Design and Implementation of the LogicBlox System

Molham Aref Balder ten Cate Todd J. Green Benny Kimelfeld Dan Olteanu
Emir Pasalic Todd L. Veldhuizen Geoffrey Washburn
LogicBlox, Inc. *
firstname.lastname@logicblox.com

ABSTRACT

The LogicBlox system aims to reduce the complexity of software development for modern applications which enhance and automate decision-making and enable their users to evolve their capabilities via a "self-service" model. Our perspective in this area is informed by over twenty years of experience building dozens of mission-critical enterprise applications that are in use by hundreds of large enterprises across industries such as retail, telecommunications, banking, and government. We designed and built LogicBlox to be the system we wished we had when developing those applications.

In this paper, we discuss the design considerations behind the LogicBlox system and give an overview of its implementation, highlighting innovative aspects. These include: LogiQL, a unified and declarative language based on Datalog; the use of purely functional data structures; novel join processing strategies; advanced incremental maintenance and live programming facilities; a novel concurrency control scheme; and built-in support for prescriptive and predictive analytics.

Categories and Subject Descriptors

H.2 [**Database Management**]

General Terms

Algorithms, Design, Languages

Keywords

LogicBlox; LogiQL; Datalog; Leapfrog Triejoin; Incremental Maintenance; Transaction Repair; Live Programming; Predictive Analytics

*We wish to acknowledge the many people at LogicBlox who contributed to the development of the LogicBlox 4.X platform and its various components described in this paper. These include Thiago Bartolomei, Martin Bravenboer, Eelco Dolstra, Mihai Gavrilescu, Shan Shan Huang, Wes Hunter, Eugene Kamarchik, Grigoris Karvounarakis, Yannis Kassios, George Kollias, Patrick Lee, Ruy Ley Wild, Rafael Lotufo, Laurent Oget, Søren Olesen, Trevor Paddock, Wael Sinno, Kurt Stirewalt, Zografoula Vagena, Nikolaos Vasiloglou, Rob Vermaas, Daniel Zinn. We also wish to acknowledge the support of our enterprise clients, our partners, our academic collaborators, and DARPA.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2742796.

1. INTRODUCTION

Recently there has been a trend towards specializing database system architectures to particular use cases, such as transactions, analytics, graphs, arrays, documents, or streams [39]. The driving argument behind this trend is performance: specialized transactional or analytical systems, for example, have demonstrated 10-100X speedups over the traditional "OldSQL" database architectures first developed over three decades ago.

At the same time, we are witnessing a trend towards sophisticated applications for enhancing and automating decision making, where application requirements combine aspects of several of these use cases. Moreover, these applications typically have to *evolve* frequently, making *developer productivity* a key concern. In our experience, developing and operating such applications using a "hairball" of specialized databases and programming languages is prohibitively expensive, and end-user maintenance of such applications is impossible.

This observation motivates us to re-examine the argument for specialization in database systems. Are the performance differences observed to date between specialized and general-purpose architectures due to reasons that are fundamentally significant? Or do traditional database architectures simply not embody the best approach to building a general-purpose system?

We believe the latter is the case. To validate our hypothesis, we have built a database system that demonstrates that it is possible to meet the requirements for such applications without unacceptable compromises in performance or scalability. In our experience, the operational and developmental efficiencies gained by using such a system provide a significant reduction in cost and time and overall improvement in the utility, usability, and continued relevance of the application.

1.1 Design Principles

In order to support sophisticated analytical applications, we have expanded our notion of "database system" to include features and capabilities commonly found in programming languages, spreadsheets, statistical systems, and mathematical optimization systems. We also have taken a fresh look at a number of the basic architectural building blocks of a general-purpose database system, challenging some deeply-held beliefs and sacred cows.

A useful analogy to what we are aiming to achieve is that of the smartphone: less than a decade ago, the iPhone redefined what it means to be a phone and subsumed a variety of specialized consumer devices such as cameras, music players, gaming devices, and GPS devices. A smartphone is not the highest resolution camera, the highest fidelity music player, the most accurate GPS device, the most entertaining gaming device, or even the best phone. However, by providing so many capabilities in one device with a unified

user experience, the smartphone has made the specialized devices irrelevant for most use cases. Moreover, smartphones have made it possible to build new kinds of applications integrating several of these capabilities in useful and interesting ways.

LogicBlox is a "smart" database system in the same sense that the iPhone was the prototypical "smart" phone. Our aim is not to beat the performance of, say, specialized column stores or in-memory transactional systems; such systems make extreme trade-offs that render them useless outside of their comfort zones. Rather, we aim to come close enough (say, within a factor of 5) to the performance of specialized systems, such that we can satisfy applications needs without sacrificing the flexibility required to cope with mixed workloads and the desire to support application evolution via self-service.

In order to achieve this, we follow a philosophy of "brains before brawn" where we prefer to achieve high performance with better algorithms and data structures before resorting to the use of brute force techniques based on throwing money and hardware at the problem (e.g., by partitioning an application across thousands of cores or loading all application data in memory). We also follow a philosophy of aggressive simplification and consolidation: we aim for a relatively small set of language and system features that compose elegantly in ways that cover a broad range of use cases. This is in contrast to competing systems,which from our point of view do not address the complexity hairball, but simply move it inside the database (e.g., by including separate column-store and row-store sub-engines or requiring the use of several disparate query and scripting languages for application development).

The interplay between a number of key themes characterizes our system architecture:

T1: Simple, declarative, and uniform language. A major goal of our system is to unify the programming model for applications that automate and enhance decision making by using a single, expressive, *declarative* language that can be used by domain experts to understand and evolve the application. To achieve this goal, we have developed *LogiQL*, an extended form of Datalog [29, 2, 19] that is expressive enough to allow coding of entire applications (including queries and views; stored procedures; reactive rules and triggers; and statistical and mathematical modeling).

LogiQL is based on Datalog much in the same way that functional languages are based on the lambda calculus. Datalog is highly declarative, and we believe it strikes the right balance between usability, expressive power, safety, and performance. The resurgence of Datalog in academic circles has been well documented [34, 1, 23]. Other recent industrial systems based on Datalog include Google's Yedalog [13], Datomic[1], and the nascent EVE project[2], funded by Andreessen Horowitz.

As a relatively simple language with a transparent semantics, LogiQL also lends itself well to presentation via alternative "surface syntaxes" which can be more appropriate for a given user community and task at hand. Like SQL and the formula languages found in spreadsheets, LogiQL is first-order and first normal form, making it easier to understand by less technical end-users, who we have previously observed to struggle with higher-order concepts such as passing functions to functions or building list-based data structures.

LogiQL is both expressive and "safe," with natural syntactic fragments capturing important complexity classes such as PTIME. The ability to control the expressive power of the language has useful practical benefits. On the one hand, we can effectively control how much power to put in the hands of end users who would otherwise be able to write non-terminating or otherwise harmful programs, which are especially unpleasant in a shared server environment. On the other hand, the ability to "dial up" the expressive power of the language (all the way to Turing-complete, if need be) makes us confident that we ultimately will not need to have support for imperative and other programming paradigms to handle unanticipated use cases.

The semantics of a LogiQL program is largely independent of the order in which elements of the program appear, and is not deeply tied to a particular physical evaluation strategy. This is a rare property among programming languages: even LogiQL's closest cousin, Prolog, relies on a fixed, top-to-bottom and left-to-write reading of the program. This "disorderliness" of the language (to borrow an apt phrase from the Bloom project [4]) allows great flexibility in optimization and evaluation, and makes LogiQL amenable to high-performance implementations based on memory-hierarchy friendly, set-at-a-time, bottom-up methods, as well as automatic parallelization of queries and views. Moreover, we are able to draw on a rich literature for automatic optimizations and incremental evaluation strategies.

T2: Clear and easy to understand semantics. The semantics of LogiQL is based on ordinary two-valued logic and sets, with fully serializable transactions. Historically, this kind of commitment to simplicity and correctness had to come at the expense of application performance and scalability. We will demonstrate below how we avoid such compromises.

LogiQL encourages a high level of schema normalization, namely sixth normal form (6NF) [11][3]. By encouraging the use of relations with at most one non-key variable, we derive a number of benefits:

1. We avoid the use of nulls, eliminating many hard-to-spot mistakes. The treatment of null values in databases, and the three-valued logic that it carries with it, has long been a source of conceptual problems and awkwardness [27].
2. We improve semantic stability by making the addition or removal of schema information easier as the application evolves. The more changes a user is forced to make to a model or query to cope with an application change, the less stable it is. Adding and removing new entities and relationships to the LogiQL application requires far less surgery and database downtime than is the case when using tables in lower normal forms (i.e., the wide tables used in SQL row stores).
3. We improve the performance of queries that involve a smaller number of attributes than would normally exist in a wide table. The low information entropy of normalized tables allows compression schemes and efficiency approaching that of column stores.
4. We make it easier to support temporal features like transaction time and valid time for each piece of information in the database.
5. We make it easier to support alternative syntaxes. For example, there is a direct mapping between LogiQL and conceptual modeling languages like ORM [22]. It is also straightforward to support function- and array-based language syntax since this highly normalized schema does not require naming of relation variables (i.e., table columns).

Conventional wisdom favors wide relations, motivated by a compulsion to avoid joins whenever possible. Naturally, embracing

[1] www.datomic.com

[2] www.incidentalcomplexity.com

[3] For example, to store (supplier-name, status, city) data in 6NF, one uses separate supplier-name, status, and city tables, which map identifiers to attribute values.

6NF introduces the challenge that queries often involve a much larger number of relations. Selection conditions in queries typically apply to multiple relations, and simultaneously considering all the conditions that narrow down the result becomes important. These concerns require new kinds of join algorithms. *Leapfrog trie-join* [40], the workhorse join algorithm in our system, lets us support a clean semantics without unacceptable compromises in performance or scalability.

Similarly, traditional SQL-based relational database systems adhere to a bag (multiset) semantics, in order to avoid the cost of frequent sorting and deduplication. This design choice drastically reduces the opportunity for query optimization, since it often happens that queries that are logically equivalent under set semantics are not under the bag-semantics. In our experience, it also makes queries harder for the end users to write and to reason about. LogiQL, in constrast, adheres to the set semantics, which enables us to implement a wider array of query rewriting and optimization techniques.

T3: Incrementality. LogicBlox supports efficient incremental materialized view maintenance. Our incremental maintenance algorithm is inspired by work done in the programming languages community. It improves significantly on the classical count and DRed algorithms [20] by guaranteeing that the work done is proportional to the trace edit distance between the before and after computations.

We have observed that although commercial database systems typically offer some degree of support for materialized views, the quality of this feature is usually rather poor: views are restricted to some subset of SQL queries and often perform very poorly in mixed OLTP and OLAP use cases (such that OLAP views must be refreshed manually at periodic offline intervals due to the inefficiency of existing view maintenance algorithms in the context of OLTP workloads). Difficulties in providing high-quality materialized view implementations have motivated some systems, such as HANA [16], to dispense with them altogether, and instead require that analytical queries always be computed on the fly. Besides sometimes being obviously wasteful in terms of computational resources, this approach also strikes us as problematic for truly computation-intensive applications. No matter how fast query evaluation can be made via parallelization, specialized storage schemes, and so forth, data sets and queries will be encountered that simply cannot be processed in this manner at interactive speed.

We will describe below how we used efficient view maintenance (along with purely functional data structures and efficient joins) as a key building block for our concurrency control scheme, *transaction repair*.

Finally, LogicBlox is designed to support efficient incremental changes to LogiQL programs. This capability makes it possible for us to support "live programming" use cases that allow end users to evolve their applications using a spreadsheet metaphor. This problem is quite different from traditional view maintenance, and requires a new set of techniques as outlined below.

T4: Immutability. The LogicBlox runtime makes pervasive use of *purely functional data structures* [35] (a form of *immutable* data structures) at all levels of the architecture. These data structures, originally motivated by purely functional programming languages, have a number of significant virtues in the context of database systems engineering:

1. Sharing of immutable substructures means that multiple versions of a relation or database can be represented together compactly, and changes between versions can be enumerated efficiently.
2. There is no need to maintain either read or write database locks. Each transaction starts by branching a version of the database in $O(1)$ time (a few nanoseconds—we have measured 80,000 branches per core per second). Changes in one branch (one transaction) have no visibility outside the transaction and are therefore perfectly isolated. Read-only transactions are perfectly scalable and do not need to be concerned about other changes occurring in other branches (i.e., throughput = # cores $\times$ single core transactions per second). Read-write transactions keep track of what they have read and written, and can be incrementally repaired with our efficient view maintenance machinery at commit time to achieve full serializability.
3. With purely functional data structures, a pointer or object-identifier uniquely identifies the state of an object. In the distributed setting, this avoids the need for cache coherence protocols.
4. We get useful temporal features such as time-travel essentially for free. We can branch any past version of the database, and the version graph can be an arbitrary directed acyclic graph.
5. There is no need for a transaction log for rollback and recovery purposes. Aborting a transaction means simply dropping all references to it; committing a transaction is, conceptually at least, just a pointer swap to the new head version of the database.

1.2 Outline

In the rest of the paper, we focus in more detail on several novel aspects of the LogicBlox system. Further aspects, including parallel and distributed computation, data compression, write-optimized data structures to support large transactional loads, and sampling-based query optimization are not discussed in this paper due to lack of space.

Section 2 presents the LogiQL language, and illustrates, by means of an example application, the way it is used. In Section 3, we highlight several innovative aspects of the LogicBlox system implementation. Section 3.1 discusses the use of persistent data structures and branching, which pervade the architecture. Section 3.2 overviews our workhorse join algorithm, leapfrog triejoin, and its associated view maintenance algorithm. Section 3.3 describes the meta-engine, which is the core architectural component enabling live programming. Section 3.4 presents our approach to concurrency control via transaction repair. We finally conclude in Section 4.

2. APPLICATION DEVELOPMENT IN LOGIQL

We illustrate the use of the LogicBlox system by means of a small example that highlights the need to support a variety of use cases in one application. After that, we give an overview of the LogiQL language itself.

2.1 Example Application

A user community made up of several hundred merchants, planners, supply chain personnel, and store managers at a large retailer wants to analyze historical sales and promotions data in order to assess the effectiveness of their product assortments, plan future promotions, predict future sales, and optimize the fulfillment of the demand generated by those assortments and promotions. The data in this scenario are several terabytes in size, and the model of the business is made up of a few thousand metrics.

There are multiple users concurrently using the application. Some are analyzing historical sales data via pivot tables, some are editing the data to specify different future promotional strategies

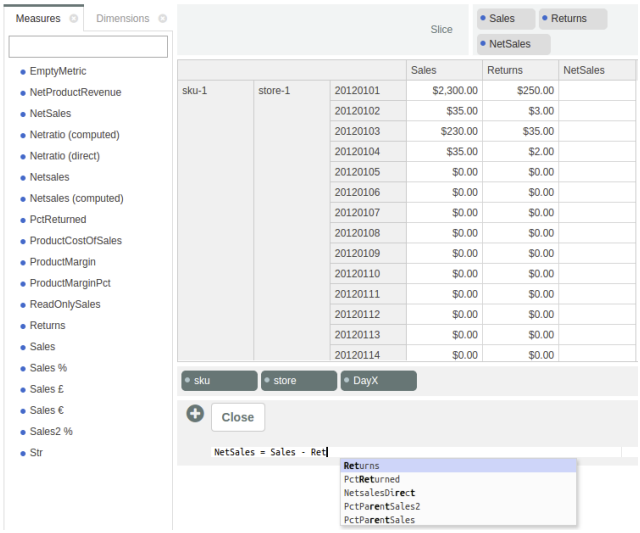


Figure 1: Screenshots of a retail planning application.

and generating new predictions of the demand created by those promotions, some are editing and overriding previously generated sales projections based on new information that is not available to the system yet, and some are asking the system for a recommended plan for fulfilling the demand generated by the promotions. All reads and writes occur at various levels of resolution, e.g., SKU/-Store/Day or Dept/Region/Month. These levels are not known a priori by the application developers.

Furthermore, as the competitive landscape and the macroeconomic picture changes, some of the privileged users of the system (i.e., power users or approved managers) would like to evolve the application to refine the financial, statistical, or mathematical models used within the application in order to enhance the groups' effectiveness and to increase the profits of the enterprise.

Figure 1 shows the environment in which users can visualize their data and their model, both of which can be evolved in order to reflect new knowledge about their business.

Through the notion of *workbooks*, we enable users to create branches of (subsets of) the database that can be modified independently. These workbooks allow us to support a variety of long-running transaction use cases. For example, workbooks can be created to allow a business person to analyze a variety of scenarios that model certain decisions that can be made to shape or fulfill client demand. Workbooks can also be created to support long running predictive and prescriptive analytics that will take several hours of machine time to run. Multiple users and processes may be working concurrently on different workbooks. It is important that the users of the system not be impacted negatively as hundreds of these long running transactions are taking place along with millions of smaller ones.

2.2 The LogiQL Language

In this section, we give an overview of the LogiQL language. Our overview is high-level and omits many details. See [21] for more details.[4]

[4]Visit `https://developer.logicblox.com` for a LogiQL tutorial and an online REPL for interactive tryout programming.

2.2.1 Basic Syntactic Elements

In this section we describe the core elements of a LogiQL program, namely *predicates*, *derivation rules*, *integrity constraints*, and *reactive rules*.

Predicates. LogicBlox supports predicates (relations) of the form $R(x_1,...,x_n)$ or $R[x_1,...,x_{n-1}] = x_n$ where each attribute x_i has either a primitive type (e.g., int, float, decimal, string, or date) or a user-defined *entity type*. As noted in T2 in Section 1, this type of schema encourages a strong form of normalization, corresponding to 6NF.

Each predicate may be declared as being either a *base predicate* (a.k.a. *extensional* or *EDB* predicate or relation), or a *derived predicate* (a.k.a. *intensional* or *IDB* predicate). If left unspecified, the predicate kind (base or derived), as well as the attribute types, are inferred from usage within the program. Base predicates contain input data and derived predicates are *views* over the base data. Derived predicates default to being materialized. However, if the derivion rule does not use aggregation or recursion, they can be left unmaterialized.

Derivation Rules. Derivation rules are used to specify view definitions. LogiQL admits two types of derivation rules.

A *basic derivation rule* is an expression of one of the two forms

$$R(t_1,\ldots,t_n) \leftarrow F$$
$$R[t_1,\ldots,t_{n-1}] = t_n \leftarrow F$$

where R is a derived predicate, $t_1,\ldots,t_n$ are variables and/or constants, and F is a formula containing all variables occurring in $t_1,\ldots,t_n$. The *formula*, here, may be a conjunction of atoms and/or negated atoms. The atoms involve base predicates, derived predicates, and/or built-in predicates such as equality, arithmetic addition, etc. An example of a basic derivation rule is

$$\text{profit}[sku] = z \leftarrow \text{sellingPrice}[sku] = x, \text{buyingPrice}[sku] = y,\ z = x - y.$$

which, using abbreviated syntax, may also be written as the expression

$$\text{profit}[sku] = \text{sellingPrice}[sku] - \text{buyingPrice}[sku].$$

Besides basic derivation rules, LogiQL also supports derivation rules that perform, for instance, aggregation. This is supported in an extensible way via the general construct of a higher order *Predicate-to-Predicate Rule* (*P2P rule*). For example, the P2P rule

$$\text{totalShelf}[\,] = u \leftarrow \text{agg}\ll u = \text{sum}(z) \gg \text{Stock}[p] = x, \text{spacePerProd}[p] = y,\ z = x * y.$$

performs a simple weighted sum-aggregation to compute the total amount of required shelf space, which, using abbreviated syntax, can also be written as:

$$\text{totalShelf}[] += \text{Stock}[p] * \text{spacePerProd}[p].$$

It is worth noting that every view definition definable in the relational algebra can be encoded by means of derivation rules (by introducing auxiliary predicates for intermediate results if necessary).[5] A collection of derivation predicates may be defined by rules with a cyclic dependency graph, in which case the rules in question can be viewed as *recursive* view definitions.

Integrity Constraints. An *integrity constraint* is an expression of the form $F \rightarrow G$ (note the use of a rightward arrow instead of a

[5]Indeed, it follows from the Immerman-Vardi theorem [25] that every PTIME-computable view definition is expressible.

// Base predicates:

$$\text{spacePerProd}[p] = v \rightarrow \text{Product}(p), \text{float}(v).$$
$$\text{profitPerProd}[p] = v \rightarrow \text{Product}(p), \text{float}(v).$$
$$\text{minStock}[p] = v \rightarrow \text{Product}(p), \text{float}(v).$$
$$\text{maxStock}[p] = v \rightarrow \text{Product}(p), \text{float}(v).$$
$$\text{maxShelf}[\,] = v \rightarrow \text{float}[64](v).$$

// Derived predicates and rules:

$$\text{Stock}[p] = v \rightarrow \text{Product}(p), \text{float}(v).$$
$$\text{totalShelf}[\,] = v \rightarrow \text{float}(v).$$
$$\text{totalProfit}[\,] = v \rightarrow \text{float}(v).$$
$$\text{totalShelf}[\,] = u \leftarrow \text{agg}\ll u = \text{sum}(z)\gg \text{Stock}[p] = x, \text{spacePerProd}[p] = y,\ z = x * y.$$
$$\text{totalProfit}[\,] = u \leftarrow \text{agg}\ll u = \text{sum}(z)\gg \text{Stock}[p] = x, \text{profitPerProd}[p] = y,\ z = x * y.$$

// Integrity constraints:

$$\text{Product}(p) \rightarrow \text{Stock}[p] \geq \text{minStock}[p].$$
$$\text{Product}(p) \rightarrow \text{Stock}[p] \leq \text{maxStock}[p].$$
$$\text{totalShelf}[\,] = u,\ \text{maxShelf}[\,] = v \ \rightarrow u \leq v.$$

Figure 2: Example constraints and rules.

leftward arrow), where F and G are formulas. Both inclusion dependencies and functional dependencies can be naturally expressed by such expressions. Examples of integrity constraints are:

$$\text{Stock}[p] = v \rightarrow \text{Product}(p), \text{float}(v).$$
$$\text{Product}(p) \rightarrow \text{Stock}[p] = \_.$$
$$\text{totalShelf}[\,] = u, \text{maxShelf}[\,] = v \rightarrow u \leq v.$$

The first constraint can in fact be viewed as a type declaration: it expresses that the key-attribute of the Stock predicate consists of products (Product is a user defined type here), and that the value-attribute is a float. The second constraint is an inclusion dependency that expresses that every product has a stock value. The third constraint expresses that the value of totalShelf[ ] is bounded by the value of maxShelf[ ].

Whereas derivation rules define views, integrity constraints specify the set of legal database states. Traditionally, integrity constraints are used to determine if a transaction succeeds. As we will see below, we also use integrity constraints to encode mathematical optimization problems.

Figure 2 shows an example that involves predicates, derivation rules, and integrity constraints. The program is intended to model a simple retail assortment-planning scenario where the products picked for an assortment cannot take up more space than is available on the shelf.

Reactive Rules. Reactive rules are used to make and detect changes to the database state. They are a special form of derivation rules that refer to versioned predicates and delta predicates [28]. A simple example of a reactive rule is:

$$+\text{sales}[\text{“Popsicle”,2015-01}] = 122.$$

which inserts a new fact into the sales predicate. A more interesting example is the following.

$$\hat{}\text{price}[\text{“Popsicle”}] = 0.8 * x \leftarrow \text{price@}start[\text{“Popsicle”}] = x, \text{sales@}start[\text{“Popsicle”, 2015-01}] < 50, +promo(\text{“}Popsicle''\text{”}, 2015-01).$$

This code discounts the price of popsicles if the sales in January 2015 are lower than 50 units, and there is a promotion being created for popsicles.

As we can see from the above examples, reactive rules are derivation rules that may refer to system-provided *versioned predicates* and *delta-predicates* such as $R@start$ (the content of R at the start of the transaction), $+R$ (the set of tuples being inserted into R in the current transaction), $-R$ (the set of tuples being deleted from R in the current transaction). $\hat{}R$ is a shorthand notation for a combination of $+R$ and $-R$. If R is a base predicate, the content of R after the transaction is determined by means of the following system-provided *frame rules*:

$$R(x_1,\ldots,x_n) \leftarrow R@start(x_1,\ldots,x_n), !(-R(x_1,\ldots,x_n)).$$
$$R(x_1,\ldots,x_n) \leftarrow +R(x_1,\ldots,x_n).$$

The above presentation is much simplified. For more details see [21].

2.2.2 Workspaces and Transactions

A *workspace* consists of (i) a collection of declared predicates, derivation rules, and constraints (collectively called *logic*) and (ii) contents of the base predicates. One can think of a workspace as an instance of the database, at a particular moment in time, including both data and logic. In order to facilitate workspace management, the logic in a workspace is organized in modules that are called *blocks*. Each block is a separate collection of predicate declarations, derivation rules and constraints. The derivation rules and integrity constraints of one block may refer to predicates declared in another block. In order to support programming in the large, LogiQL also supports a more refined module system, involving abstractions and hiding, which we will not explain for lack of space. We will only describe some types of transactions supported by the system.

Query transactions. These transactions are used for querying the workspace state. A query is specified by means of a program that has a designated answer predicate, for example,

$$\text{query}\{\ \_(icecream, week, sales, revenue, profit) \leftarrow \text{week\_sales}[icecream, week] = sales, \text{week\_revenue}[icecream, week] = revenue, \text{week\_profit}[icecream, week] = profit.\}$$

where _ is the designated answer predicate.

Exec transactions. These are used to modify the workspace state by changing the content of base predicates (which may subsequently trigger changes in the content of derived predicates). Exec transactions are specified using reactive logic as described above.

Addblock and Removeblock. These transactions are used to add or remove named collections of rules to the workspace program.

For example,

addblock –*name salesAgg1*{
Sales_yr[*sku*, *store*, *yr*] = *z* ←
agg≪*z* = sum(*s*)≫ Sales[*sku*, *store*, *wk*] = *s*, *year*[*wk*] = *yr.*}

installs a new view into the workspace, and

removeblock salesAgg1

removes it again, thereby restoring the workspace to its prior state. Addblock transactions do not effectuate any changes to existing base predicates. Instead, they may create new base predicates and/or derived predicates, add derivation rules for new or existing derived predicates, and/or add new integrity constraints.

Addblock and removeblock transactions can be viewed as supporting a form of *live programming* (see Section 3.3). In addition, they are central to the workflow of many applications developed using LogiQL, cf. Section 2.1.

Branch and Delete-branch. These transactions create and delete branches of the workspace, that is, copies of the workspace that can be manipulated independently. Branching can be used for creating checkpoints, as well as for supporting speculative what-if analyses. These are crucial in the context of modeling and prediction. As discussed in Section 3.1, the use of purely functional data structures in our implementation makes it possible to support the creation of a new branch as an instantaneous operation, as no actual (deep) copying of data takes place when a new branch is created.

2.3 Prescriptive and Predictive Analytics

The subset of LogiQL presented so far (namely derivation rules, integrity constraints, and reactive rules) allows us to define application logic that traditionally would have to be defined using SQL, SQL triggers, an imperative stored-procedure language such as PL/SQL, and an imperative object-oriented language such as Java or C#.

We now illustrate how LogiQL supports prescriptive and predictive analytics, providing functionality that is not traditionally offered by SQL databases and traditional application servers and is usually supported by the addition of specialized analytic systems to the application architecture.

2.3.1 Prescriptive analytics

By adding one more language feature, LogiQL can be extended to gracefully supports mathematical optimization and prescriptive analytics [8]. The idea is that a predicate $R[x_1,\ldots,x_n] = y$ can be declared to be a *free second-order variable*, which means that the system is responsible for populating it with tuples, in such a way that the integrity constraints are satisfied. Furthermore, a derived predicate of the form $R[\,] = y$ can be declared to be an objective function that should be minimized or maximized.

Continuing the example program in Figure 2, suppose that we would like to automatically compute stock amounts so as to maximize profit. Then, it suffices to add to the program the following lines of code:

lang:solve:variable(‘Stock).
lang:solve:max(‘totalProfit).

The first line is shorthand for an second order existential quantifier and it states that the predicate Stock should be treated as a free second-order variable that we are solving for, while the second line states that the predicate totalProfit is an objective function that needs to be maximized (subject to the integrity constraints).

Under the hood, the program is translated into a Linear Programming (LP) problem and passed on to the appropriate solver, e.g., [36, 3]. LogicBlox grounds (i.e., eliminates the quantifiers in) the problem instance in a manner similar to [33] via automatic synthesis of another LogiQL program that translates the constraints over variable predicates into a representation that can be consumed by the solver. This improves the scalability of the grounding by taking advantage of all the query evaluation machinery in the LogicBlox system (e.g. query optimization, query parallelization, etc). The system then invokes the appropriate solver and populates the value of existentially quantified predicates with the results (turning unknown values into known ones). Furthermore, the grounding logic incrementally maintains the input to the solver, making it possible for the system to incrementally (re)solve only those parts of the problem that are impacted by changes to the input.

If the sample application is changed such that the stock predicate is now defined to be a mapping from products to integers, LogicBlox will detect the change and reformulate the problem so that a different solver is invoked, one that supports Mixed Integer Programming (MIP).

Examples of research in this area include [12, 18, 38, 30]. As far as we know, LogicBlox is the first commercial database system that provides native support for prescriptive analytics.

2.3.2 Predictive analytics

Predictive analytics in LogicBlox was initially supported by means of a collection of built-in machine learning algorithms. This is done via special “predict” P2P rules that come in two modes: the *learning* mode (where a model is being learned) and the *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 predicate Sales[*sku*, *store*, *wk*] = *amount* as well as a predicate Feature[*sku*, *store*, *name*] = *value* that associates with every sku, store 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 a handle to a representation of the model) in the predicate SM[*sku*, *store*] = *model*.

$$\text{SM}[sku, store] = m \leftarrow \text{predict} \ll m = \text{logist}(v|f) \gg \text{Sales}[sku, store, wk] = v\,, \text{Feature}[sku, store, n] = f.$$

The following rule evaluates the model to get specific predictions.

$$\text{Sales\_pred}[sku, store] = v \leftarrow \text{predict} \ll v = \text{eval}(m|f) \gg \text{SM}[sku, store] = m\,, \text{Feature}[sku, store, n] = f.$$

The above rules are evaluated using a built-in machine learning library, which implements a variety of state-of-the-art, scalable machine learning algorithms to support regression, clustering, density estimation, classification, and dimensionality reduction.

2.3.3 Towards Declarative Probabilistic Modeling

The above approach to statistical model building and predictive analytics is not fully satisfying as it is imperative and requires the user to be well-versed in statistical modeling. We briefly discuss some extensions to LogiQL that we are currently developing to make statistical and probabilistic model building more natural by using the native modeling constructs of the language.

Statistical Relational Models. Markov Logic Networks (MLN) [15] and Probabilistic Soft Logic (PSL) [9] are two examples of languages that combine logical and statistical model-

ing. This is accomplished through the use of *soft constraints*, i.e., weighted constraints that are not required to always hold, but whose violations carry a specified penalty.

As an example, suppose that we wish to predict whether customer c will purchase product p. Further assume that we have base predicates that specify similarity among products, $\mathsf{Similar}(p_1, p_2)$; friendship among customers, $\mathsf{Friends}(c_1, c_2)$; and products under promotion, $\mathsf{Promoted}(p)$. Consider the following soft constraints.

$$w_1 : \text{Customer}(c), \text{Promoted}(p) \rightarrow \text{Purchase}(c, p)$$
$$w_2 : \text{Customer}(c), \text{Promoted}(q), \text{Similar}(p, q) \rightarrow !\text{Purchase}(c, p)$$
$$w_3 : \text{Purchase}(d, p), \text{Friends}(c, d) \rightarrow \text{Purchase}(c, p)$$
$$w_4 : !\text{Purchase}(d, p), \text{Friends}(c, d) \rightarrow !\text{Purchase}(c, p)$$

Here, w_1, w_2, w_3 and w_4 are numerical weights. While ordinary (hard) constraints in a LogiQL program specify the set of legal database states, soft constraints assign to each state a score indicating whether it is more or less "likely" compared to other database states. For example, in the case of MLNs, the likelihood of a possible world is proportional to the product of the factors, where a *factor* is defined for each satisfying grounding a rule (in the possible extension) and constitutes (a function of) the weight of that rule. As an example, if the weight of the first rule, w_1, is 2.0, then the likelihood of a possible world is multiplied by $e^{2.0}$ for every product p and customer c that satisfy $\mathsf{Promoted}(p) \rightarrow \mathsf{Purchase}(c, p)$.

In these formalisms, *Maximum-A-Priori* (*MAP*) inference finds the most likely possible world, such as the most likely purchases given partial knowledge about real purchases. This can be formulated as a mathematical optimization problem, which can be solved using the machinery described in Section 2.3.1.

Probabilistic-Programming Datalog. Formalisms for specifying general statistical models, such as probabilistic-programming languages [17], 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 for probabilistic programming, in order to facilitate the design and engineering of machine-learning solutions. Towards that, we have initiated a theoretical exploration of such an extension. In a recent paper [5], we considered the LogiQL fragment of conjunctive derivation rules (i.e., traditional Datalog) and arbitrary integrity constraints. We proposed an extension that provides convenient mechanisms to include common numerical probability functions; in particular, conclusions of rules may contain values drawn from such functions.

As an example, we will consider a case where we wish to detect that a product is being promoted, which leads to a significant increase in sales. We have a relation $\mathsf{Promotion}[p] = b$, determining whether product p is likewise promoted (where $b = 1$ or $b = 0$), where we do not know the value of b. We also have the relation $\mathsf{BuyRate}[p, b] = r$ that determines the probability r that a product p is sold without a promotion ($b = 0$) and with a promotion ($b = 1$). Finally, we have a relation $\mathsf{Customer}(c)$ of customers and a relation $\mathsf{Buys}[c, p] = b$ that determines whether client c purchased product p. The following program models a random behavior of customers ($\text{Flip}[r]$ defines a Bernoulli distribution with parameter r).

$$\text{Promotion}[p] = \text{Flip}[0.01] \leftarrow .$$
$$\text{Buys}[c, p] = \text{Flip}[r] \leftarrow \text{BuyRate}[p, b] = r, \text{Promotion}[p] = b.$$

To determine whether or not a promotion is in place, we use the (say, daily) purchase data to condition the probability space.

$$\text{Visited}(c), \text{Bought}[c, p] = b \rightarrow \text{Buys}[c, p] = b.$$

Here, Visited and Bought are base predicates with known values. The resulting probability space is the one defined by the two derivation rules, conditioned on the following integrity constraint (observations). Inference on the program is supported: for example, one can ask for the most likely value b such that $\mathsf{Promotion}[p] = b$.

Defining the precise semantics of such a program is not at all straightforward, and we refer the interested reader to [5] for the formal framework underlying our proposal.

3. FACETS OF THE IMPLEMENTATION

In this section, we highlight several innovative aspects of the LogicBlox system implementation.

3.1 Branching and persistent data structures

One of the core design choices in the implementation of the LogicBlox engine is the use of persistent data structures. Our structures have *"mutable until shared"* objects that sit at a useful tradeoff point between imperative and purely functional. The objects are mutable when created and while local to a thread, and become immutable at synchronization and branching points (e.g., when the data structure is communicated to another thread, committed, or branched). This allows the efficiency benefits of the imperative RAM model while doing thread-local manipulations, but preserves the *"pointer value uniquely determines extensional state"* property of purely functional data structures when objects are shared, which simplifies the programming model for incremental maintenance, concurrent transactions, and live programming. In addition, our internal framework transparently persists, restores, and garbage-collects these objects as needed, which greatly simplifies the engineering of many aspects of the runtime internals.

Persistent data structures are used for both paged relational data and meta-data, which are C++ objects representing for instance LogiQL programs and workspaces. For paged data, we use a family of persistent B-tree-like data structures. Collections of meta-data objects such as sets, vectors, and maps are implemented as treaps. These are randomized binary search trees that offer efficient search, insertion, and deletion [37]. Treaps have the unique representation property: the structure of the tree depends only on its contents, not on the operation history. With memoization, this permits extensional equality testing in $O(1)$ time, using pointer comparison. The treaps are purely functional [35], i.e., the treap nodes cannot be modified once constructed and all mutating (insert/erase) operations are performed by duplicating a path from the root to where the change occurs. Set intersection, union, and difference are also efficient [7].

Recall that the *branch* transaction command of LogiQL creates a new branch of a workspace. Since both data and the meta-data are stored using persistent data structures, this is an $O(1)$ operation. Moreover, the branching functionality, coupled with functional-style techniques, enables versioned data structures that provide efficient search and set operations such as the difference between versions of objects. Efficient diffing is crucial for incremental maintenance, while efficient branching is used by our lock-free concurrent transactions to execute in parallel on different branches of the current version of the database.

3.2 Queries and incremental maintenance

Leapfrog Triejoin (LFTJ) [42] is a join algorithm developed at LogicBlox and can be seen as an improved version of the classical

sort-merge join algorithm. It is used to compute the derived predicates (more specifically, it enumerates the satisfying assignments for bodies of derivation rules). We next give an overview of the algorithm and its properties. To simplify the presentation, we focus on equi-joins. As the details of this algorithm are needed for describing various important aspects of our system (e.g., indices, concurrency and incremental maintenance), the description here is more elaborate than that of the other components we consider in this paper.

LFTJ for unary predicates. We first describe how LFTJ works for the special case of joining unary predicates $A_1(x),\ldots,A_k(x)$. This case is of no particular novelty (see, e.g., [24, 14]), but it serves as the basic building block towards the general case.

The input predicates A_i are assumed to be given in sorted order, and accessible though a *linear iterator* interface that supports the following methods:

- next() proceeds to the next value (i.e., tuple of the unary A_i).
- seek(v) proceeds to the least upper bound for v; that is, it positions the iterator at the smallest value u satisfying $u \geq v$, or the end if no such u exists. The given v must be greater or equal to the value at the current position.

These methods are required to take $O(\log N)$ time, where N is the cardinality of the predicate. Moreover, if m values are visited in ascending order, the amortized complexity is required to be $O(1+\log(N/m))$, which can be accomplished using standard data structures such as B-trees. Initially, the linear iterator method is located at the first value in the predicate.

The unary LFTJ algorithm itself implements the same *linear iterator* interface: it provides an iterator for the join of A_1 to A_k, i.e., for the intersection $A_1 \cap \cdots \cap A_k$, that efficiently supports next() and seek(). The algorithm maintains a priority queue of pointers to input iterators, one for each predicate being joined, where the priority reflects the value at which the iterator is currently positioned. The algorithm repeatedly takes an iterator with the smallest value and performs a seek for the largest value, "leapfrogging" the iterators until they are all positioned at the same value.

Figure 3 illustrates a join of three unary predicates, A, B, and C, with $A = \{0,1,3,4,5,6,7,8,9,11\}$, $B = \{0,2,6,7,8,9\}$, and $C = \{2,4,5,8,10\}$. Initially, the iterators for A, B, and C are positioned at the first elements 0, 0, and respectively 2. Since 2 is the largest from these three values, the iterator for A performs a seek(2) which positions the iterator for A at 3. The iterator for B then performs a seek(3), which lands at 6. The iterator for C does seek(6), which lands at 8, and so on. LFTJ ends when the iterator for B, while performing seek(11), reaches the end.

Sensitivity intervals. As LFTJ calls the next() and seek() methods of the participating iterators, there is a natural notion of a *sensitivity interval*: an interval where changes may actually affect the result of the LFTJ computation. For instance, in the example in Figure 3, inserting the fact $C(3)$ or deleting the fact $C(4)$ would not affect the computation, given that the seek(6) instruction skips over these values anyway. The sensitivity intervals, i.e., intervals where changes *do* affect the LFTJ computation, for the example in Figure 3 are as follows:

$$A: [-\infty,0],[2,3],[8,8],[10,11]$$
$$B: [-\infty,0],[3,6],[8,8],[11,+\infty]$$
$$C: [-\infty,2],[6,8],[8,10]$$

The sensitivity indices can, in this example, also be viewed as representing an execution trace of the LFTJ algorithm. In the LogicBlox runtime, sensitivity intervals play an important role for both incremental maintenance and concurrent transactions.

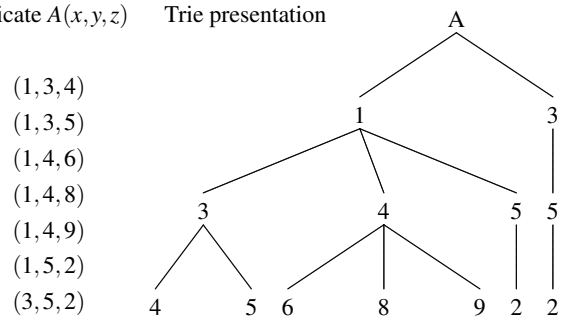


Figure 4: Example: Trie presentation of a ternary predicate.

Arbitrary arity. Non-unary input predicates, such as $A(x,y,z)$, are (logically) presented as *tries*, that is, trees in which each level corresponds to a different argument position, and each tuple $(a,b,c) \in A$ corresponds to a unique root-to-leaf path. The linear iterator interface is augmented with two methods for vertical trie navigation:

- open() for proceeding to the first value at the next depth (i.e., the first child of the current node in the trie).
- up() for returning to the parent of the current node in the trie.

These methods can also be supported in $O(\log N)$ time for the input predicates. The next() and seek() methods perform horizontal navigation between siblings in the trie, as before.

We now describe the LFTJ algorithm for equi-joins of predicates of arbitrary arity. We assume that each variable appears at most once in each atom. For example, $R(x,x)$ can be rewritten to $R(x,y),x=y$ to satisfy this requirement. Likewise, we may assume that constants may not appear in the query: a subformula such as $A(x,2)$ is rewritten to $A(x,y)$, $\mathsf{Const}_2(y)$, where $\mathsf{Const}_2 = \{2\}$. Here, the equality ($=$) and Const_2 are virtual (i.e., non-materialized) predicates that allow for efficient access through the same trie-iterator interface.

The algorithm requires a *variable ordering* that is consistent with the order in which variables occur in the atoms in the query. For instance, in the join $R(a,b),S(b,c),T(a,c)$ we might choose the variable ordering $[a,b,c]$. In cases where no consistent variable order exists, such as for the join $R(a,b,c),S(c,b)$, a secondary index is required on one of the two predicates. For example, if the chosen variable ordering is $[a,b,c]$, a secondary index on S is needed that efficiently supports access to $S'(c,b) = S(b,c)$ via the trie-iterator interface.

Once a variable order is chosen, the algorithm proceeds by performing a unary LFTJ for each variable. Consider the example $R(a,b)$, $S(b,c)$, $T(a,c)$ with the variable ordering $[a,b,c]$. We first apply the unary LFTJ algorithm to enumerate the values for a that are in both the projections $R(a,\_)$ and $T(a,\_)$. For each successful binding for a, we proceed to the next variable in the chosen order. Again, a unary LFTJ is performed to enumerate bindings for b that satisfy both $R(a,b)$ and $S(b,\_)$ (for the current value of a). Finally, for each successful binding of b, we proceed to the variable c, and perform a unary LFTJ to enumerate values for c satisfying both $S(b,c)$ and $T(a,c)$ (for the current values of a and b). Each time a unary LFTJ run finishes, having exhausted its list of possible bindings, we retreat to the previous level and seek another binding for the corresponding variable. Thus, conceptually, we can regard the entire LFTJ run as a backtracking search through a trie of potential variable bindings.

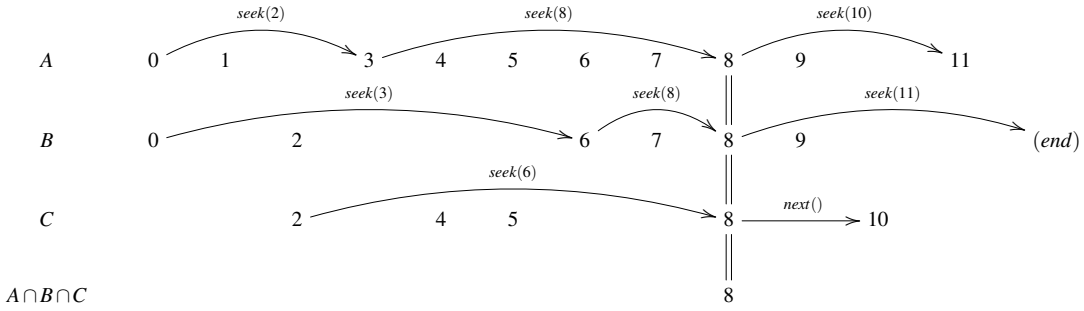


Figure 3: Example of a LFTJ run with unary predicates.

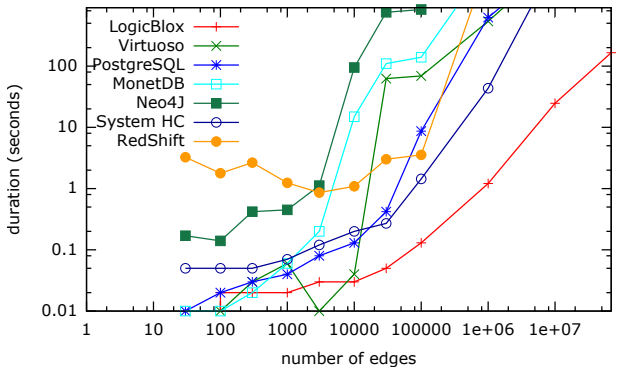


Figure 5: Running time of the 3-clique query on (increasingly larger subsets of) the LiveJournal graph dataset [26] using LogicBlox 4.1.4, Virtuoso 7, PostgreSQL 9.3.4, Neo4j 2.1.5, MonetDB 1.7 (Jan2014-SP3), a commercial in-memory column store (System HC), and RedShift. See [32] for full details and more experiments.

LFTJ is a *worst-case optimal join algorithm* for any equi-join query, in the following sense (cf. [31]): the running time of the algorithm is bounded by the worst-case cardinality of the query result (modulo logarithmic factors) [42].

Scenarios where LFTJ excels particularly compared to other join algorithms are multi-way joins such as the join returning all 3-cliques on the LiveJournal graph dataset, cf. Figure 5.

Optimization and parallelism. When joins are evaluated using LFTJ, query optimization essentially boils down to choosing a good variable order. Recall that, depending on the chosen variable order, secondary indices may need to be created and maintained for some predicates. The LogicBlox query optimizer uses sampling-based techniques: small representative samples of predicates are maintained. These samples are used to compare candidate variable orderings for LFTJ evaluation, and, consequently, also for automatic index creation. Furthermore, the samples are used to determine domain decompositions for automatic parallelization.

Incremental maintenance. As we explained in the introduction, incremental maintenance is of central importance in LogicBlox applications. We support incremental maintenance of derived predicates efficiently by means of an extension of the LFTJ algorithm. The basic setup is as follows. We have a derivation rule such as

$$T(x) \leftarrow A_1(x,y), A_2(y,z), A_3(x,z)$$

and, for each input predicate A_i, we are given an *old* version A_i^{old}, a *new* version A_i^{new}, and a *delta* predicate A_i^{Δ}, where A_i^{Δ} is a set of insertions and deletions that, when applied to A_i^{old}, yields A_i^{new}. We are also given T^{old}, and the task is to compute T^{new} as well as T^{Δ} which is then propagated forward to other rules. Recall that our versioned data structures allow us to compute differences between two versions of a predicate efficiently.

The derivation rule maintenance problem is, conceptually, split into two parts: *maintaining the set of satisfying assignments for the rule body under changes to the input predicates* ("rule body maintenance") and *maintaining the rule head predicate under changes to the set of satisfying assignments for the rule body* ("rule head maintenance"). For recursive LogiQL programs, additional machinery is used to maintain the results of fixpoint computations.

Rule head maintenance is implemented using a variety of data structures for different derivation rules. In the case of the above example, a count predicate is maintained, indicating, for each value a, the number of different ways in which it is derived (i.e,. the number of satisfying assignments of the rule body that support the derivation of that value). For P2P rules performing operations such as aggregation, different data structures are used. Support for efficient *rule body maintenance* is provided by the LFTJ algorithm through the use of *sensitivity indices*. These sensitivity indices maintain sensitivity intervals for a LFTJ run at various levels in the trie and in various contexts (where the context of a sensitivity interval consists of the values for earlier-chosen variables under which the sensitivity occurs). The maintenance algorithm is designed to be optimal in the sense that the cost of maintenance under changes to the input predicates is proportional to the trace-edit distance of the corresponding runs of the LFTJ algorithm [41].

3.3 Live programming and self service

In order to support end-user driven application evolution, we have added support for live programming in LogicBlox, where the traditional edit-compile-run cycle is abandoned in favor of a more interactive user experience with live feedback on a program's runtime behavior [10]. For instance, in our retail planning applications, we enable users to define and change schemas and formulas on the fly. These changes trigger updates to the application code in the database server and the challenge is to quickly update the user views in response to these changes. Section 2.1 describes such a scenario.

From a technical perspective, live programming is far from trivial—especially when working with programs and data of the scale encountered in the real world (e.g., a typical application has about 5k LogiQL rules and several TBs of data). To achieve interactive response times in those scenarios, changes to application

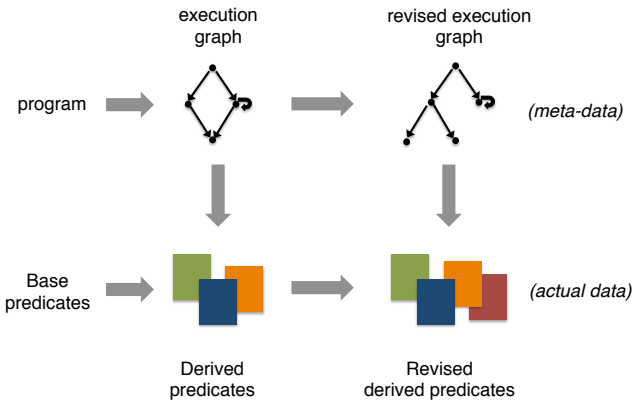


Figure 6: While the engine proper deals with maintenance of the derived predicates (materialized views) for a given program (left half), the meta-engine maintains the program under code updates and informs the engine proper which derived predicates should be revised (right half).

code must be quickly compiled and "hot-swapped in" to the running program, and the effects of those changes must be efficiently computed in an incremental fashion.

Live programming is supported in the LogicBlox system via the meta-engine, which is a *higher-level lightweight engine* that manages metadata representing LogiQL application code. The meta-engine maintains the program state declaratively and incrementally. Figure 6 depicts schematically how the meta-engine differs from the engine proper and how the two engines work together. The LogiQL program is compiled into an execution graph, where the predicates are the nodes and the edges between nodes represent rules. The engine proper evaluates the execution graph bottom-up on the input database and materializes the derived predicates. The meta-engine is activated when the program changes: it incrementally maintains the execution graph (depicted by a revised execution graph) and informs the engine proper which derived predicates have to be maintained as result of the program change. To achieve this, the meta-engine uses *meta-rules* to declaratively describe the LogiQL code as collections of meta-facts and their relationships. These collections are implemented using persistent data structures, cf. Section 3.1. There are currently about 200 meta-rules that support the incremental code maintenance mechanism and various code invariants and optimizations, such as rule inlining and on-the-fly creation of sampling rules to support LFTJ optimization. The meta-rules are expressed in non-recursive Datalog with negation, and additional functionality is available via an extensible set of primitive built-ins and computed meta-predicates.

Compared with an imperative specification of the complicated state transitions involved in live programming, the declarative approach is simpler, uses less code, is more robust, and is easily extensible.

We next give two examples of actual meta-rules. The first example meta-rule is used to determine whether a given predicate is a base predicate (recall from Section 2 that the user is not required to specify if a given predicate is a base predicate or a derived predicate, and that, when not specified explicitly, the information in question is derived from the way the predicate is used):

$$\text{lang\_edb}(name) \leftarrow \text{lang\_predname}(name), !\text{lang\_idb}(name).$$

The meta-predicate lang_edb denotes the set of all base predicates, whereas the meta-predicate lang_idb denotes the set of all derived predicates. The above meta-rule states that every predicate that is not implied (by other meta-rules) to be an derived predicate is a base predicate (the ! symbol is used for negation). In a live programming application, any of the two body meta-predicates may change, i.e., new predicates are declared or existing ones are removed, and the meta-engine updates the lang_edb meta-predicate (i.e., the set of base predicates) after each such change.

As a second example, we give (a simplified version of) the meta-rule that determines whether frame rules are needed for a specific base predicate. Recall from Section 2 that frame rules are used to compute the new extension of an base predicate Foo based on the previous version Foo@*start* together with the extension of the corresponding delta-predicates +Foo and −Foo. The system needs to maintain the following logical invariant: *if +Foo or −Foo appears in the head of a rule, then we need a frame rule for Foo.* The following meta-rule maintains this invariant:

$$\begin{aligned}\text{need\_frame\_rule}(predName) \leftarrow\ & \text{user\_rule}(\_, rule),\\ & \text{rule\_head}[rule] = head, \text{head\_predicate}(head, predName),\\ & \text{is\_delta\_predicate}(predName).\end{aligned}$$

Here, the meta-predicate needs_frame_rule denotes the set of delta predicates for which a frame rule needs to be generated. The above meta-rule inspects the head of each user rule and checks whether it contains a delta predicate.

3.4 Transaction Repair

In this section we describe our ongoing effort to support *concurrent write-transactions*. Recall that the LogicBlox engine aims to provide a unified platform for mixed workload data processing. As such, efficient and transparent support for concurrent transactions is of crucial importance.

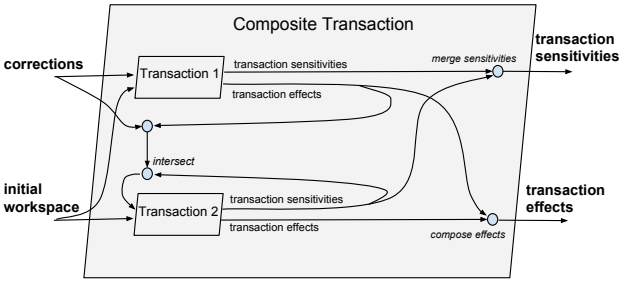
Traditional approaches to concurrent transactions either settle for lower levels of isolation, such as Snapshot Isolation, or employ costly locking mechanisms that reduce the potential for parallel speedup. Instead, our approach, which we call *transaction repair* [43], provides full serializability (the highest level of correctness and semantic transparency) while avoiding the use of locks. Unlike other concurrency mechanisms that support full serializability through the use of versioned data structures, transaction repair is able to accommodate large-footprint write-transactions efficiently, which are common in planning scenarios. To do so, we take advantage not only of our purely functional and versioned data structures, but also of the fact that our system supports efficient incremental maintenance:

1. Branching an entire workspace is an $O(1)$ operation due to the use of persistent data structures, cf. Section 3.1. This means that we can run transactions simultaneously, with no locking, each in a separate branch of the database. This immediately supports a "multiple reader, single writer" form of concurrency.

2. As explained in Section 3.2, sensitivity indices can be used to track which changes in the underlying data may affect the evaluation of derivation rules. In the same way, sensitivity indices can be used to track what changes to a workspace may affect the execution of a transaction. In particular, we can use sensitivity indices to detect conflicts between concurrent transactions.

3. In case of conflicts, our incremental maintenance machinery can be used to maintain the result of the second transaction under relevant changes resulting from the first transaction, i.e., to *repair transactions*. This is what allows us to support concurrent transactions with full serializability without using locks.

(a) A single transaction



(b) Composition of two transactions

Figure 7: The transaction-repair framework

4. Once all conflicts for the group of transactions in flight (or an initial segment) have been repaired, they can be committed together.

Below, we describe the transaction-repair framework in somewhat more detail.

Transaction Maintenance In the transaction-repair framework, a transaction takes as input a workspace and any number of corrections (which may be supplied over time) and provides as output *transaction effects* and *transaction sensitivities*. Moreover, these output effects and sensitivities are kept up-to-date as corrections are received. This is depicted graphically in Figure 7(a).

The transaction sensitivities are not the intervals where changes could affect the output workspace, but rather intervals where changes could affect the *transaction effects*, i.e.,, the changes that, when applied to the corrected input workspace, yield the output workspace.

For example, suppose the workspace contains base predicates inventory and auto_order, and a derived predicate place_order defined by the derivation rule

$$\text{place\_order}(x) \leftarrow \text{inventory}[x] = 0, \text{auto\_order}(x).$$

Consider the transaction

exec{
  ^inventory["Popsicle"] = $x \leftarrow$
    inventory@*start*["Popsicle"] = $y, x = y - 1$.}

(Assume in the discussion below that l stands for "Popsicle".) If the input workspace contains a record inventory$[l] = 2$, then the transaction effects will include

$$-\text{inventory}[l] = 2 \text{ and } +\text{inventory}[l] = 1,$$

while the transaction sensitivities for the predicate inventory will contain the (singleton) interval $[l,l]$. Now, suppose the transaction receives the above two side effects on inventory as incoming corrections. Then, the updated transaction effects will include

$$-\text{inventory}[l] = 1, +\text{inventory}[l] = 0, \text{ and } +\text{place\_order}(l),$$

and the sensitivities for auto_order will include the singleton interval $[l,l]$. Note that the precise set of sensitivity indices recorded depends on the execution of the LFTJ algorithm, and, more specifically, the order in which the iterators are visited.

Transaction Circuits Two transactions performed concurrently can be composed into a single object that implements the same interface described above, cf. Figure 7(b).

Observe that, in the case where transaction effects of the first transaction do not intersect with the transaction sensitivities of the second transaction (and no incoming corrections are received), each transaction is executed exactly once, and no incremental maintenance is needed. In general, however, the second transaction may need to be incrementally repaired multiple times.

By repeated application we can treat an entire sequence of transactions as a single composite transaction, implemented through a binary tree-shaped circuit whose nodes correspond to different processes that are run in parallel. This is a simplified picture, and further details can be found in [43].

Illustration: Transaction Repair vs Row-Level Locking. Consider the following variation of the above example, involving a large number of transactions. Each transaction adjusts a number of items in the `inventory` predicate. Suppose there are n items in total, and each transaction modifies the `inventory` value for any given item with independent probability $\alpha n^{-1/2}$, for some fixed parameter $\alpha > 0$. That is, each transaction has the form

exec{
  ^inventory$[S_1] = x \leftarrow$ inventory@*start*$[S_1] = y, x = y - 1$.
  $\vdots$
  ^inventory$[S_k] = x \leftarrow$ inventory@*start*$[S_k] = y, x = y - 1$.}

where $S_1, \ldots, S_k$ are items chosen independently and at random with probability $\alpha n^{-1/2}$. Most pairs of transactions will conflict when $\alpha \gg 1$: the expected number of items common to two transactions is $E[\cdot] = n \cdot (\alpha n^{-1/2})^2 = \alpha^2$, an instance of the Birthday Paradox. Row-level locking [6] is a bottleneck when $\alpha \gg 1$: since most transactions have items in common, they quickly encounter lock conflicts and are put to sleep. Even with an efficient implementation of row-level locking, on a multi-core machine, only a limited parallel speedup can be obtained unless the expected number of conflicts is very small (say, $\alpha = 0.1$). Even for $\alpha = 1$, parallel speedup is sharply limited; and for $\alpha = 10$ almost no parallel speedup is possible. Transaction repair allows us to achieve near-linear parallel speedup in the number of cores, even for high values of α such as $\alpha = 10$ [43].

4. CONCLUSION

We presented the design considerations behind the LogicBlox system architecture, and gave an overview of the implementation of the system. We highlighted several innovative aspects, which include: LogiQL, a unified and declarative language based on Datalog; the use of purely functional data structures; novel join processing strategies; advanced incremental maintenance and live programming facilities; and built-in support for prescriptive and predictive analytics.

Our program for the LogicBlox system is admittedly an ambitious one, and replacing the enterprise hairball requires a pragmatic and incremental approach. But already today, the LogicBlox platform has matured to the point that it is being used daily in dozens of mission-critical applications in some of the largest enterprises in the world, whose aggregate revenues exceed $300B.

5. REFERENCES

[1] http://www.datalog20.org/programme.html.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Tobias Achterberg. SCIP: Solving constraint integer programs. *Math. Programming Computation*, 1(1):1–41, 2009.

[4] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in Bloom: a calm and collected approach. In *CIDR*, 2011.

[5] Vince Barany, Balder ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. Declarative statistical modeling with Datalog. *arXiv:1412.2221*, 2014.

[6] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[7] Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *SPAA*, pages 16–26, 1998.

[8] C. Borraz-Sánchez, J. Ma, D. Klabjan, R. Fourer, E. Pasalic, and M. Aref. Algebraic modeling in Datalog. http://dynresmanagement.com/uploads/3/3/2/9/3329212/datalog_modeling.pdf.

[9] Matthias Bröcheler, Lilyana Mihalkova, and Lise Getoor. Probabilistic similarity logic. In *UAI*, pages 73–82, 2010.

[10] Brian Burg, Adrian Kuhn, and Chris Parnin. First Int. Workshop on Live Programming (LIVE). In *ICSE*, pages 1529–1530, 2013.

[11] S. Swaminathan C. J. Date, A. Kannan. *An Introduction to Database Systems*. Pearson Education, eighth edition, 2008.

[12] Marco Cadoli, Giovambattista Ianni, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. Np-spec: an executable specification language for solving all problems in np. *Computer Languages*, 26(2):165–195, 2000.

[13] B. Chin, D. von Dincklage, V. Ercegovak, P. Hawkins, M. S. Miller, F. Och, C. Olston, and F. Pereira. Yedalog: Exploring knowledge at scale. In *SNAPL*, 2015. To appear.

[14] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *SODA*, pages 743–752, 2000.

[15] Pedro Domingos and Daniel Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on AI and Machine Learning. 2009.

[16] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, 2012.

[17] Noah D. Goodman. The principles and practice of probabilistic programming. In *POPL*, pages 399–402, 2013.

[18] Sergio Greco, Cristian Molinaro, Irina Trubitsyna, and Ester Zumpano. NP datalog: A logic language for expressing search and optimization problems. *TPLP*, 10(2):125–166, 2010.

[19] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.

[20] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally (extended abstract). In *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data SIGMOD'93*, pages 157–166, Washington, DC, 1993.

[21] Terry Halpin and Spencer Rugaber. *LogiQL: A Query Language for Smart Databases*. CRC Press, 2014.

[22] Terry A. Halpin, Matthew Curland, Kurt Stirewalt, Navin Viswanath, Matthew J. McGill, and Steven Beck. Mapping ORM to datalog: An overview. In *OTM Workshops'10*, pages 504–513, 2010.

[23] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. In *SIGMOD*, pages 1213–1216, 2011.

[24] Frank K. Hwang and Shen Lin. A simple algorithm for merging two disjoint linearly-ordered sets. *SIAM J. Comput.*, 1(1):31–39, 1972.

[25] N. Immerman. *Descriptive Complexity*. Springer, 1999.

[26] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[27] Leonid Libkin. SQL's three-valued logic and certain answers. In *ICDT*, 2015. To appear.

[28] Bertram Ludäscher. *Integration of Active and Deductive Database Rules*, volume 45 of *DISDBIS*. Infix Verlag, St. Augustin, Germany, 1998.

[29] David Maier and David S. Warren. *Computing With Logic: Logic Programming With Prolog*. Addison-Wesley, 1988.

[30] Alexandra Meliou and Dan Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.

[31] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012.

[32] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. arXiv:1503.04169, 2015.

[33] Feng Niu, Christopher Ré, AnHai Doan, and Jude W. Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *PVLDB*, 4(6):373–384, 2011.

[34] Tim Furche Andrew Sellers Oege de Moor, Georg Gottlob, editor. *Datalog Reloaded: Proceedings of the First International Datalog 2.0 Workshop*. Springer, 2011.

[35] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1999.

[36] Gurobi Optimization. Gurobi optimizer reference manual, 2015.

[37] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[38] Timothy E. Sheard. Painless programming combining reduction and search: Design principles for embedding decision procedures in high-level languages. In *ICFP*, pages 89–102, 2012.

[39] Michael Stonebraker and Ugur Cetintemel. "One size fits all": An idea whose time has come and gone. In *ICDE*, pages 2–11, 2005.

[40] Todd L. Veldhuizen. Incremental maintenance for leapfrog triejoin. *CoRR*, abs/1303.5313, 2013.

[41] Todd L. Veldhuizen. Incremental maintenance for leapfrog triejoin. Technical Report LB1202, LogicBlox Inc., 2013. arXiv:1303.5313.

[42] Todd L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.

[43] Todd L. Veldhuizen. Transaction repair: Full serializability without locks. *CoRR*, abs/1403.5645, 2014.