LogicBlox, Platform and Language: a Tutorial

Todd J. Green, Molham Aref, and Grigoris Karvounarakis

LogicBlox, Inc
1349 W Peachtree St NW
Atlanta, GA 30309 USA
{todd.green|molham.aref|grigoris.karvounarakis}@logicblox.com

Abstract. The modern enterprise software stack—a collection of applications supporting bookkeeping, analytics, planning, and forecasting for enterprise data—is in danger of collapsing under its own weight. The task of building and maintaining enterprise software is tedious and laborious; applications are cumbersome for end-users; and adapting to new computing hardware and infrastructures is difficult. We believe that much of the complexity in today's architecture is accidental, rather than inherent. This tutorial provides an overview of the LogicBlox platform, a ambitious redesign of the enterprise software stack centered around a unified declarative programming model, based on an extended version of Datalog.

1 The Enterprise Hairball

Modern enterprise applications involve an enormously complex technology stack composed of many disparate systems programmed in a hodgepodge of programming languages. We refer to this stack, depicted in Figure 1, as "the enterprise hairball."

First, there is an *online transaction processing* (OLTP) layer that performs *bookkeeping* of the core business data for an enterprise. Such data could include the current product catalog, recent sales figures, current outstanding invoices, customer account balances, and so forth. This OLTP layer typically includes a relational DBMS—programmed in a combination of a query language (SQL), a stored procedure language (like PL/SQL or TSQL), and a batch programming language like Pro*C—an application server, such as Oracle WebLogic [35], IBM WebSphere [36], or SAP NetWeaver [26]—programmed in an object-oriented language like Java, C#, or ABAP—and a web browser front-end, programmed using HTML and Javascript.

In order to track the performance of the enterprise over time, a second *business intelligence* (BI) layer typically holds five to ten years of historical information that was originally recorded in the OLTP layer and performs read-only analyses on this information. This layer typically includes another DBMS (or, more commonly, a BI variant like Teradata [33] or IBM Netezza [25]) along with a BI application server such as Microstrategy [23], SAP BusinessObjects [5], or IBM Cognos [7], programmed using a vendor-specific declarative language.

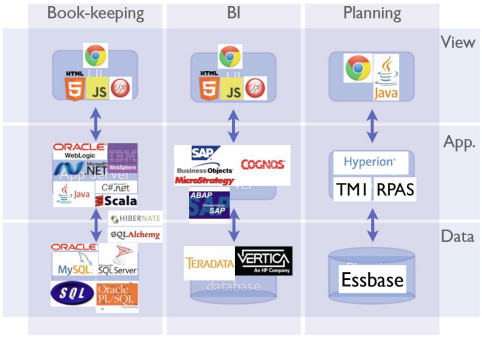


Fig. 1: Enterprise software components and technology stack example.

Data is moved from the transaction layer to the BI layer via so-called *extract-transform-load* (ETL) tools that come with their own tool-specific programming language.

Finally, in order to plan the future actions of an enterprise, there is a *planning* layer, which supports a class of read-write use cases for which the BI layer is unsuitable. This layer typically includes a planning application server, like Oracle Hyperion [13] or IBM Cognos TM1 [34], that are programmed using a vendor-specific declarative language or a standard language like MDX [21], and spreadsheets like Microsoft Excel that are programmed in a vendor-specific formula language (e.g. `A1 = B17 - D12`) and optionally a scripting language like VBA. In order to enhance or automate decisions made in the planning layer, statistical predictive models are often prototyped using modeling tools like SAS, Matlab, SPSS, or R, and then rewritten for production in C++ or Java so they can be embedded in the OLTP or OLAP layers.

In summary, enterprise software developed according to the hairball model must be programmed using a dozen or so different programming languages, running in almost as many system-scale components. Some of the languages are imperative (both object-oriented and not) and some are declarative (every possible flavor). Most of the languages used are vendor-specific and tied to the component in which they run (e.g. ABAP, Excel, R, OPL, etc.). Even the languages that are based on open standards are not easily ported from one component to another because of significant variations in performance or because of vendor specific extensions (e.g., the same SQL query on Oracle will perform very differently on DB2). Recent innovations in infrastructure technology for supporting much larger numbers of users (Web applications) and big data (predictive analytics), including NoSQL [28], NewSQL [27] and analytic databases, have introduced even more components into the technology stack described above, and have not helped reduce its overall complexity.

This complexity makes enterprise software hard to build, hard to implement, and hard to change. Simple changes—like extending a product identifier by 3 characters or an employee identifier by 1 digit—often necessitate modifications

to most of the different components in the stack, requiring thousands of days of person effort and costing many millions of dollars. An extreme example of the problems caused by this accidental complexity occurred in the lead-up to the year 2000, where the simple problem of adding two digits to the year field (the Y2K problem) cost humanity over $300 billion dollars [24].

Moreover, as a result of the time required for such changes, individuals within an enterprise often resort to ad hoc, error-prone methods to perform the calculations needed in order to make timely business decisions. For example, they often roll their own extensions using spreadsheets, where they incompletely or incorrectly implement such calculations based on copies of the data that is not kept in sync with the OLTP database and thus may no longer accurately reflect the state of their enterprise.

2 LogicBlox

To address these problems, the LogicBlox platform follows a different approach, based on ruthless simplification and consolidation. Its main goal is to unify the programming model for enterprise software development that combine transactions with analytics, by using a single expressive, *declarative* language amenable to efficient evaluation schemes, automatic parallelizations, and transactional semantics. To achieve this goal, it employs *DatalogLB*, a strongly-typed, extended form of Datalog [17,1] that is expressive enough to allow coding of entire enterprise applications (including business logic, workflows, user interface, statistical modeling, and optimization tasks).

Datalog has, historically, been used as a toy language not intended for practical applications, so the choice of Datalog as a unifying language for enterprise application development may be a bit surprising. One reason for its selection for LogicBlox was that, in our experience, Datalog programs are easier to write and understand by the target users of such systems (i.e., business consultants, not Computer Science researchers), compared to languages such as Haskell [12] or Prolog [17]. Moreover, with Datalog we are able to draw on a rich literature for automatic optimizations and incremental evaluation strategies; the latter is of paramount importance for enterprise applications, where small changes to the input of a program are common, and need to be handled at interactive speeds. Changes in an Excel-like spreadsheet application, for instance, need to be reflected immediately.

Today, the LogicBlox platform has matured to the point that it is being used daily in mission-critical applications in some of the largest enterprises in the world. In the tutorial, we present an overview of the language and platform, covering standard Datalog features but emphasizing extensions to support general-purpose programming and the development of various OLTP, BI and planning components of enterprise applications. We also highlight some of the engineering challenges in implementing our platform, which must support mixed transactional and analytical workloads, high data volumes, and high numbers

of concurrent users, running applications which are executed—conceptually, at least—entirely within the database system.

3 The DatalogLB Language

The bulk of the tutorial will focus on DatalogLB, the *lingua franca* of the LogicBlox platform. We sketch some highlights of the language in this section.

Rules. DatalogLB rules are specified using a `<-` notation (instead of the traditional "`:-`"), as in the example below:

```
person(x) <- father(x,y).
person(x) <- mother(x,y).
grandfather(x,z) <- father(x,y), father(y,z) ; father(x,y), mother(y,z).
mother(x) <- parent(x,y), !father(x).
```

In this example, `;` indicates disjunction while `!` is used for negation[1]. Predicate and variable names may use lower/upper case freely. The first two rules copy data from the `father` and `mother` predicates into `person`. The third rule computes the `grandfather` predicate, essentially as the union of two conjunctive queries. Finally, the fourth rule specifies that all parents that are not fathers are mothers, with negation interpreted under the stratified semantics.

Entity types and constraints. The main building-blocks of the DatalogLB type system are *entities*, i.e., specially declared unary predicates corresponding to some concrete object or abstract concept. The DatalogLB type system also includes various primitive types (e.g., numeric types, strings etc). For example, the following DatalogLB program declares (using a `->` notation) that `person` is an entity:

```
person(x) -> .
```

Entities can have various properties, expressed through predicates with the corresponding entity as the type of some argument, e.g.,:

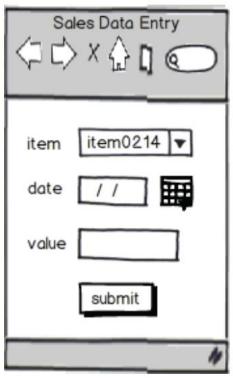
```
ssn[x] = y -> person(x), int[32](y).
name[x] = n -> person(x), string(n).
```

The first declaration says that `ssn` is a functional predicate mapping `person` entities to integer-valued Social Security Numbers, while the second maps `person` entities to string names.

Entities can be arranged in subtyping hierarchies, e.g., the following example declares that `male` is a subtype of `person`:

```
male(x) -> person(x).
```

[1] Not to be confused with the Prolog cut operator



(a)

```
sales_entry_form(f) -> form(f).

form_title[f] = "Sales Data Entry"
   <- sales_entry_form(f).

component[f] = d, dropdown(d), label[d] = "item"
   <- sales_entry_form(f).

submit_button[f] = b, label[b] = "submit"
   <- sales_entry_form(f).
```

(b)

Fig. 2: A UIBlox form and an excerpt from its specification

As expected, subtypes inherit the properties of their supertypes and can be used wherever instances of their supertypes are allowed by the type system. For example, according to the declarations above, a `male` also has an `ssn` and a `name`.

One can also use the `->` notation to specify runtime *integrity constraints*, such as that every `parent` relationship is also either a `father` or `mother` relationship, but not both:

```
parent(x,y) -> father(x,y), !mother(x,y) ; mother(x,y), !father(x,y).
```

As another example, the `[]` notation used earlier for the predicate `ssn` is just syntactic sugar for the following type declaration and integrity constraint:

```
ssn(x,y) -> person(x), int[32](y).
ssn(x,y), ssn(x,z) -> y = z.
```

Updates and events. The needs of interactive applications motivate procedural features in DatalogLB (inspired by previous work on Datalog with updates [2] and states [16]). For instance, LogicBlox provides a framework for user interface (UI) programming that allows the implementation of UIs over stored data through DatalogLB rules. Apart from being able to populate the UI based on results of DatalogLB programs, UI events are also handled through DatalogLB rules that are executed in response to those events.

For example, consider a simple application in which managers are allowed to use the form in Figure 2a to modify sales data for planning scenarios. This form, including the title of the page, the values in the drop-down menu and the text on the "submit" button, are generated by the DatalogLB rules shown in Figure 2b.

The selection of values for particular items from the drop-down menu, specifying a UI view, also corresponds to a database view:

```
dropdown_values(d,i)
   <- component[f] = d, sales_entry_form_user(f,u), modifiable_by(i,u).
```

UI events, such as when the submit button in Figure 2a is pushed, are represented as predicates, and one can write rules—such as the one below—that are executed when these events happen:

```
^sales[p,d,s] = v
   <- +button_clicked(f,s),
      sales_entry_form_user(f,u), dropdown_selected[f] = p,
      date_fld_value[f,_] = d, num_fld_value[f,_] = v, manager(s,u).
```

This is an example of what LogicBlox terms a *delta rule*[2], used to insert data into the edb predicate `sales`. In this body, the atom `button_clicked(f,s)` is preceded by the *insert* modifier "+", which indicates an insertion to the corresponding predicate. As a result, the rule will only be fired when the submit button is pushed and the corresponding fact is inserted in the `button_clicked` predicate. Similarly, the symbol "^" in the head is the *upsert* modifier, indicating that if the corresponding key already exists in `sales`, its value should be updated to the one produced by the rule, otherwise a new entry with this key-value pair should be inserted.

Constructors. DatalogLB allows the invention of new values during program execution through the use of *constructors* (aka *Skolem functions* [9]) in the heads of rules. DatalogLB programs using recursion through constructors are not guaranteed to terminate on all inputs. For this reason, the DatalogLB compiler implements a safety check that exploits the connection between Datalog evaluation and the chase procedure [22], and warns if termination cannot be guaranteed. (The same safety check is used for programs using recursion through arithmetic.)

Programming in the large. Enterprise applications written in DatalogLB can contain tens of thousands of predicates and rules. To support such large-scale projects, DatalogLB also supports organization of programs into *modules* and reusable *libraries*.

Second-order existential quantifiers. Combinatorial optimization problems arise in many enterprise applications. To support this class of problems, DatalogLB allows predicates to be marked as existentially quantified, subject to specified constraints, using the usual facilities of the language as a "syntax skin" for an underlying solver. For instance, the following is a fragment of a program that solves Sudoku puzzles via linear programming:

```
X[i,j,z,t,k]=v -> index(v), index(i), index(j), index(z),
   index(t), number(k), v >= 0, v <= 1.
index(i), index(j), index(z), index(t), number(k) -> X[i,j,z,t,k]= _.
lang:solver:variable(`X).
Obj[] = v -> float[64](v).
lang:solver:minimal(`Obj).
Obj[] += X[x,y,z,t,k] * f[x,y,k].
```

[2] not to be confused with the delta rules transformation used in semi-naive evaluation

In the example, `X` is the existentially-quantified predicate predicate, and `Obj` is the objective function for the solver. (The last rule uses an aggregate syntax, `+=`, inspired by the Dyna project [8].)

Provenance. DatalogLB includes an option to record *provenance information* [10] during program evaluation and query [14] it afterwards, to facilitate debugging.

BloxAnalysis. The BloxAnalysis feature of the platform allows one to import DatalogLB programs as data in appropriate predicates in a LogicBlox workspace. As a result, one can use DatalogLB programs to perform static analysis of DatalogLB programs, as well as to rewrite them for optimization purposes.

4 Academic Collaborations

We will close the tutorial by highlighting successful collaborations with academics using LogicBlox as a motivating setting and research vehicle. These collaborations have resulted in publications in diverse areas including declarative networking [18], program analysis [4,3], distributed query evaluation [37], software engineering and testing [31,20,19], data modeling [11] constraint handling rules [30,29,6], magic sets transformations [32], and debugging Datalog programs [15].

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci.*, 43(1), 1991.
3. M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: better together. In *ISSTA*, 2009.
4. M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.
5. `http://www.sap.com/solutions/analytics/business-intelligence`.
6. D. Campagna, B. Sarna-Starosta, and T. Schrijvers. Approximating constraint propagation in datalog. In *CICLOPS*, 2011.
7. `http://www.ibm.com/software/analytics/cognos`.
8. J. Eisner and N. Filardo. Dyna: Extending Datalog for modern AI. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 181–220. Springer, 2011.
9. H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1st edition, 1972.
10. T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.
11. T. A. Halpin. Structural aspects of data modeling languages. In *BM-MDS/EMMSAD'11*, 2011.

12. P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 27(5):1–164, May 1992.
13. `http://www.oracle.com/hyperion`.
14. G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying Data Provenance. In *SIGMOD*, 2010.
15. S. Köhler, B. Ludäscher, and Y. Smaragdakis. Declarative datalog debugging for mere mortals. In *Datalog 2.0*. 2012.
16. B. Ludäscher. *Integration of Active and Deductive Database Rules*, volume 45 of *DISDBIS*. Infix Verlag, St. Augustin, Germany, 1998.
17. D. Maier and D. S. Warren. *Computing With Logic: Logic Programming With Prolog*. Addison-Wesley, 1988.
18. W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. Secureblox: customizable secure distributed data processing. In *SIGMOD*, 2010.
19. M. J. McGill, L. K. Dillon, and R. E. K. Stirewalt. Scalable analysis of conceptual data models. In *ISSTA*, 2011.
20. M. J. McGill, R. E. K. Stirewalt, and L. K. Dillon. Automated test input generation for software that consumes orm models. In *OTM*, 2009.
21. `http://www.xmla.org`.
22. M. Meier, M. Schmidt, and G. Lausen. On chase termination beyond stratification. *PVLDB*, 2(1):970–981, 2009.
23. `http://www.microstrategy.com`.
24. R. L. Mitchell. Y2K: The good, the bad and the crazy. *ComputerWorld*, December 2009.
25. `http://www.ibm.com/software/data/netezza`.
26. `http://www.sap.com/platform/netweaver/index.epx`.
27. `https://www.451research.com/report-short?entityId=66963`.
28. `http://nosql-database.org/`.
29. B. Sarna-Starosta and T. Schrijvers. Transformation-based indexing techniques for constraint handling rules. In *CHR*, 2008.
30. B. Sarna-Starosta, D. Zook, E. Pasalic, and M. Aref. Relating Constraint Handling Rules to datalog. In *CHR*, 2008.
31. R. E. K. Stirewalt, S. Rugaber, H.-Y. Hsu, and D. Zook. Experience report: Using tools and domain expertise to remediate architectural violations in the logicblox software base. In *ICSE*, 2009.
32. K. T. Tekle and Y. A. Liu. More efficient datalog queries: subsumptive tabling beats magic sets. In *SIGMOD*, 2011.
33. `http://www.teradata.com`.
34. `http://www.ibm.com/software/analytics/cognos/products/tm1`.
35. `http://www.oracle.com/technetwork/middleware/weblogic`.
36. `http://www.ibm.com/software/webservers/appserv/was`.
37. D. Zinn, T. J. Green, and B. Ludäscher. Win-move is coordination-free (sometimes). In *ICDT*. 2012.