, while others are in either development or planning phases.

2 LogiQL Basics

In this section we briefly (and informally) describe LogiQL. (See [13] for a full description of the language.) The core components of LogiQL are its rules and constraints, which are stated over the *predicates* of the database schema. A *(basic) derivation rule* is a standard Datalog rule that defines how new facts are derived in the database. For example, the following rules define Anc as the transitive closure of the predicate Par.

$$\mathrm{Anc}(x, y) \leftarrow \mathrm{Par}(x, y)$$
$$\mathrm{Anc}(x, y) \leftarrow \mathrm{Anc}(x, z), \mathrm{Par}(z, y)$$

An *integrity constraint* does not derive new facts, but rather fires an error when violated. For example, the rule $\mathrm{Anc}(x, y) \rightarrow x \neq y$ states that ancestorship is antireflexive, and the rule

$$\mathrm{Par}(x, y), \mathrm{Par}(x, z) \rightarrow y = z$$

states that every object has at most one parent, that is, in Par the first element is a key. For interpretability sake, brackets are used to implicitly express key constraints, as in $\mathrm{Par}[x] = y$. Derivation rules and constraints syntactically differ in the direction of the implication arrow. The logical language for specifying the right-hand-side of

[⋆] Taub Fellow – supported by the Taub Foundation rules, as well as both sides of constraints, allows arbitrary propositional formulas with numerical operations and comparisons (while safety conditions, such as stratification, may be imposed to bound complexity).

Predicate-to-Predicate (*P2P*) rules are used for deriving whole relations over tuples. An example of such a rule is the *aggregate* P2P rule, such as the following one that sums up the salaries for each employee.

$$\mathrm{Annual}[e] = a \leftarrow \mathsf{agg}{\ll} a = \mathsf{sum}(s) {\gg}\ \mathrm{Salary}(e, 2014, s)$$

Such a rule always includes a component of the form func≪arg≫, where func is the type of the predicate rule, and arg is an additional argument that gives specific arguments for the rule. In the above rule, e is a *grouping variable* (as it occurs in the head). This rule derives a fact $\mathrm{Annual}(e, a)$ for every e in the first attribute of Salary.

3 Extensions for Analytics

We now describe several extensions of the LogicBlox system, for supporting prescriptive and predictive analytics.

3.1 Mixed Integer Programming (MIP)

MIP is often applied for effectively solving real-world optimization problems that naturally arise in prescriptive analytics, such as network flow optimization, resource allocation, and scheduling. Solutions for various classes of mathematical programming problems can be obtained by deploying specialized, highly optimized solvers (e.g., [12, 1]). As an example, a warehouse supplies stores s with products i from its available inventory a_i. For each store s and product i there is a demand d_i^s and available inventory a_i^s. For each product i, let w_i denote the quantity of i in the warehouse, and p_i denote its unit price. A set of N trucks delivers commodity, and for simplicity assume that each truck makes a single warehouse-to-store trip. We need to determine the quantity q_i^s to ship to each store in order to maximize revenue. In standard MIP notation, the problem can be phrased as follows. Here, ψ_s is a binary variable (with values in $\{0, 1\}$) determining whether a truck should be sent to store s, and m_i^s is the quantity of product i missing in order to satisfy the demand at store s.

$$\text{Maximize} \quad (\sum_{i,s} d_i^s - m_i^s) * p_i \ \text{ subject to:}$$

$$\forall i, s \quad w_i^s + q_i^s + m_i^s \geq d_i^s\,,\ \ q_i^s \leq B * \psi_s\,,\ \sum_{s'} q_i^{s'} \leq a_i\,,\ \sum_{s'} \psi_{s'} \leq N$$

$$\forall i, s \quad m_i^s \geq 0\,,\, q_i^s \geq 0\,,\, \psi_s \in \{0, 1\}$$

In this program, we assume that a store can hold at most 1000 units of each product. Observe that the unknown variables are the q_i^s, m_i^s and ψ_s (while the others are fixed).

MIP declaration in LogiQL is done by means of predicates with *free second-order variables*, which are essentially unknown functions over predefined domains, except that they are eventually assigned actual values (by invoking a MIP solver). Moreover, the objective function (that one wishes to minimize or maximize) is simply an attribute of a relation. Linear constraints are phrased as LogiQL constraints. Hence, a LogiQL program with MIP is an ordinary program, with the addition that attributes are marked as second-order variables or objectives. As an example, the following program corresponds to the above MIP specification. Here, the second-order variables are $\mathrm{m}[i,s]$, $\mathrm{q}[i,s]$ and $\mathrm{psi}[s]$, and the objective is $\mathrm{Obj}(v)$.

$$
\begin{aligned}
\mathrm{Obj}(v) &\leftarrow \mathsf{agg}\ll v = \mathsf{sum}(z)\gg\ z = (\mathrm{d}[i,s] - \mathrm{m}[i,s]) * \mathrm{p}[i] \\
\mathrm{Store}(s), \mathrm{Prod}(i) &\rightarrow \mathrm{w}[i,s] + \mathrm{q}[i,s] + \mathrm{m}[i,s] \geq \mathrm{d}[i,s] \\
\mathrm{TotalTrans}[i] = v &\leftarrow \mathsf{agg}\ll v = \mathsf{sum}(z)\gg\ \mathrm{q}[i,s] = z \\
\mathrm{TotalTrans}[i] = v &\rightarrow v \leq \mathrm{a}[i] \\
\mathrm{q}[i,s] = v_1, \mathrm{psi}[s] = v_2 &\rightarrow v_1 \leq v_2 * 1000 \\
\mathrm{TotalTrucks}(v) &\leftarrow \mathsf{agg}\ll v = \mathsf{sum}(z)\gg\ \mathrm{psi}[s] = z \\
\mathrm{TotalTrucks}(v) &\rightarrow v \leq \mathrm{AvailableTrucks}[] \\
\mathrm{m}[i,s] = v &\rightarrow v \geq 0\,, (\mathrm{psi}[s] = 0\ \mathsf{or}\ \mathrm{psi}[s] = 1)
\end{aligned}
$$

Observe that the constraints are used in a fashion similar to the *stable-model* (or *answer-set*) semantics [8], except that we also optimize an objective. Other similar approaches include [5, 10, 16, 14]. As far as we know, LogicBlox is the first commercial database system to provide native support for prescriptive analytics.

The engine automatically synthesizes the necessary mathematical programming instances from the program, and invokes a MIP solver (e.g., [12, 1]). Specifically, we ground (i.e., eliminates quantifiers in) the problem instance in a manner similar to [15], and translate the constraints over variable predicates into a representation that can be consumed by the solver. Then, the solver output is used for populating the value of marked predicates (turning unknown values into known ones). The evaluation engine listens to updates in the relevant relations (as part of standard view maintenance), and invokes the solver when necessary to populate unknown values. The underlying MIP instance is constructed incrementally. For example, if a store demand changes, then only the portion of the program that is relevant to that store is replaced in the current MIP instance (before being re-sent to the solver to obtain an updated solution). We found that this optimization often leads to considerable reduction in execution cost.

3.2 Machine Learning (ML) Aggregates

The next extension is predictive analytics by means of a built-in set of ML algorithms. We use the special predict P2P rule that comes in two modes: *learning* mode (where a model is being learned) and *evaluation* mode (where a model is being applied to make predictions). We do not give here the formal syntax and semantics for these rules; rather, we give an illustrative example.

Suppose that we wish to predict the monthly sales of products in branches. We have the following predicates:

- $\mathrm{Hstr}[\mathrm{sku}, \mathrm{branch}, \mathrm{month}]{=}\mathrm{amount}$ contains historical sales per sku ("stock keeping unit") and branch;
- $\mathrm{Ftr}[\mathrm{sku}, \mathrm{branch}, \mathrm{name}]{=}\mathrm{value}$ associates with every sku, branch and feature name a corresponding feature value.

The following learning rule learns a logistic-regression model for each sku and branch, and stores the resulting *model object* (which is handle to a representation of the model) in the predicate $\mathrm{SM}[\mathrm{sales}, \mathrm{branch}] = \mathrm{model}$.

$$\mathrm{SM}[s, b] = m \leftarrow \mathsf{predict}{\ll} m = \mathsf{logist}(v|f){\gg}\ \mathrm{Hstr}[s, b, t] = v, \mathrm{Ftr}[s, b, t, n] = f$$

And the following evaluation rule evaluates the model to get specific predictions.

$$\begin{gathered}\mathrm{Sales}[s, b, t] = v \leftarrow \mathsf{predict}{\ll} v = \mathsf{eval}(m|f){\gg}\\ \mathrm{Unknown}(s, b, t), \mathrm{SM}[s, b] = m, \mathrm{Ftr}[s, b, t, n] = f\end{gathered}$$

3.3 Statistical Relational Models

Such models are specified by various mechanisms, including Markov Logic Networks (MLN) [6] and Probabilistic Soft Logic (PSL) [4]. MLNs and PLSs are, intuitively, logical formalisms that allow for *soft rules*. The canonical example is

$$R(x) \leftarrow R(y)\ ,\ \mathrm{Friends}(x, y)$$

where R describes a person property (e.g., "smokes" or "votes"). This rule states that whenever x and y are friends, the property R propagates from y to x; this rule should be taken as a hint on the unknown, and not as rigid truth. While an ordinary Datalog program specifies a unique extension of the database, soft rules specify a probability space over such extensions. Intuitively, the probability of a possible extension is determined by the extent to which the rules are satisfied (where weights of rules are taken into account). A common practice is to find the *most likely* extension (i.e. Maximum A-Priori, or MAP, inference), such as the most likely votes given partial knowledge about votes, and use that world as an ordinary database. We make an ongoing effort to support MLN and PSL within LogicBlox. Our current implementation applies MAP inference by translation into MIP. In future work we plan to include specialized algorithms to reduce the execution cost.

3.4 Probabilistic Programming Datalog (PPDL)

Formalisms for specifying general statistical models, such as probabilistic-programming languages [9], typically consist of two components: a specification of a stochastic process (the prior), and a specification of observations that restrict the probability space to a conditional subspace (the posterior). We plan to enhance LogiQL with capabilities of probabilistic programming, in order to facilitate the design and engineering of ML solutions. Towards that, we have initiated a theoretical exploration of such an extension. In a recent paper [3], we have proposed *Probabilistic Programming Datalog* (*PPDL*), which is a framework that extends LogiQL with convenient mechanisms to include common numerical probability functions; in particular, conclusions of rules may contain values drawn from such functions. As a (simplistic) example, assume the relation $\mathrm{Client}(\mathrm{ssn}, \mathrm{branch}, \#\mathrm{visits})$ that associates clients with social security numbers, local branches, and an average number of visits (per month) in the branch. The rule

$$\mathrm{Visits}(c, b, \mathrm{Poisson}[\lambda]) \leftarrow \mathrm{Client}(c, b, \lambda)$$

associates with the client a random number of visits in the branch, where that number is drawn from the Poisson distribution with average (parameter) $\#\mathrm{visits}$.

The semantics of a program is a probability distribution over the possible outcomes of the input database with respect to the program; these possible outcomes are minimal solutions with respect to a related program that involves existentially quantified variables in conclusions. Observations are naturally incorporated by means of constraints. We focused on discrete numerical distributions (such as Poisson), but even then the space of possible outcomes may be uncountable (as a solution can be infinite). We defined a probability measure over possible outcomes by applying the known concept of *cylinder sets* [2] to a probabilistic chase procedure. This chase is similar to that of data exchange with *tuple-generating dependencies* [7], except that instead of introducing named nulls, we sample real values from the associated distribution (e.g., $\mathsf{Poisson}[5]$). We have shown that the resulting semantics is invariant under different chases.

4 Conclusions

We described four approaches taken by LogicBlox to extend LogiQL with built-in analytics. While MIP and ML aggregates are conceptually syntactic bridges between Datalog and external solvers, statistical relational models, and PPDL feature stronger ties to Datalog (and naturally require more in-house implementation effort). In future work we plan to investigate the applicability of our statistical specifications to real-life problems that arise in our business, as well as their theoretical and system aspects.

References

1. T. Achterberg. Scip: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1), 2009. http://mpc.zib.de/index.php/MPC/article/view/4.
2. R. B. Ash and C. Doleans-Dade. *Probability & Measure Theory*. Harcourt Academic Press, 2000.
3. V. Barany, B. t. Cate, B. Kimelfeld, D. Olteanu, and Z. Vagena. Declarative statistical modeling with datalog. *arXiv preprint arXiv:1412.2221*, 2014.
4. M. Bröcheler, L. Mihalkova, and L. Getoor. Probabilistic similarity logic. In *UAI*, 2010.
5. M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. Np-spec: an executable specification language for solving all problems in np. *Computer Languages*, 26(2):165–195, 2000.
6. P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on AI and Machine Learning. Morgan & Claypool Publishers, 2009.
7. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, volume 2572 of *LNCS*. Springer, 2003.

8. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.
9. N. D. Goodman. The principles and practice of probabilistic programming. In *POPL*, 2013.
10. S. Greco, C. Molinaro, I. Trubitsyna, and E. Zumpano. NP datalog: A logic language for expressing search and optimization problems. *TPLP*, 10(2):125–166, 2010.
11. T. J. Green, M. Aref, and G. Karvounarakis. LogicBlox, platform and language: A tutorial. In *Int Conf on Datalog in Academia and Industry*, 2012.
12. I. Gurobi Optimization. Gurobi optimizer reference manual, 2015.
13. T. Halpin and S. Rugaber. *LogiQL: A Query Language for Smart Databases*. CRC Press, 2014.
14. A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.
15. F. Niu, C. Ré, A. Doan, and J. W. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *PVLDB*, 4(6):373–384, 2011.
16. T. E. Sheard. Painless programming combining reduction and search: Design principles for embedding decision procedures in high-level languages. In *ICFP*, pages 89–102, 2012.