Differential Datalog

Leonid Ryzhyk and Mihai Budiu

VMware Research

Abstract. Many real-world applications based on deductive databases require incrementally updating output relations (tables) in response to changes to input relations. To make such applications easier to implement we have created Differential Datalog (DDlog), a dialect of Datalog that automates incremental computation. A DDlog programmer writes traditional, non-incremental Datalog programs. However, the execution model of DDlog is *fully incremental*: at runtime DDlog programs receive streams of changes to the input relations (insertions or deletions) and produce streams of corresponding changes to derived relations. The DDlog compiler translates DDlog programs to Differential Dataflow (DD) [17] programs; DD provides an incremental execution engine supporting all the relational operators, including fixed-point.
The DDlog language is targeted for system builders. In consequence, the language emphasizes usability, by providing a rich type system, a powerful expression language, a module system, including string manipulation, arithmetic, and integration with C, Rust, and Java. The code is open-source, available using an MIT permissive license [1].

1 Introduction

Motivation. Many real-world applications must update their output in response to input changes. Consider, for example, a cluster management system such as Kubernetes [11], that configures cluster nodes to execute a user-defined workload. As the workload changes, e.g., container instances are added or removed from the system, the configuration must change accordingly. In a large cluster computing configuration from scratch is prohibitively expensive. Instead, modern cluster management systems, including Kubernetes, apply changes incrementally, only updating state effected by the change.

As another example, program analysis frameworks like Doop [5] evaluate a set of rules defined over the abstract syntax tree of the program. Such an analyzer can be integrated into an IDE to alert the developer as soon as a potential bug is introduced in the program. This requires re-evaluating the rules after every few keystrokes. In order to achieve interactive performance when working with very large code bases, the re-evaluation must occur incrementally, preserving as much as possible intermediate results computed at earlier iterations.

Incremental algorithms tend to be significantly more complex than their non-incremental versions. An incremental algorithm must propagate input changes to the output via all intermediate computation steps. This, in turn, requires (1) maintaining intermediate computation results for each step, and (2) implementing an incremental version of each operation, which, given an update to its input, computes an update to its output. Incremental computations that operate on relational state are ubiquitous throughout systems management software stacks. The complexity of the incremental algorithms greatly impacts the development cost, feature velocity, maintainability, and performance of the control systems.

We argue that, instead of dealing with the complexity of incremental computation on a case-by-case basis, developers should embrace programming tools that solve the problem once and for all. In this paper we present one such tool — Differential Datalog (DDlog) — a programming language that automates incremental computation. A DDlog programmer only has to write a Datalog specification for the original (non-incremental) problem. From this description the DDlog compiler generates an efficient incremental implementation.

Overview. DDlog is a *bottom-up*, *incremental*, *in-memory*, *typed* Datalog engine for building *embedded* deductive databases.

Bottom-up: DDlog starts from a set of ground facts (provided by the user) and computes all possible derived facts by following Datalog rules, in a bottom-up fashion. (In contrast, top-down engines are optimized to answer individual user queries without computing all possible facts ahead of time.)

Incremental: whenever presented with changes to the ground facts, DDlog only performs the minimum computation necessary to compute all changes in the derived facts. This has significant performance benefits, and only produces output of minimum size, also reducing communication requirements. DDlog evaluation is *always incremental*; non-incremental (traditional) evaluation can be implemented as a special case, starting from empty relations.

In-memory: DDlog stores and processes data in memory[1]. At the moment, DDlog keeps all the data in the memory of a single machine[2].

Typed: Pure Datalog does not have concepts like data types, arithmetic, strings or functions. To facilitate writing of safe, maintainable, and concise code, DDlog extends Datalog with:

- A powerful type system, including Booleans, unlimited precision integers, bit-vectors, strings, tuples, and Haskell-style tagged unions.
- Standard integer and bit-vector arithmetic.
- A simple functional language containing functions that allows expressing many computations over these data-types in DDlog without resorting to external functions.
- String operations, including string concatenation and interpolation.
- The ability to store and manipulate sets, vectors, and maps as first-class values in relations, including performing aggregations.

[1] In a typical use case, a DDlog program is used in conjunction with a persistent database; database records are fed to DDlog as inputs and the derived facts computed by DDlog are written back to the database; DDlog does not include a storage engine.

[2] The core engine of DDlog is differential dataflow [14], which supports distributed computation over partitioned data; we may add this capability in the future.

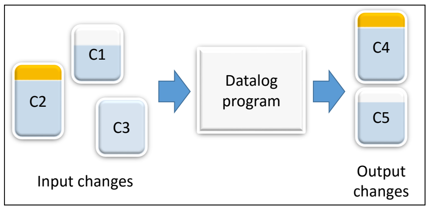

Fig. 1. Incremental evaluation of a Datalog program.

Embedded: while DDlog programs can be run interactively via a command line interface, the primary use case is to run DDlog in the same address space with an application that requires deductive database functionality. A DDlog program is compiled into a Rust library that can be linked against a Rust, C/C++ or Java program (bindings for other languages can be easily added).

DDlog is an open-source project, hosted on github [1] using an MIT-license.

2 Differential Datalog (DDlog)

A DDlog program operates on typed relations. The programmer defines a set of rules to compute a set of output relations based on input relations (Figure 1). Rules are evaluated incrementally: given a set of *changes* to the input relations (insertions or deletions), DDlog produces a set of changes to the output relations (expressed also as insertions or deletions).

Here we give a brief overview of the language; the DDlog language reference [22] and tutorial [23] provide a detailed presentation of language features.

2.1 Type system.

DDlog is a statically-checked, strongly-typed language; users specify types for relations, variables, functions, but often DDlog can infer types from the context. The type system is inspired by Haskell, and supports a rich set of types. Base types include Booleans, bit-strings (e.g., `bit<32>`), infinite-precision integers (`bigint`), and UTF-8 strings. Derived types are tuples, structures, and tagged unions (which generalize enumerated types). We currently do not allow defining recursive types like lists or trees; however DDlog contains three built-in collection types: maps, sets, and arrays (described in Section 2.4). Figure 2 shows several type declarations.

Generic types are supported; type variables are syntactically distinguished by a tick: `'A`. The language contains a built-in reference type `Ref<'T>`. Unlike other languages, two references are equal if the objects referred are equal; thus references do not alter the nature of Datalog. References can be used to reduce memory consumption when complex objects are stored in multiple relations.

```
// A tagged-union type
typedef IPAddress = IPv4Address{ipv4addr: bit<32>}
                  | IPv6Address{ipv6addr: bit<128>}
// A generic option type
typedef Option<'A> = None
                   | Some{value: 'A}
typedef OptionalIPAddress = Option<IPAddress>
```

Fig. 2. Type declarations in DDlog. `None`, `Some`, `IPv6Address` are *pattern constructors*.

```
input relation Edge(from: node_t, to: node_t)
                    primary key (e) e.from
input relation Exclude(node: node_t)
output relation Path(src: node_t, dst: node_t)
/* The Path relation is computed as a union of two rules */
// Rule 1: base case
Path(x, y) :- Edge(x,y).
// Rule 2: recursive step: join Path relation with Edge
// followed by an antijoin with Exclude.
Path(x, z) :- Path(x, w), Edge(w, z), not Exclude(z).
```

Fig. 3. A graph described by two relations and a rule to compute paths in the graph that exclude some nodes.

2.2 Relations and Rules.

Relations are strongly typed; the value in each column must have a statically-determined type. There are three kinds of relations in DDlog:
Input relations: the content of these relations is provided by the environment, in an incremental way.
Output relations: these are computed by the DDlog program, and the DDlog runtime will inform the environment of changes in these relations.
Intermediate relations: these are also computed by the DDlog program, but they are hidden from the environment.

Figure 3 shows three relation declarations. An input relation may declare an optional primary key — which is a set of columns that can be used to delete entries efficiently by specifying only the key.

DDlog rules are composed of standard Datalog operators: joins, antijoins, and unions, illustrated in Figure 3, as well as aggregation, and flatmap, discussed in Section 2.4. DDlog allows recursive rules with stratified negation: intuitively, a DDlog relation cannot recursively depend on its own negation.

2.3 Computations in rules.

Much of DDlog's power stems from its ability to perform complex computation inside rules. For example, the rule in Figure 4 computes an inner join of height

```
input relation Height(object_id: int, height: int)
input relation Width(object_id: int, width: int)
output relation Area(object_id: int, area: int)
// Compute the area of an object as the product of
// its height and width.
Area(oid, area) :- Height(oid, h), Width(oid, w),
                   var area = w * h.

// Alternative syntax for defining the same relation.
for o in Height // o is a tuple with fields (object_id, height)
   for o1 in Width if o1.oid == o.oid
       Area(o.oid, o1.width * o.height)
```

Fig. 4. Rule examples. The first rule uses expressions and variables — `var` introduces a *variable binding*. The second rule is equivalent with the first one, but is written using an imperative-style syntax.

and width tables on the object id column, and then computes the area of the object as the product of its height and width.

The DDlog expression language supports arithmetic, string manipulation, control flow constructs and function calls.

Local variables. Local variables are used to store intermediate results of computations. In DDlog, local variables can be introduced in three different contexts: (1) variables can be defined directly in the body of a rule, e.g., the `area` variable in Figure 4; (2) a variable can be defined in a `match` pattern, as in Figure 5; and (3) finally, a variable can be defined inside an expression, e.g., the `res` variable in Figure 6. A variable is visible within the syntactic scope where it was defined.

"Imperative" rule syntax. We have also defined an alternative syntax for rules, inspired by the FLWOR syntax of XQuery expressions [4]. The "imperative" fragment offers several statements: `skip` (does nothing), `for`, `if`, `match`, block statements (enclosed in braces), and variable definitions `var...in`. An example is shown in Figure 4. This language is essentially a language of monoid comprehensions [7], so it is easily converted to a traditional Datalog representation using a syntax-directed translation in the compiler front-end. Recursive relations cannot be expressed using this syntax.

Integers. The integer types (`bigint` and `bit<N>`) provide the standard arithmetic operations, as well as bit-wise operations, bit selection `v[15:8]`, shifting, and concatenation.

Strings All primitive types contain built-in conversions to strings, and users can implement string conversion functions for user-defined types (like Java's `toString()` method). Expressions enclosed within `${...}` in a string literal are

```
function lastByte(a: OptionalIPAddress): bit<8> = {
  match (a) {
    None -> 0,
    Some{IPv4Address{.ipv4addr = addr}} -> addr[7:0],
    Some{IPv6Address{.ipv6addr = addr}} -> addr[7:0]
  }
}
relation Host(address: OptionalIPAddress)
// Rule that performs matching on address structure
IPv6Addr(addr) :- Host(.address=Some{IPv6Address{addr}}).
```

Fig. 5. Pattern matching used in a DDlog function and in a rule.

interpolated: they are evaluated at run-time, converted to strings and substituted; this is a feature inspired by JavaScript; for example `"x+y=${x+y}"`.

Pattern matching. DDlog borrows the `match` expression from ML and Haskell; a `match` expression simultaneously performs pattern-matching against type constructors or values, and also can bind values. Figure 5 shows a `match` expression that uses a nested pattern to extract a byte from a value `a` with type `OptionalIPAddress` (this type was defined in Figure 2). For example, the last case binds the `addr` variable to the value of the `ipv6addr` field.

Pattern matching can also be used directly in the body of a rule, as in the last line from Figure 5, which extracts only IPv6 addresses from the `Host` relation and binds their value to the `addr` variable, which in turn is used in the left-hand side of the rule defining `IPv6Addr` relation.

Functions. DDlog functions encapsulate pure (side-effect-free) computations. Example functions are `lastByte` from Figure 5, and `concat` from Figure 6. Recursive functions are not supported. Users and libraries can declare prototypes of `extern` functions, which must be implemented outside of DDlog (e.g., in Rust), and linked against the DDlog program at link time. The compiler assumes that extern functions are pure.

2.4 Collections

The DDlog standard library contains three built-in generic collection types (implemented natively in Rust): `Vec<'T>`, `Set<'T>` and `Map<'K, 'V>`. Values of these types can be stored as first-class values within relations. Equality for values of these types is defined element-wise. In theory such types are not necessary, since collections within relations can be represented using separate relations. We have introduced them into the language because many practical applications have data models that contain nested collections; by supporting collection-valued columns natively in DDlog we can more easily interface with such applications,

```
// declare external function returning a vector of strings
extern function split(s: string, sep: string): Vec<string>
// DDlog function to concatenate all elements of a vector
function concat(s: Vec<string>, sep: string): string = {
  var res = "";
  for (e in s) {
    res = (if (res != "") (res + sep) else res) + e
  };
  res   // last value is function evaluation result
}

input relation Phrases(p: string)
relation Words(w: string)
// Words contains all words that appear in some phrase
Words(w) :- Phrases(p), var w = FlatMap(split(p, "␣")).

// Shortest path between each pair of points x, y
// (x, y) is the key for grouping
// min is the function used to aggregate data in each group
ShortestPath(x, y, min_cost) :- Path(x, y, cost),
                   var min_cost = Aggregate((x, y), min(cost)).
```

Fig. 6. Operations on collections: iteration, flattening, aggregation.

without the need to write glue code to convert collections back and forth into separate relations using foreign keys.

Figure 6 shows the declaration in DDlog of an external function which splits a string into substrings using a separator; this function returns a vector of strings.

`for` loops can be used to iterate over elements in collections. Figure 6 shows an implementation of the function `concat`, the inverse of `split`, which uses a loop.

The `FlatMap` operator can be used to flatten a collection into a set of DDlog records, as illustrated in the definition of relation `Words` in Figure 6.

The `Aggregate` operator can be used to evaluate the equivalent of SQL groupby-aggregate queries. The aggregate operator has two arguments: a key function, and an aggregation function. The aggregation function receives a group of records that share the same key. The `ShortestPath` relation in Figure 6 is computed using aggregation.

2.5 Module system

DDlog offers a simple module system, inspired by Haskell and Python, which allows importing definitions (types, functions, relations) from multiple files. The user can add imported definitions directly into the name space of the importing module or keep them in a separate name space to prevent name collisions. Similar to Java packages, module names are hierarchical and the module name hierarchy must match the paths on the the filesystem where modules are stored. The directive `import library.module` will load the module from file `library/module.dl`.

The DDlog *standard library* is a module containing a growing collection of useful functions and data structures: some generic functions and data-types, such as `min`, string manipulation and conversion to strings, functions to manipulate vectors, sets, maps (insertion, deletion, lookup, etc.).

3 DDlog Implementation

3.1 Compiling DDlog to Differential Dataflow

Differential Dataflow. The core execution engine of DDlog is Differential Dataflow (DD). Differential Dataflow [17] is a streaming big-data processing system which provides incremental (differential) computation. DD is an *incremental* map-reduce-like system, but supporting a wide set of relational operators, including recursion (fixed-point) and joins. Section 4.3 in [17] describes the core relational operators that are used by our compiler to implement DDlog operators. DD is described in several publications [20, 17] and has an open-source implementation with online documentation [16, 15].

Compilation. Figure 7 shows how DDlog programs are compiled. The DDlog compiler is written in Haskell. The compiler generates Rust code (as text files); the Rust code is compiled and linked with the open-source Rust version of the DD library [14]. The DD engine operates on multisets, where elements can have positive or negative cardinalities; to get a set semantics we need to apply `distinct` operators on some multisets, in particular, output collections.

The DDLog compiler performs parsing, type inference, validation, and several optimization steps. In order to compute incremental results the DD runtime has to maintain temporal indexes (indexed by logical time), containing previous versions of relations. Many of our optimizations are geared towards reducing memory consumption; for example, we attempt to share indexes between multiple collections and operators. We use reference counting for large values, but stack-based implementations for small values. The non-linear operators (like `distinct`) can be very expensive, so we try to minimize their usage. The compiler attempts reuse common prefixes of disjoint rules.

The output of the DDlog compiler is a dataflow graph, which may contain cycles (introduced by recursion). The nodes of the graph represent relations; the relations are computed by dataflow relational operators. Edges connect each operator to its input and output relations. DD natively implements the following operators: map, filter, distinct, join, antijoin, groupby, union, aggregation, and flatmap. Each operator has a highly optimized implementation, incorporating temporal indexes that track updates to each relation over time and allow efficient incremental evaluation. The DD library is responsible for executing the emitted dataflow graph across many cores by running a user-defined number of worker threads.

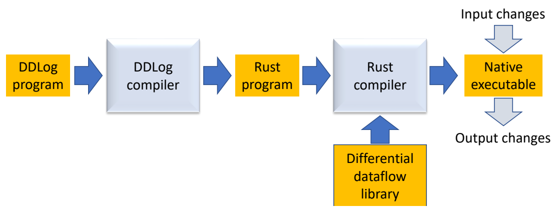

Fig. 7. DDlog compilation flow.

3.2 Interacting with DDlog programs

Transactional API. The interaction with a running DDlog program is done through a transactional API. At any time only one transaction can be outstanding. After starting a transaction the user can insert and delete any number of tuples from input relations. When attempting to commit a transaction all updates are applied atomically and changes to all output relations are produced. Users register an upcall to be notified of these changes.

The DDlog API is implemented in Rust, with bindings available for other languages, currently C and Java.

The command-line interface. For every DDlog program the compiler generates a library that exports the transactional API, which can be invoked from Rust or other languages. The compiler also generates a command-line interface program (CLI) that allows users to interact with the DDlog program directly via a command line or a script. The CLI allows users to start transactions, insert or delete tuples in input relations, commit transactions, dump the contents of relations, and get statistics about resource consumption. The CLI is also used for regression testing and debugging.

4 Applications

Controller for Virtual Networks. The most significant DDlog program we have written so far is a reimplementation of OVN [21] – a production-grade virtual network controller used to implement the network substrate for cloud management systems.

OVN translates a set of network management policies into OpenFlow rules that have to be installed on the virtual switches in the network. The logic is very complicated, comprising tens of input, output and intermediate relations.

The original program was written in C, and is not fully incremental. The DDlog implementation has about 6000 lines of code, about the same size as the original code base, but it is fully incremental. With the exception of a small number of library functions imported from C, we were able to implement the entire OVN logic in DDlog. This would not be feasible with a more traditional dialect of Datalog that does not support types and expressions.

Firewall management. We have re-implemented a proprietary network management application in DDlog. The application manages a firewall in a network of switches and virtual machines (VMs). The firewall is driven by a centralized policy; when the policy changes, the local rules have to be updated in all network devices. The core of this program is a graph reachability problem in a directed graph, which is a recursive query written in a few lines of DDlog.

We compare the performance of the DDlog implementation (blue lines) with a production-ready hand-written incremental Java program, which has been heavily optimized (pink lines). The Java program has several thousands lines of code. The graphs are synthetic network topologies; the average node degree is 2.

Figure 8 shows execution time and memory consumption of the two implementations. On the X axis we always have the graph size in nodes, and on the Y axis performance. You will note that the DDlog program performs several times better than the hand-optimized Java implementation.

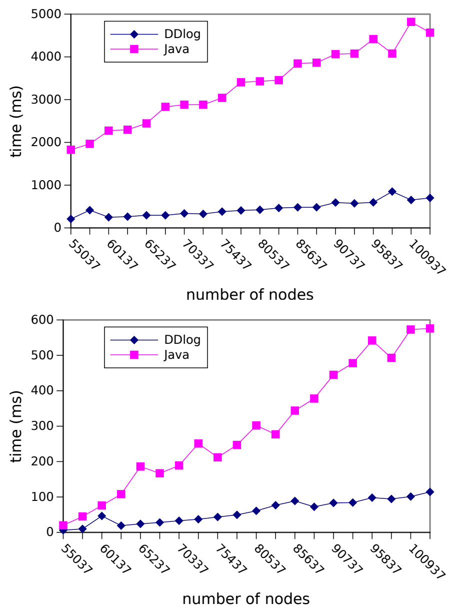

Fig. 8. DDlog performance as a function of the graph size. (1) the time taken to execute the non-incremental reachability computation (inserting all nodes and edges in a single transaction); (2) the peak memory consumption; (3) the time to insert an additional 12% edges into the graph, and (4) the time to delete 3% of the edges.

5 Related work

A survey of Datalog engines can be found in [13]. Here we focus on incremental evaluation; a survey of incremental evaluation is [9]. Notable algorithms include Delete-Rederive [10], FOIES [6]. Saha [24] provides an algorithm for tabled logic programs. The Backward-Forward algorithm [19] improves DRed under some circumstances. IncA [25, 26] is a Datalog dialect for incremental program analysis; it introduces the DRed_L algorithm. Another class of algorithms use provenance to perform incremental computation [12]. Several recent paper describe systems that use incremental evaluation for relational computation models: [2, 27].

The only other incremental Datalog engine that we are aware of is a LogiQL [8], a commercial product of LogicBlox [3]. Unfortunately there is no published data about the performance of the LogicBlox incremental engine.

DDlog is built on top of Differential Dataflow [14]; several declarative incremental query engines generalizing Datalog were built on top of Differential Dataflo [20, 17]. Some of the DDlog features were inspired by .Net LINQ [18].

6 Conclusion and future work

DDlog is a young project. The language is evolving quickly, driven by the use cases. We place paramount importance on language usability; this is why we have enhanced Datalog with many non-traditional constructs. Our goal is to reduce as much as possible the need to transition between multiple languages when writing large projects.

Our ongoing work on DDlog focuses on continuous improvement of its performance and memory utilization, as well as use-case-driven evolution of its syntax, features, and libraries. We welcome any contributions or users!

References

1. Differential Datalog. `https://github.com/ryzhyk/differential-datalog`. Retrieved December 2018.
2. Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment*, 5(10):968–979, 2012.
3. 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 *International Conf. on Management of Data (SIGMOD)*, pages 1371–1382, 2015.
4. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Ştefănescu. XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery`, 2002. Retrieved March 2019.
5. M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 243–262, 2009.
6. G. Dong and J. Su. First-order incremental evaluation of Datalog queries. In *Database Programming Languages (DBPL)*, pages 295–308. London, UK, 1994.

7. L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *ACM SIGMOD Record*, volume 24, pages 47–58. ACM, 1995.
8. T. J. Green. LogiQL: A declarative language for enterprise applications. In *Symposium on Principles of Database Systems (PODS)*, pages 59–64, 2015.
9. A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull*, 18(2):3–18, 1995.
10. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *International Conf. on Management of Data (SIGMOD)*, pages 157–166, 1993.
11. Kubernetes. Production-grade container orchestration. `https://kubernetes.io/`.
12. M. Liu, N. E. Taylor, W. Zhou, Z. G. Ives, and B. T. Loo. Recursive computation of regions and connectivity in networks. In *International Conference on Data Engineering (ICDE)*, pages 1108–1119, Shanghai, China, 29 March–2 April 2009.
13. D. Maier, K. T. Tekle, M. Kifer, and D. S. Warren. Datalog: Concepts, history, and outlook. In M. Kifer and Y. A. Liu, editors, *Declarative Logic Programming*. 2018.
14. F. McSherry. Differential Dataflow. `https://github.com/TimelyDataflow/differential-dataflow`, Retrieved January 2019.
15. F. McSherry. Differential Dataflow API reference. `https://docs.rs/differential-dataflow/0.8.0/differential_dataflow`, Retrieved March 2019.
16. F. McSherry. Differential Dataflow documentation. `https://timelydataflow.github.io/differential-dataflow`, Retrieved March 2019.
17. F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential Dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, January 2013.
18. E. Meijer, W. Schulte, and G. Bierman. Unifying tables, objects and documents. In *International Workshop on Declarative Programming in the Context of Object-Oriented Languages (DPCOOL)*, Uppsala, Sweden, August 25 2003.
19. B. Motik, Y. Nenov, R. Piro, and I. Horrocks. Incremental update of Datalog materialisation: The backward/forward algorithm. In *AAAI Conference on Artifficial Intelligence (AAAI)*, pages 1560–1569, Austin, TX, January 25–30 2015.
20. D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, Farminton, Pennsylvania, 2013.
21. OVN. Open Virtual Network architecture. `http://www.openvswitch.org/support/dist-docs/ovn-architecture.7.html`, Retrieved March 2019.
22. L. Ryzhyk and M. Budiu. Differential Datalog (DDLog) language reference. `https://github.com/ryzhyk/differential-datalog/blob/master/doc/language_reference/language_reference.md`. Retrieved December 2018.
23. L. Ryzhyk and M. Budiu. A differential Datalog (DDLog) tutorial. `https://github.com/ryzhyk/differential-datalog/blob/master/doc/tutorial/tutorial.md`. Retrieved December 2018.
24. D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *International Conference on Logic Programming (ICLP)*, pages 392–406, 2003.
25. T. Szabó, G. Bergmann, S. Erdweg, and M. Voelter. Incrementalizing lattice-based program analyses in Datalog. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Boston, MA, Oct. 2018.
26. T. Szabó, S. Erdweg, and M. Voelter. IncA: A DSL for the definition of incremental program analyses. In *International Conference on Automated Software Engineering (ASE)*, pages 320–331, 2016.
27. W. Zhao, F. Rusu, B. Dong, K. Wu, and P. Nugent. Incremental view maintenance over array data. In *ACM International Conference on Management of Data (ICMD)*, pages 139–154. ACM, 2017.