Live Programming in the LogicBlox System: A MetaLogiQL Approach

Todd J. Green Dan Olteanu Geoffrey Washburn
LogicBlox, Inc.*
firstname.lastname@logicblox.com

ABSTRACT

The emerging category of self-service enterprise applications motivates support for "live programming" in the database, where the user's iterative data exploration triggers changes to installed application code and its output in real time.

This paper discusses the technical challenges in supporting live programming in the database and presents the solution implemented in the LogicBlox commercial system. The workhorse architectural component is a "meta-engine" that incrementally maintains metadata representing application code, guides its compilation into an internal representation in the database kernel, and orchestrates maintenance of materialized views based on those changes. Our approach mirrors LogicBlox's declarative programming model and describes the maintenance of application code using declarative meta-rules; the meta-engine is essentially a "bootstrap" version of the database engine proper.

Beyond live programming, the meta-engine turns out effective for a range of static analysis and optimization tasks. Outside of the database context, we speculate that our design may even provide a novel means of building incremental compilers for general-purpose programming languages.

Categories and Subject Descriptors

H.2 [**Database Management**]

General Terms

Algorithms, Design, Languages

Keywords

LogicBlox, LogiQL; Datalog; Incremental Maintenance

*We wish to acknowledge the many people at LogicBlox who contributed to the development of the LogicBlox 4.X platform, in particular Todd Veldhuizen for discussions on the runtime engine, Shan Shan Huang for introduction to self-service enterprise applications, and Feliks Kluźniak for contributions to the meta-engine implementation.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

Yo dawg, we heard you like Datalog engines...
so we put a Datalog engine in your Datalog engine,
so you can derive while you derive![1]

1. INTRODUCTION

An increasing amount of self-service enterprise applications require *live programming* in the database, 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 [6]. For instance, in retail-planning spreadsheets backed by scalable full-fledged database systems, users can define and change schemas of pivot tables and formulas over these schemas on the fly. These changes trigger updates to the application code on the database server and the challenge is to quickly update the user spreadsheets in response to these changes.

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. To achieve interactive response times in those scenarios, changes to application code must be quickly compiled and "hot-swapped" into the running program, and the effects of those changes must be efficiently computed in an incremental fashion.

In this paper, we discuss the technical challenges in supporting live programming in the database. The workhorse architectural component is a "meta-engine" that incrementally maintains metadata representing application code, guides its compilation into an internal representation in the database kernel, and orchestrates maintenance of materialized results of the application code based on those changes. In contrast, the engine proper works on application data and can incrementally maintain materialized results in the face of data updates. The meta-engine instructs the engine which materialized results need to be (partially or completely) recomputed. Without the meta-engine, the engine would unnecessarily recompute from scratch all materialized results every time the application code changes and would render the system unusable for live programming.

We present the meta-engine solution that we designed and implemented in the LogicBlox commercial system[2] [4]. LogicBlox offers a unified runtime for the enterprise software stack that sharply contrasts with currently popular stacks with dozens of specialized systems and programming languages, where non-trivial integration effort is usually spent at the interface between these systems.

[1]With apologies to Xzibit and "Pimp My Ride"
[2]`http://www.logicblox.com`

LogicBlox applications are written in an extension of Datalog [22, 2, 11] called LogiQL[3] [15]. 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 [9, 16]. Other recent industrial systems based on Datalog include Google's Yedalog [7], Datomic[4], and EVE[5].

There are several key aspects of LogiQL that influenced the design of the meta-engine:

- LogiQL acts as a declarative programming model unifying OLTP, OLAP, and prescriptive and predictive analytics.
- It offers rich language constructs for expressing derivation rules (defining queries and views), integrity constraints, event-condition-action rules (defining updates), mathematical optimization, and predictive analytics.
- It supports rule templates and modules to enable large, modularized declarative code.
- Several code optimizations are expressed in LogiQL and generated on demand, e.g., rules to create indices to speed up join processing, to cope with event-condition-action rules in case of data updates, or inline view definitions.
- A wealth of logical invariants are checked at compile time to ensure the correctness of LogiQL programs. Their interaction is non-trivial and it has become increasingly challenging to enforce them in the right order in monolithic imperative implementations.

These aspects make possible new classes of hybrid applications within the same platform. Being declarative and high-level, LogiQL code takes much less space than equivalent imperative code and is less error prone. It is consistently reported in practice that high-level languages increase productivity and improve the maintainability and agility of software development, e.g., [1, 20]. Since LogiQL acts as a unifying programming model for various workloads that enables the application logic to be written entirely in LogiQL, LogiQL programs tend to be much larger than the usual Datalog programs reported in the research literature. For instance, a forecast manager, which is a typical LogicBlox application in the retail sector, needs about 50K lines of LogiQL code versus millions of lines of C++ code; similar observations on the gap of code complexity have been previously made for Datalog-based declarative networking [21] and metacompilation [8].

Building the LogicBlox meta-engine to effectively support live programming is a software engineering challenge. We approached it by relying on database and programming language principles and set for declarative, high-level and extensible metadata management: we effectively use declarative programming to improve the implementation of a declarative database system [8]. In particular, the meta-engine uses rules expressed in a Datalog-like language called MetaLogiQL[6]; these operate on metadata representing LogiQL rules to capture the logical invariants of LogiQL programs and related transformations and optimizations. In the current LogicBlox version, there are 200+ meta-rules and more existing imperative code for metadata is increasingly migrating to meta-rules. There are several advantages of this declarative approach over an imperative specification of the meta-engine, including less code, less error prone code, correctness guarantees while providing efficient maintenance, and easy extensibility via new MetaLogiQL rules.

Although the functionality of the meta-engine is theoretically subsumed by that of the engine proper, the former is much more lightweight as it does not need the extensive code optimizations of the latter, and it works directly on an objected-oriented representation of LogiQL code that is not naturally presented in a relational format as required by the latter.

In the rest of the paper, we introduce various aspects of the meta-engine, including its persistent, purely-functional metadata layer, the language MetaLogiQL, and the incremental maintenance and rule materialization mechanisms.

2. LIVE PROGRAMMING APPLICATION

Live programming is indispensable in interactive planning applications. In this section, we describe the modeler, a self-service OLAP application built on top of LogicBlox, which requires live programming in the database, and discuss a variety of common cases where changes to application code happen and need to be addressed efficiently.

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 (hierarchical) tables, some are editing the data to specify different future promotional strategies 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.

LogicBlox supports multiple concurrent users and processes via workbooks, which are branches of (fragments of) the database that can be modified independently. 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 can take several hours of machine time to run.

Figure 1 shows two screenshots of the modeler in which users can visualize their data and model, both of which can be evolved in order to reflect new knowledge about their business. The modeler presents the users with a spreadsheet interface listing possible dimensions, which are attributes of

[3]Pronounced "logical"
[4]`www.datomic.com`
[5]`www.incidentalcomplexity.com`
[6]Pronounced "metalogical"

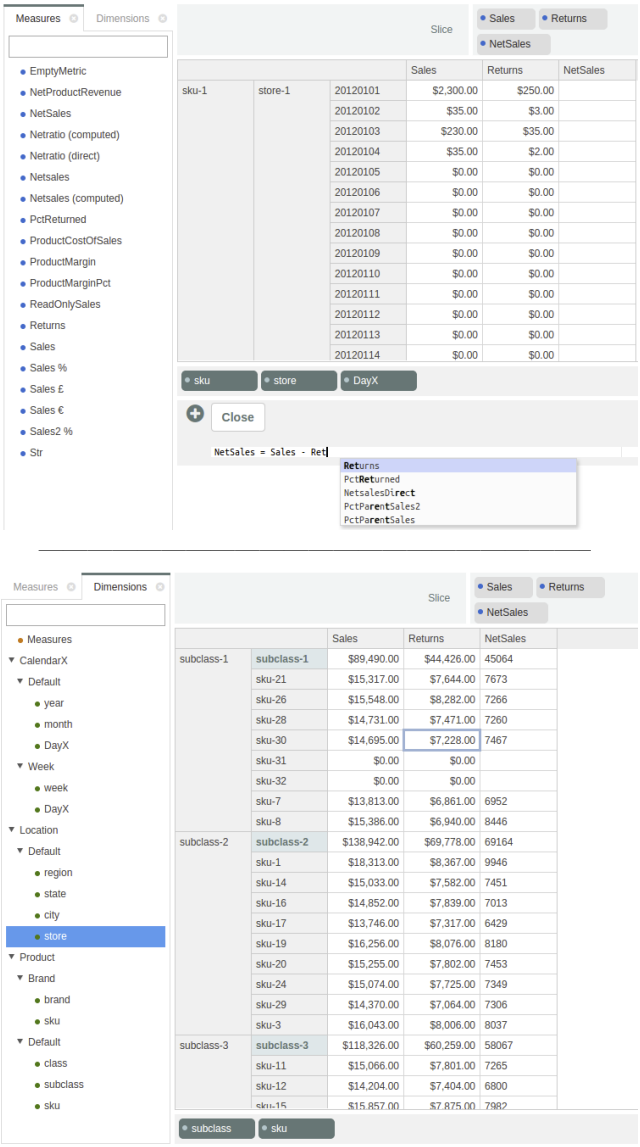

Figure 1: Excerpts from screenshots of a retail planning application requiring live programming.

the database schema, and existing measures, which are materialized views over the input data expressed in LogiQL. The users can explore the data by defining and changing the pivot table over dimensions and measures on the fly. This contrasts with existing static approaches with predefined dimensions such as traditional BI tools and OLAP cubes. Users can also define and change measures, which are formulas over the existing schema. This in turn triggers the update of existing LogiQL code for materialized views at the database server.

We next discuss common cases of changes to the application code in the LogicBlox modeler and how the meta-engine can be beneficial in each of these cases.

2.1 Derivation rules

Figure 1 (top) shows how the user defines a new formula for NetSales over existing measures Sales and Returns, which are in turn defined over dimensions *sku* (stock keeping unit), *store*, and *day*. The pivot table is presented grouped by the dimensions *sku*, *store*, and *day* with measures Sales, Returns, and NetSales computed for each distinct triple of these dimensions. This formula definition triggers the addition of a LogiQL derivation rule defining a new predicate (relation) NetSales, along with updates to the modeler configuration predicates. The formula for NetSales is translated to a LogiQL derivation rule:

$$\begin{aligned}\mathsf{NetSales}[sku,store,day] = v \leftarrow\ & \\ & \mathsf{Sales}[sku,store,day] = v_0, \\ & \mathsf{Returns}[sku,store,day] = v_1, v = v_0 - v_1.\end{aligned}$$

which states that, for a given tuple of *sku*, *store*, and *day*, NetSales is the difference between Sales and Returns. The bracket notation emphasizes that all three predicates are functions of the parameters between the brackets.

Figure 1 (bottom) shows the pivot table after it is aggregated by the subclass of products for each sku, with Sales, Returns, and NetSales measures. Again, requesting this aggregated pivot view triggers an update to the LogiQL program whose result is then shown in the spreadsheet. A snippet of the LogiQL code for this aggregation is given by the next derivation rule:

$$\begin{aligned}\mathsf{Sales\_at\_subclass}[subclass] = v \leftarrow \mathrm{agg}\ll v = \mathrm{sum}(v_0) \gg\ & \\ & \mathsf{Sales}[sku,store,day] = v_0, \\ & \mathsf{sku\_to\_subclass}[sku] = subclass.\end{aligned}$$

The above derivation rule is a Predicate-to-Predicate Rule (P2P rule) and sums up all sales for a particular subclass of products. Similar P2P rules are derived for Returns and NetSales.

In both examples above, the meta-engine would just add the new derived predicates without recomputing from scratch the entire program including the existing predicates. This can lead to huge performance savings in practice.

Besides derivation rules, the application can trigger updates for code representing integrity constraints (e.g., inclusion or functional dependencies), reactive rules, prescriptive and predictive analytics.

2.2 Integrity constraints

The following integrity constraint 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:

$$\mathsf{Stock}[p] = v \rightarrow \mathsf{Product}(p), \mathsf{float}(v).$$

In general, integrity constraints are expressions of the form $F \rightarrow G$ (note the use of a rightward arrow for constraints instead of a leftward arrow as for derivation rules), where F and G are formulas. Whereas derivation rules define views, integrity constraints specify the set of legal database states. A possible change to the above constraint is to limit the Stock domain to integer instead of float:

$$\mathsf{Stock}[p] = v \rightarrow \mathsf{Product}(p), \mathsf{int}(v).$$

This change can impact the evaluation of the program, especially if the predicate Stock is subject to mathematical optimization, as explained next.

2.3 Mathematical optimization

LogiQL has constructs to support mathematical optimization. A predicate $\mathsf{R}[x_1, \ldots, x_n] = y$ 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 $\mathsf{R}[\,] = y$ can be declared to be an objective function that should be minimized or maximized.

Suppose there is a predicate totalProfit defined using the predicate Stock. Assume we would like to automatically compute stock amounts so as to maximize profit. This can be expressed in LogiQL as follows:

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

The first line is shorthand for a 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., [14, 3]. LogicBlox grounds the problem instance via automatic synthesis of another LogiQL program that translates the constraints over variable predicates into a representation that can be consumed by the solver [4].

If the application code is changed such that the predicate Stock is now defined to be a mapping from products to integers, then LogicBlox detects the change and reformulates the problem so that a different solver is invoked, one that supports Mixed Integer Programming (MIP).

This change from LP to MIP has visible performance implications. The meta-engine enables an important saving factor by avoiding to repeat the expensive grounding task. A further saving component happens at a lower level in the incremental maintenance mechanism built in the engine proper: 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 data.[7]

2.4 Logical Invariants

A further role of the meta-engine is to maintain logical invariants of the LogiQL program. We discuss one such invariant that is necessary for handling correctly data updates.

Reactive rules are used in LogiQL 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. Here are two examples of reactive rules:

$$
\begin{aligned}
&+\mathsf{Sales}[\text{“Chocolate”, M\&S, 2015-01}] = 122.\\
&\hat{}\mathsf{Price}[\text{“Chocolate”, M\&S}] = 0.8 * x \leftarrow\\
&\quad \mathsf{Price@start}[\text{“Chocolate”, M\&S}] = x,\\
&\quad \mathsf{Sales@start}[\text{“Chocolate”, M\&S, 2015-01}] < 50,\\
&\quad +\mathsf{Promo}(\text{“Chocolate”, M\&S, 2015-01}).
\end{aligned}
$$

The first reactive rule inserts a new fact into the sales predicate. The second reactive rule discounts the price of chocolate at M&S if the sales in January 2015 are lower than 50 units, and chocolate is under promotion.

[7]A capability supported by most modern solvers

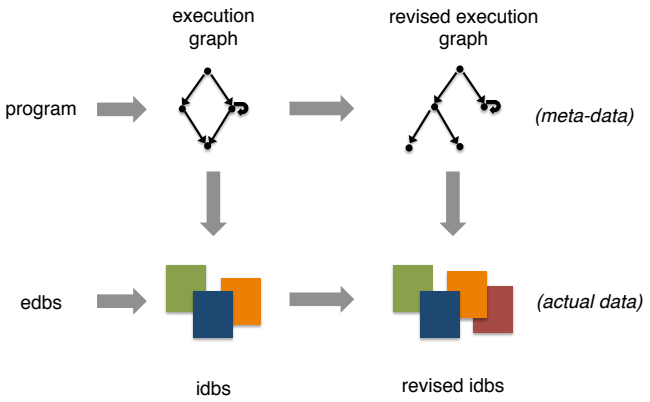

Figure 2: While the engine proper deals with maintenance of the materialized views for a given program (left half), the meta-engine maintains the program under code updates and informs the engine proper which views should be revised (right half).

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), $+\mathsf{R}$ (the set of tuples being inserted into R in the current transaction), $-\mathsf{R}$ (the set of tuples being deleted from R in the current transaction) [15]. The shorthand notation $\hat{}\mathsf{R}$ is a combination of $+\mathsf{R}$ and $-\mathsf{R}$. If R is a base predicate, the content of R after the transaction is determined by means of the following system-provided *frame rules*:

$$
\begin{aligned}
&\mathsf{R}(x_1, \ldots, x_n) \leftarrow \mathsf{R@start}(x_1, \ldots, x_n), !(\text{-}\mathsf{R}(x_1, \ldots, x_n)).\\
&\mathsf{R}(x_1, \ldots, x_n) \leftarrow +\mathsf{R}(x_1, \ldots, x_n).
\end{aligned}
$$

LogicBlox maintains the following logical invariant:

> *If +R or -R appears in the head of a rule in stage X, then we need a frame rule for R at stage X.*

The meta-engine automatically generates such frame rules whenever necessary.

3. THE LOGICBLOX META-ENGINE

The LogicBlox meta-engine supports declarative and incremental maintenance of program state under changes of the LogiQL program.

Figure 2 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 (materialized views) are the nodes and the edges between nodes represent Datalog-like derivation rules with the children being predicates used as atoms in the body of a rule whose head predicate is the parent node. The engine proper evaluates the execution graph bottom-up on the input database (also called extensional database and consisting of edb predicates) and materializes the predicates (also called intensional predicates or idbs). 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 materialized views have to be maintained as result of the program change. To achieve this, the meta-engine describes declaratively the underlying program together with

the dependencies between the program rules using meta-rules. This declarative specification of the program code can then be maintained incrementally using well-known techniques for incremental maintenance of Datalog rules.

In a broader sense, the meta-engine provides a unified machinery to incrementally maintain the program state and to orchestrate the evolution of the program code through all compilation stages. Once the text corresponding to the new code is parsed and represented as metadata, the meta-engine uses specific meta-rules to perform a wide range of tasks, including type inference; view unfolding to avoid unnecessary maintenance of intermediate predicates; frame rule generation to account for reactive rules that update the data; and construction of the internal executable representation of the code. Further compilation stages can be added as blocks of meta-rules.

The next sections describe the components of the LogicBlox meta-engine:

- The underlying object-oriented model for metadata features object immutability and persistent data structures for efficient search and update operations (Section 4).
- A declarative Datalog-like language called MetaLogiQL is used to express meta-rules on metadata representation of programs (Section 5).
- The maintenance facilities for MetaLogiQL are based on well-known algorithms for incremental maintenance of Datalog programs (Section 6).

There are significant advantages of this design over an imperative specification of the meta-engine:

- Being declarative, it shares all desirable properties of Datalog (less code, less error prone).
- It guarantees correctness (following known Datalog maintenance techniques) while providing efficient maintenance.
- The order of enforcing the logical invariants and applying code optimizations is simply captured by the dependency graph of meta-rules, no extra treatment is needed as would be the case for imperative specification of this logic in a large, monolithic code.
- The meta-engine is extensible by design. Adding new meta-rules is as simple as writing them down in MetaLogiQL syntax. It is the job of the system to register them, detect dependencies with other meta-rules, and maintain them.

These benefits come at the cost of some moderate performance overheads. The meta-rules introduce one extra level of indirection in program code maintenance as they are interpreted at runtime. It is therefore to be expected that the meta-engine is slower than a hardcoded, monolithic approach. The meta-engine has to be fast, but not nearly as fast as the engine proper, since meta-data is many orders of magnitude smaller than data. By instructing the engine proper to incrementally maintain the program result rather than to recompute it from scratch, the overall LogicBlox system gains arbitrary speedups.

An obvious question is why not use the LogicBlox engine proper to evaluate the MetaLogiQL program in lieu of the new meta-engine. In our case at least, the design choice was dictated by pragmatic concerns:

- There is an impedance mismatch between the relational format of the input data and the object-oriented representation of LogiQL programs; while shredding programs into relations is definitely possible (programs can be seen as graphs representable as a binary edge relation), such a translation comes with high performance penalty and an unnecessarily complex maintenance mechanism.
- The meta-program is written by the LogicBlox runtime team who can present the meta-rules already in an optimized form. There is thus no need for heavy optimization as done by the engine proper. The meta-program is also orders of magnitude smaller than the usual LogiQL programs backing up client applications.

4. PERSISTENT DATA STRUCTURES FOR META-DATA MANAGEMENT

The meta-engine is built on top of a C++ object management system that provides support for persistent data structures and is used internally in the LogicBlox engine to manage metadata such as program ASTs; we note that paged (and also persistent) data structures used for storing actual database tables are *not* managed by this system. We present here several aspects of this object management system that directly influenced the design of the meta-engine, in particular object immutability and branching, and purely functional implementations of versioned data structures; a full treatment is beyond the scope of this paper.

The object management system provides fast creation and manipulation of temporary objects, object persistence, versioning, and support for parallelism. The objects are heap-allocated and managed. Their references are counted by object nurseries, and offer an interface similar in spirit to Java-style objects. This offers protection against bad casts and segfaults due to null pointers (exceptions are thrown instead) and against heap corruption. The objects are transparently persisted and restored. Examples of managed objects are databases, transactions, predicates, and LogiQL programs.

A key feature of this object management system that is extensively used by the meta-engine is its *"mutable until shared"* objects that sit at a useful tradeoff point between imperative and purely functional: Such 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, persisted, or branched. This gives 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, distributed processing, and, as discussed next, metadata management in the meta-engine.

Branching an object means to create a mutable fresh (shallow) copy of it. This is an $O(1)$ operation in our object management system (recall from Section 2 that this is used to

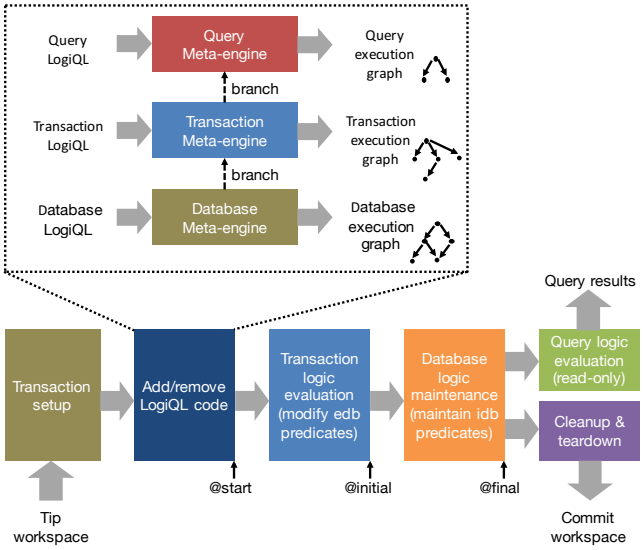

Figure 3: Chaining meta-engines for different lifetimes (database, transaction, queries) in the LogicBlox runtime system.

support concurrent access to the database via workbooks). When an object is branched, it is marked immutable together with all mutable objects transitively reachable from it. Versioned data structures are supported by coupling the branching functionality with functional-style implementations that allow for efficient search and set operations such as the difference between different versions. They represent a further key feature extensively used by the meta-engine.

We next discuss how this object management philosophy influenced the design of the meta-engine for two concrete tasks: managing permanent and transient metadata and incremental maintenance of LogiQL programs.

The metadata representing LogiQL programs can have three different *lifetimes*: database lifetime, which persists beyond ad-hoc queries and transactions, transaction lifetime, which is only relevant for a particular running transaction and discarded once the transaction is committed or discarded, and query lifetime, which is only relevant for a particular query and discarded afterwards. To manage metadata of different lifetimes in a uniform way, we chain meta-engines for different lifetimes as shown in Figure 3. A transaction-lifetime meta-engine is a branch of the underlying database-lifetime meta-engine and thus has access to and builds on a new version of its data structures for metadata. Once the transaction is completed, the instance of the transaction-lifetime meta-engine is simply discarded including its version of the metadata as it has no side-effect on the underlying database-lifetime meta-engine and its metadata. The same mechanism works for query-lifetime meta-engines, which sit on top of transaction-lifetime meta-engines. To speed up processing, transaction and query-lifetime meta-engines for blocks of code that are used repeatedly are cached. This is similar in spirit to precompiled queries.

To efficiently support incremental maintenance of metadata, the meta-engine relies on versioned data structures and in particular on persistent treaps. Treaps are randomized binary search trees that offer expected $\Theta(\log n)$ search, insertion, and deletion [24]. 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. Our treap implementation is purely functional [23], 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. Treaps offer efficient support for set intersection, union, and difference: with memoization, the cost of computing a set operation $R = op(A, B)$ is $\Theta(\delta \log n)$, where $\delta = \min(d(A, R), d(B, R))$, and $d(X, Y)$ is the edit distance measured in update operations between the sets X and Y [5].

Collections of objects such as meta-predicates, which are materialized views consisting of features of the program code such as the heads or bodies of all rules, are stored by the meta-engine as treaps or their specializations as maps, sets, or vectors. They naturally meet the requirements of live programming applications, in particular they offer support for efficient incremental maintenance. Detecting the difference between the previous and current versions of the program is implemented by taking the difference between two versions of a meta-predicate that represents the set of program rules and is implemented as a persistent treap.

5. METALOGIQL

MetaLogiQL, the language for expressing meta-rules on metadata, inherits the syntax of LogiQL. The core functionality required by the existing meta-rules is that of non-recursive Datalog with negation. Additional functionality can be encapsulated via built-in meta-functions that are exposed in meta-rules as meta-predicates with restricted access patterns, i.e., they have input and output variables and compute value bindings for output variables only when presented with bindings for the input variables. Such meta-functions are supported by C++ code to perform any required computation. Some of this additional functionality, in particular aggregates and a limited form of recursion in the form of transitive closure, surfaces in the language for two reasons: they are used often enough and we have mechanisms in place to incrementally maintain meta-rules with such language constructs more efficiently than re-evaluating them from scratch. In contrast, there is no incremental maintenance mechanism for built-in meta-functions.

Meta-rules are registered with the meta-engine in the runtime code using C++ macros. When the database system, including the meta-engine, is first started, the meta-engine collects all meta-rule declarations, builds their execution graph, and then evaluates and maintains this graph bottom-up as the database system operates.

To give a sense of the language and the scope of the meta-rules, we next discuss several meta-rules of varying structural complexity and give them as they are defined in the LogicBlox runtime. The meta-engine in the current LogicBlox distribution uses 200+ such meta-rules.

5.1 Negation

Our first meta-rule states that the (intensional) meta-predicate `lang_edb`, which consists of all extensional predicates (i.e., metadata sources) defined in the input program, is the difference between the meta-predicates `lang_ predname`, which consists of all predicates in the program, and `lang_idb`,

which consists of all intensional predicates (i.e., views). The difference operation is expressed using the exclamation (`!`) symbol.

```
BLOX_META_DECLARE_RULE(lang_edb,
 "lang_edb(name) <- lang_predname(name),!lang_idb(name).");
```

The C++ macro environment `BLOX_META_DECLARE_RULE` associates the object `lang_edb` with the meta-rule given as string. 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 set of extensional predicates (edb) in reaction to such changes.

5.2 Joins and Built-in meta-functions

As a second example, consider a conjunctive meta-rule defining the meta-predicate `user_predicates`:

```
BLOX_META_DECLARE_RULE(user_predicates,
   "  user_predicates(lifetime, predname, predicate) <-     "
   "     user_blocks(lifetime, name, block),                "
   "     block_installables(block, inst),                   "
   "     installable_directory(inst, directory),            "
   "     directory_get_index(directory, PREDICATES, index), "
   "     string_object_map(index, predname, predicate).     ");
```

This meta-rule expresses a navigation in the `block` metadata object consisting of user code, first by extracting its so-called installables, iterating over the directories of each installable, iterating over the elements in each directory and selecting only those corresponding to predicates. The meta-rule uses objects specific to the LogicBlox runtime internals. Note that in addition to user predicates, there are also predicates generated by the system for internal purposes, such as derived predicates for sampling and indexing to support query optimization.

The meta-predicate `user_blocks` is the only extensional meta-predicate and passed by the compiler to the runtime environment; all other meta-predicates are derived from it using meta-rules. It consists of all blocks of LogiQL code for database/query/transaction lifetime (recall that Section 4 discusses the concept of the lifetime of predicate.).

```
BLOX_META_DECLARE_PREDICATE(user_blocks);
```

The remaining meta-predicates in the body of the above meta-rule are built-in accessors or meta-functions that navigate inside the code block. The types of their parameters are dictated by the runtime API. They only work under restricted access patterns: given a binding for the parameter `block` of type `hBlock`, it binds the parameter `inst` to objects of type `hInstallable` as returned by the function `installables` of type `hBlock` $\rightarrow$ `hInstallable`. The `installable_directory` built-in accessor works similarly.

```
//! Given a block, return its installables
BLOX_META_DECLARE_BUILTIN_ACCESSOR(block_installables,
      hBlock, hInstallable, installables);
//! Given an installable, return its directory
BLOX_META_DECLARE_BUILTIN_ACCESSOR(installable_directory,
      hInstallable, hDirectory, directory);
```

We next give a complete specification of the built-in meta-function `directory_get_index`:

```
//! Given a directory and an object kind, return the
//  directory index
Directory::hIndex directory_get_index(hDirectory directory,
          object_kind_t kind)
{   return directory->getIndex(kind); }
BLOX_META_DECLARE_BINARY_BUILTIN_FUNCTION(directory_get_index,
      hDirectory, object_kind_t, Directory::hIndex,
      directory_get_index);
```

The built-in meta-predicate `string_object_map` iterates over the map `index` and binds its entries to tuples of predicate names and objects. The constant `PREDICATES` is of type `object_kind_t`.

5.3 Logical invariants

Meta-rules can also encode logical invariants. We discuss the invariant presented in Section 2.4 for delta predicates, i.e., a predicate `R` that occurs in a code block with a delta operator `+` or `-`. For such predicates, a specialized event-condition-action (frame) rule is created to allow trigger-like functionality to be specified in a declarative fashion. The logical invariant is that if `+R` or `-R` appear in the head of a rule in stage X, then we need a frame rule for `R` at stage X. The following meta-rule determines which predicates need frame rules for a given stage:

```
BLOX_META_DECLARE_RULE(need_frame_rule,
   "   need_frame_rule(stage, predName) <-        "
   "      user_rule(stage, _, rule),              "
   "      rule_head(rule, head),                  "
   "      head_predicate(head, predName),         "
   "      is_delta_predicate(predName).           ");
```

The meta-rule inspects the head of each user rule and checks whether it contains a delta predicate.

5.4 Constraints

The next example shows how to express constraints using meta-logic. The following meta-rule records violation of the constraint that any predicate is either intensional (idb) or extensional (edb). For each binding for query variable `x`, the built-in meta-function `edb_idb_violation` binds `msg` to a message stating that `x` is both idb and edb.

```
BLOX_META_DECLARE_RULE(constraint_fail,
   "  constraint_fail(msg) <- lang_edb(x), lang_idb(x),     "
   "                          edb_idb_violation(x, msg).    ");
```

The meta-predicate `constraint_fail` is checked for non-emptiness before a transaction is allowed to commit.

5.5 Transitive closure

The language of meta-rules does not support arbitrary recursion but only the classical transitive closure pattern (which can be statically checked at compile time).

We exemplify it next with a meta-rule that detects cycles in the LogiQL program. Assume we have already defined the meta-predicate `dependencies` that collects for all user predicates the pairs of their names and the names of all other predicates referenced in their bodies. We can then express the transitive closure of this meta-predicate as expected:

```
BLOX_META_DECLARE_RULE(predicates_tc,
   " predicates_tc(X, Y) <-                          "
   "    dependencies(X, Y);                          "
   "    dependencies(X, Z), predicates_tc(Z, Y).     ");
```

In the meta-engine, the above meta-rule pattern is detected and represented by one transitive closure node in the execution graph of the meta-rules to ease further maintenance. Note the semi-colon in the meta-rule declaration: this is used to express disjunction. The following meta-predicate consists of all user predicates in the LogiQL program that are cyclic, i.e., the transitive closure of the dependency relation of the user predicates contains a pair (X, X) for each such predicate X:

```
BLOX_META_DECLARE_RULE(cyclic,
   " predicates_cyclic(lifetime, predName, predicate) <-    "
   "    predicates_tc(predName, predName),                  "
   "    user_predicates!120(predName, predicate, lifetime). ");
```

The trailing `!120` in the name of the last meta-predicate above is due to our indexing convention. This meta-predicate is the same as `user_predicates`, but sorted using a different order of columns: the first (0) is moved to the end, the second (1) becomes first, and the third (2) becomes second. The order of columns is relevant for indexing purposes and efficient join evaluation, cf. Section 6.

5.6 Maps

In the meta-logic, we often need to execute code on each element of a collection. This is supported in meta-rules using a functional-style `map` higher-order function. For instance, the following meta-rule creates an execution unit in the execution graph of the evaluation engine for each LogiQL rule in the workspace:

```
BLOX_META_DECLARE_RULE(exec_rule_units,
   " exec_rule_units(lifetime, name, rule, unit) <-         "
   "    map<<>> workspace_rules(lifetime, name, rule),      "
   "     unit = create_rule_unit[lifetime, name, rule].     ");
```

The meta-rule applies the function `create_rule_unit` to each record in `workspace_rules` and collects the created units in the new meta-predicate `exec_rule_units`. The meta-function is declared as follows:

```
BLOX_META_DECLARE_BINARY_BUILTIN_FUNCTION(
      create_rule_unit,
      hExecutionUnit, lifetime_t, hRule,
      create_rule_unit);
```

The C++ function used by the meta-function has the following signature (its definition is skipped):

```
hExecutionUnit create_rule_unit(lifetime_t lifetime, hRule rule)
```

5.7 Group-By

MetaLogiQL also features a special *group-by* construct that is natural in a nested model, such as the object-oriented model of metadata used in the LogicBlox runtime. For instance, given a collection of tuples (name, lifetime, head_pos, rule) defining occurrences of predicate names in a rule head at position head_pos, the following meta-rule groups them by the pair of name and lifetime and, for each such distinct pair, it creates a set of pairs (head_pos, rule):

```
BLOX_META_DECLARE_RULE(grouped_predicates,
   " grouped_predicates(name, lifetime, {head_pos, rule}) <-"
   "          group-by<<>> head_predicates                  "
   "                       (name, lifetime, head_pos, rule).");
```

This functionality is useful for collecting and merging all definitions of a predicate into a single definition, as expected by the evaluation engine. We next explain this program rewriting effected by the meta-engine.

Assume the input program has rules of the form

$$\mathrm{A}(x) \leftarrow \mathrm{B}(x, \_).$$
$$\mathrm{C}(y), \mathrm{A}(x) \leftarrow \mathrm{D}(\_, y, x), \mathrm{E}(y).$$

The predicate A is defined by both rules and thus its content is a union of values x occurring in the predicates B and D at corresponding positions. The meta-engine rewrites the program such that the two definitions of the predicate A are identified, labeled as distinct definitions A_1 and A_2, and a new definition for A is added via a merge rule, which explicitly states that A is the union of A_1 and A_2:

$$\mathrm{A}_1(x) \leftarrow \mathrm{B}(x, \_).$$
$$\mathrm{C}(y), \mathrm{A}_2(x) \leftarrow \mathrm{D}(\_, y, x), \mathrm{E}(y).$$
$$\mathrm{A}(x) \leftarrow \mathrm{A}_1(x); \mathrm{A}_2(x).$$

The meta-rule `grouped_predicates` thus collects all occurrences of the predicate A by grouping the collection of all predicates ocurring in rule heads (as defined by the meta-predicate `head_predicates`) on their names and lifetimes. Subsequent meta-logic, which is not shown here, maps each element in the set to its index in the set so as to create the indices 1 and 2 in the example. The variable head_pos is necessary in case the same predicate occurrs multiple times in a rule head, for instance:

$$\mathrm{U}(x, y), \mathrm{U}(y, x) \leftarrow \mathrm{E}(x, y).$$

6. META-RULE EVALUATION

There are two flavors for meta-rule evaluation: full evaluation (or evaluation from scratch) and incremental maintenance. For both flavors, the meta-engine uses variants of known algorithms. We do not re-iterate these textbook algorithms here, but point the reader to works describing them in detail and focus on several novel aspects.

For incremental maintenance, we adapted the classical `count` algorithm on $\mathbb{Z}$-relations [26, 13, 18, 12] to the language of meta-rules introduced in Section 5. We also used a popular algorithm for efficient maintenance of transitive closure via relational calculus [10].

The `count` algorithm maintains reference counts for each derived tuple in a materialized view, since changes in the input (i.e., tuple inserts or deletes) may only lead to changes in the number of times a tuple is derived and not necessarily to its presence or absence from the result. We therefore need to consider a model based on bag semantics, where we also record the number of derivations of meta-tuples in meta-predicates. This contrasts with the set semantics used by the LogicBlox engine.

For both full evaluation and maintenance, we use a variant of the leapfrog triejoin (LFTJ) algorithm, the workhorse join algorithm used by the LogicBlox engine [25]. The extension to view maintenance is done by running LFTJ as usual, but using `count`-style delta rules derived from the meta-rules. The main distinction is that we need to consider view materialization under the bag semantics instead of the set semantics, as per our discussion above. This distinction leads to a different behavior of the algorithm, and

several optimizations relevant to the context of bag semantics. We next quickly explain the standard LFTJ and then our optimizations.

LFTJ is an improved version of the classical sort-merge join [25]. It is used to compute the materialized views, or more specifically, to enumerate the satisfying assignments of the variables in the bodies of derivation rules. A conjunctive derivation rule is a multi-way join. LFTJ decides on an order of the variables in the body of the rule and for each variable x_i in this order, it incrementally performs a multi-way intersection of the meta-predicates on the column x_i. Once an assignment for x_i is found in all meta-predicates whose atoms contain the variable x_i, the algorithm proceeds to the next variable x_{i+1} in order and seeks satisfying assignments for it within the scope of the assignment for x_i.

For instance, for the meta-rule

$$A(x, y, z) \leftarrow R(x, y), S(y, z), T(x, z),$$

LFTJ first finds a good variable order, say x, y, z. Then, it intersects $\pi_x[R(x, y)]$ with $\pi_x[T(x, z)]$. For each satisfying assignment $x = a$, it proceeds to the second variable y and intersects $\pi_y[\sigma_{x=a}(R(x, y))]$ with $\pi_y[S(y, z)]$. For each satisfying assignment $y = b$, it proceeds to the last variable z and intersects $\pi_z[\sigma_{y=b}(S(y, z))]$ with $\pi_z[\sigma_{x=a}(T(x, z))]$. For each satisfying assignment $z = c$, we obtain a result tuple (a, b, c). The algorithm then backtracks and considers further assignments of z under $x = a, y = b$. When these are exhausted, it backtracks and considers further assignments of y and so on. To iterate over the satisfying assignments of any of the variables, LFTJ *leapfrogs* (hence its name), i.e., makes increasingly larger jumps in the meta-predicates; this is different from a multi-way sort-merge join, which proceeds tuple by tuple in each meta-predicate.

LFTJ is a general-purpose algorithm which works well in practice for a large range of workloads, and has even been proved to be worst-case optimal for projection-free[8] conjunctive queries [25], under the mild assumption that the input meta-predicates are suitably indexed: for instance, clustered B+tree indices on R, S, and T that match the order of the variables would be needed to efficiently perform the intersection operation.

The meta-engine proceeds exactly as indicated above for that particular meta-rule. We next present three optimizations used by the meta-engine for meta-rule evaluation.

Firstly, it may be that there is no good join order. For instance, the previous variable order does not work for the following meta-rule

$$A(x, y, z) \leftarrow R(x, y), S(y, z), T(z, x),$$

since if we first join by x, the meta-predicate T is not sorted by x and this means there is no efficient support for the intersection of R and T on x. To overcome this, the meta-engine uses indices defined by additional meta-rules:

$$T!10(x, z) \leftarrow T(z, x).$$

This meta-rule creates a new meta-predicate whose content is the same as T but with an index first on x and then on z. The naming convention !10 states that the first (0) and second (1) columns have been swapped. For simplicity of implementation, the use of indices is manually specified by the developer when writing the rule; this is in contrast with LogicBlox runtime proper, where indices are installed and used transparently.

The meta-rules in the meta-engine are designed such that the order of occurrences of variables in the body of the meta-rule is compatible with the order of variables within each atom in the body. This order is also the LFTJ variable order. Since the variable order can have a huge impact on the performance of the evaluation, extra care is needed by the runtime developer to make sure the best possible order is used. In the meta-engine, the meta-rules are known beforehand and can thus be optimized. In contrast, in the LogicBlox engine, the user rules may not be known beforehand and good orders can only be found heuristically using sampling.

Secondly, the meta-engine's LFTJ variant needs to compute reference counts for all tuples in the result, which is potentially an expensive operation. Two optimizations have been implemented to avoid the need for tracing reference counts in special, yet common cases of meta-rules in the meta-engine. For the case of projection-free meta-rules, there is no need to trace reference counters since the counts will always be one: a satisfying assignment of all the variables can only be found once in the input, when the input meta-predicates are sets and not bags. For the case of "pure projection" meta-rules, i.e., of indices that just permute the variables and do not otherwise perform joins, we can skip LFTJ and create the head meta-predicate in one scan of the input meta-predicate. For example, this optimization is applied in case of index creation, such as the predicate $T!10$ above.

Thirdly, for meta-rules with projection, a further optimization may be performed. Consider for instance the meta-rule

$$A(x) \leftarrow B(x, y), C(x, z).$$

Since we are only interested in values for x and, for reference counting, in how many times we can derive each value for x, we simply iterate over values of x, count the y's and z's for a given x, and multiply these counts.

7. CONCLUSIONS AND FUTURE WORK

Since version 4.0.6 of May 2014, LogicBlox ships with the meta-engine. The initial version has been continuously refined since then. For live programming applications, there is a significant performance gain in the engine proper enabled by smart recomputation of materialized views as instructed by the meta-engine. This makes a stark, night-and-day difference in live programming that enabled several client applications not possible before.

The meta-engine also enables new runtime features, which were previously prohibitively awkward to implement, e.g., removal of blocks of LogiQL code while preserving logical invariants of programs. The current focus is to refactor more monolithic C++ runtime code into meta-logic since meta-rules represent much less code and are much easier to maintain. The meta-engine also helps generate only the necessary number of system LogiQL code for building indices, inlining view definitions, and properly addressing data updates via frame rules. This has not been the case before. Further code optimization is now possible via program transformation specified using meta-rules. Being more judicious with

[8] i.e, where all variables in the body also occur in the head (or more generally, where all body variables are functionally determined by the head variables).

internal rule generation makes a significant difference in performance for schemas with upwards of 100K predicates, as encountered in some client applications.

The meta-engine leverages database principles and techniques, including declarativity and efficient Datalog maintenance, as well as programming language principles, including functional data structures, high-level programming and modularity, to solve what is essentially a software engineering challenge. Its declarative nature and the high-level specification of meta-rules come with a price: When compared with compiled C++ code, there is a performance penalty for interpreting meta-rules at runtime. A promising direction for future work is to use specialized compilers to compile meta-rules to low-level optimized code [17, 19].

8. REFERENCES

[1] *Python Programming Language Website. Quotes about Python*, 2014. http://www.python.org/about/quotes/.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[4] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382, 2015.

[5] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *SPAA*, pages 16–26, 1998.

[6] B. Burg, A. Kuhn, and C. Parnin. 1st int. workshop on live programming (live). In *ICSE*, pages 1529–1530, 2013.

[7] 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*, pages 63–78, 2015.

[8] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *PVLDB*, 1(1):1153–1165, 2008.

[9] O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors. *Datalog Reloaded: Proceedings of the First International Datalog 2.0 Workshop*. Springer, 2011.

[10] G. Dong and J. Su. Incremental maintenance of recursive views using relational calculus/sql. *SIGMOD Record*, 29(1):44–51, 2000.

[11] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.

[12] T. J. Green, Z. G. Ives, and V. Tannen. Reconcilable differences. *Theory Comput. Syst.*, 49(2):460–488, 2011. Extended abstract in ICDT 2009.

[13] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.

[14] Gurobi. Gurobi optimizer reference manual, 2015. http://www.gurobi.com.

[15] T. Halpin and S. Rugaber. *LogiQL: A Query Language for Smart Databases*. CRC Press, 2014.

[16] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: An interactive tutorial. In *SIGMOD*, pages 1213–1216, 2011.

[17] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[18] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, pages 87–98, 2010.

[19] C. Koch. Abstraction without regret in database systems building: a manifesto. *IEEE Data Eng. Bull.*, 37(1):70–79, 2014.

[20] G. Lea. *Survey results: Are developers more productive in Scala?*, 2013. http://www.grahamlea.com/2013/02/survey-results-are-developers-more-productive-in-scala/.

[21] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.

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

[23] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

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

[25] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014. Also CoRR abs/1210.0481 (2012).

[26] O. Wolfson, H. M. Dewan, S. J. Stolfo, and Y. Yemini. Incremental evaluation of rules and its relationship to parallelism. In *SIGMOD*, pages 78–87, 1991.