Yedalog: Exploring Knowledge at Scale

Brian Chin[1], Daniel von Dincklage[1], Vuk Ercegovac[1], Peter Hawkins[1], Mark S. Miller[1], Franz Och*[2], Christopher Olston[1], and Fernando Pereira[1]

1 **Google, Inc.**
{brianchin,danielvd,vuke,phawkins,erights,olston,pereira}@google.com
2 **Human Longevity, Inc.**
och@humanlongevity.com

Abstract

With huge progress on data processing frameworks, human programmers are frequently the bottleneck when analyzing large repositories of data. We introduce Yedalog, a declarative programming language that allows programmers to mix data-parallel pipelines and computation seamlessly in a single language. By contrast, most existing tools for data-parallel computation embed a sublanguage of data-parallel pipelines in a general-purpose language, or vice versa. Yedalog extends Datalog, incorporating not only computational features from logic programming, but also features for working with data structured as nested records. Yedalog programs can run both on a single machine, and distributed across a cluster in batch and interactive modes, allowing programmers to mix different modes of execution easily.

1998 ACM Subject Classification D.3.2 Data-flow languages, Constraint and Logic Languages

Keywords and phrases Datalog, MapReduce

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.63

1 Introduction

Organizations such as Google are busy assembling and exploiting vast and diverse repositories of "digital knowledge", such as corpora of natural language documents [18], the Knowledge Graph [17], and search engine query logs. Working with these repositories is largely semi-automatic: programmers assemble, curate, and learn from these data sets by repeatedly deploying simple algorithms over vast amounts of data.

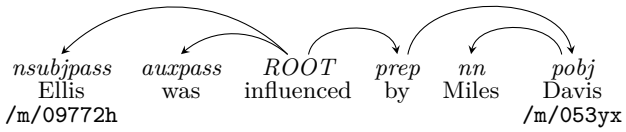
With huge progress on data processing frameworks, such as MapReduce [11] or Spark [41], the main bottleneck is the human programmer. The speed at which we can conduct experiments on our data, iterate upon those experiments, and scale our code up to larger data sets, are all fundamental factors limiting progress.

As an example of a data problem, suppose we want to find influential jazz musicians. One way we might do so is to look for sentences similar to "X was influenced by Y" in a large corpus of crawled web documents. We assume standard NLP techniques have already been applied to our corpus; for example, the text has been parsed using a dependency parser (e.g., [25]) and nouns that name entities have been resolved to unique identifiers (e.g., to Freebase MIDs [4, 22, 18]). Figure 1 shows a typical parsed, annotated sentence from our corpus.

Given a corpus containing billions of such sentences, we might find influential musicians by computing statistics, such as the frequency with which a particular musician is a dependent

* Work done at Google, now at Human Longevity, Inc.

1st Summit on Advances in Programming Languages (SNAPL'15).
Eds.: Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett; pp. 63–78
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany




```
{url: "http://en.wikipedia.org/wiki/Cold_Sweat",
 hostname: "en.wikipedia.org", size: 200, accessed: 1418500000,
 tokens:[{text: "Ellis", kind: 'NSUBJPASS, parent: 2, entity: "/m/09772h"},
        {text: "was", kind: 'AUXPASS, parent: 2},
        {text: "influenced", kind: 'ROOT},
        {text: "by", kind: 'PREP, parent: 2},
        {text: "Miles", kind: 'NN, parent: 5},
        {text: "Davis", kind: 'POBJ, parent: 3, entity: "/m/053yx"}]}
```

Figure 1 A parsed document and its representation as a Protocol Buffer, in Yedalog syntax.

of the verb "influenced". This example is typical of the kind of experiment that programmers commonly want to conduct when working with such data sets. Each experiment may or may not work out, so we want to run many experiments casually and quickly.

Solving this task has both computational aspects, namely navigating parse trees, and data-parallel aspects, namely distributing the work of processing a corpus across a cluster. Most existing tools for data-parallel computation embed a sublanguage of data-parallel pipelines in a general purpose language (e.g., LINQ [27], DryadLINQ [40], Flume [6], Spark [41]), or vice-versa (e.g., Pig [31], Dremel [28], and SQL-based approaches [16, 13]). There are few languages where both computation and data-parallelism are first class (a notable exception is Jaql [3]).

In practice, programmers use an assortment of tools, each with different strengths. A programmer might use a batch pipeline system, like Flume [6] or Pig [31], to iterate over the corpus of documents, and use Java or C++ to navigate the parse trees. If the output is large, then a programmer will make a sequence of ad hoc queries to understand the result using a distributed SQL-like system, such as Dremel [28] or Spark [41], or even scripts running locally on a workstation.

Switching among languages and tools has both cognitive and practical costs. For our example, the programmer must mix many different kinds of code (e.g., Flume pipelines, Java, SQL, and SQL UDFs), leading to complex code that is difficult to write, extend, or debug. Because each task is in a different language, it is difficult to try out a task on small data on a local machine, or to re-deploy a batch job on a high fan-in topology if the intermediate data turns out to be small enough. The programmer must duplicate code, and write boilerplate to deal with mismatches between the data models of each language and the on-disk data.

In this paper we describe Yedalog (Knowledge Logic), a declarative programming language based on Datalog. Yedalog has two primary goals:

- Make computations over semi-structured data easy and succinct (Section 3). Data problems require lots of experimentation, and smaller code allows faster iteration. A declarative language enables programmers to solve problems quickly by focusing on the high-level logic of a task instead of low-level details. Working with semi-structured data is often laborious and error prone. Yedalog's data model is based around protocol buffers [19], which combined with unification, makes it easy to read, write, match, and transform semi-structured data. Yedalog is based on Dyna-style [14] weighted deduction,

```
Documents = Load{path: "document-parse-trees"};

# Code to execute for each document:
module PerDocument{tokens: T} = {
  # Computes parent-child relationships
  Child{p} = c :- T[c] == {parent: p, ..._};

  # Computes nodes that transitively descend from "influenced"
  Descendants{t: c} :- T[p] == {text: "influenced", ..._}, c == Child{p};
  Descendants{t: c} :- Descendants{t: p}, c == Child{p};

  # Counts nodes in which each entity appears under the verb "influenced".
  Influence{mid} += 1 :- Descendants{t}, T[t] == {entity: mid, ..._};
};

# Each entity's influence from each hostname
Influential{mid, hostname} += count :-
  Documents{tokens, hostname, ..._},
  PerDocument{tokens}.Influence{mid} == count;

# Persists Influential as protocol buffer data
? Store{data: Influential, path: "influential-entities"};
```

Figure 2 A batch Yedalog job: find influential entities using parsed web documents.

not only making it easy to compute and work with statistics, but also forming a pleasing hybrid between functional and logic languages.

- Combine data-parallel pipelines and computation in a single language (Section 4). Both are necessary for working with large corpora of rich data, such as parse trees, and combining both in a single language leads to simpler code. Yedalog can run data-parallel pipelines on three different execution platforms, each useful for different tasks: (a) on a single machine over small data, (b) on an high fan-in distributed system for interactive queries that yield small answers from big data (like Dremel [28]), and (c) as pipelines of map-reduce jobs for batch tasks (using Flume [6]). Yedalog complements purely declarative programming by allowing the programmer to override Yedalog's automatic choices using physical annotations. Physical annotations help programmers scale up declarative code without having to rewrite it in a lower-level language.

We have built a prototype implementation, in use by a handful of teams at Google (Section 5). Yedalog is an ongoing research project – this paper describes our progress to date and motivates future research (Section 6).

2 A Running Example

We introduce Yedalog using a running example. Subsequent sections explore specific features in more depth: Section 3 describes a number of language features that have proved especially valuable, and Section 4 discusses data-parallelism.

Yedalog builds on a long history of related work, notably Datalog, Dyna [14] and Jaql [3]. Our goal is not to give a formal description, but instead to survey the key features and to motivate future research.

As a running example, we use the code in Figures 2 and 3 that finds influential jazz

```
1  Influential = Load{path: "influential-entities"};
2  Freebase = Load{path: "freebase-snapshot"};
3
4  # Computes IDs of jazz artists
5  @small(JazzArtist); # physical annotation
6  JazzArtist{mid} :-
7    Freebase{sub: mid, pred: "/music/artist/genre", obj: "/m/03_d0", ..._};
8
9  # A jazz artist's influence, summed across documents from Wikipedia
10 JazzInfluence{mid} += count :-
11   Influentual{hostname: "en.wikipedia.org", mid} == count,
12   JazzArtist{mid};
13
14 # Prints the ID and count of each jazz artist.
15 ? JazzInfluence{mid} == count;
```

Figure 3 A high fan-in job: find influential jazz musicians using the output of Figure 2.

musicians using a corpus of parsed web documents. We briefly walk through the example. Figure 2 loads a corpus of parse trees represented as a collection of protocol buffers (`Documents`), iterates over the corpus counting entities that are dependents of the verb "influenced" (`Influential`), and stores the result as a protocol buffer collection. Figure 3 takes as input the `Influential` data saved by the first part, joins it with a snapshot of `Freebase` [4] on `mid` to find jazz artists, and prints their mids and counts.

A Yedalog program, like Datalog, consists of *rules* and *queries*. Rules bind a value or a predicate to a variable name. For example, in Figure 2, line 17 defines a predicate named `Influential`.

Semantically, as in Datalog, a predicate is a collection of records. There are three key differences from Datalog:

- records have *named* fields. Since predicates are just bags of records, both predicate definitions and calls have *named* arguments, written `{f:x}`. There is a simple correspondence between Yedalog records and protocol buffer messages (Section 3.1).
- records can have *values*. Inspired by Dyna [14], Yedalog extends Datalog with values (Section 3.2). All predicates and expressions may have values, representing, say, statistics, weights, or probabilities.
- by default predicates are bags (multisets) of records, not sets.

There are two shorthands that we use frequently. The common case `{f:f}` where the argument and the variable have the same name can be written using the shorthand `{f}`. It is also often convenient to use a more traditional positional notation. Tuples (e.g., `(1, "foo")`) are a shorthand for records that have numbers as field names (e.g., `{0:3.1416, 1:"foo"}`). Predicates can thereby use positional notation (e.g., `Store(x, y)` as a shorthand for `Store{0: x, 1: y}`).

Core operators include unification (`==`), conjunction (`,`), disjunction (`|`), negation (`!`), if-then-else (`if` x `then` y `else` z), together with the usual arithmetic operators. The `+=` operator in the head of line 17 is an aggregator (Section 3.2). Operator `x[y]` is the list indexing operator, which looks up or scans over the elements of a list by index.

Predicates are first-class in Yedalog; the result of the `Load` predicate in Line 1 is a predicate that opens an on-disk collection of protocol buffers, binding `Documents` to a predicate that reads the data as a value. Evaluation in Yedalog is lazy, so the data will be read on-demand.

In Figure 2 line 4, `PerDocument` is a module constructor – a predicate whose value is a record representing an instance of that module. Here, the module primarily saves us from repeating the `tokens` argument. Definitions in a module's body become fields of the record; for example, the value of `PerDocument{tokens}.Influence` is a closure of `Influence` with a specific value for `tokens`.

For each document in `Documents`, predicate `Influential` instantiates the `PerDocument` module with the list of token records from that document, where the token's `parent` fields link them into parse trees. Predicate `Descendants` finds nodes that descend from the text `"influenced"` in those parse trees. `Influence` counts how many such tokens refer to any particular entity. `Influential` counts tokens for each entity (`mid`) per `hostname`.

In Figure 3, Predicate `JazzArtist` computes the mids of jazz artists in Freebase. Finally, for each influential entity that is also a jazz artist, `JazzInfluence` sums influence counts across Wikipedia documents. The query at the end starts the evaluation and displays the results.

3 Working with semi-structured data

Yedalog makes many extensions to Datalog. We focus on two that are particularly useful for working with semi-structured data. Firstly, Yedalog's data model is based on Protocol Buffers (Section 3.1); when combined with unification this gives us a convenient facility for pattern matching on structured data. Secondly, Yedalog incorporates Dyna-style weights, which is key to working with statistics over data (Section 3.2).

3.1 Data Model: Protocol Buffers and Unification

When working with data on disk or over the network, programmers write large amounts of code to convert data to and from its serialized representation. Yedalog's data model is based on Protocol Buffers [19], which Google uses as a common representation of data. The direct correspondence between the two allows us to work directly with external data in protocol buffer form, removing the need for tedious serialization code.

Yedalog *records* correspond directly with protocol buffer messages, which consist of name-value pairs. Yedalog's primitive data types correspond with the primitive protocol buffer types: `null` (representing absence), booleans, integers, floating-point numbers, and strings. Repeated fields in protocol buffers correspond to *lists* (e.g., `[1, 7]`), and protocol buffer enumerations correspond to *symbols* (e.g., `'PREP`). Records and lists may be nested (giving up the decidability of Datalog).

In Figure 2, each web document is represented as a nested protocol buffer, as shown in Figure 1. The `Documents` rule (line 1) loads a table of such data from disk as a predicate. We can work with the result like any other predicate.

3.1.1 Record Unification

A key operation when working with semi-structured data is *pattern matching*, that is, selecting records that match particular criteria and extracting data from them. Yedalog adapts the term unification of logic programming to nested records of named fields, giving a convenient way to match and build protocol buffers. Unification over records of key-value pairs dates

back to Kay's work on feature structure unification [23]; we argue that it is time that the idea was revived.

A *record pattern* is a record containing free variables that unifies with a subset of records at run time. (Yedalog does not allow non-ground terms, so record patterns are a strictly syntactic construct.) Since predicates are just bags of records, both the head of a predicate's definition and the arguments to a call to a predicate are record patterns. We can use record patterns to select records matching particular values, or to extract fields:

```
? Documents{accessed:1418500000, size, .._};
# size: 200

? Documents{hostname, url:_, tokens:_, ..rest};
# hostname: "en.wikipedia.org", rest: {size: 200, accessed: 1418500000}
```

A record pattern consists of *fields* `{f:e}`, and an optional *rest* `{..e}`. Fields match the value of a field `f` against an expression `e`. A rest is a wildcard that matches all fields not explicitly named. As with argument records, we abbreviate field patterns `{p: p}`, where the argument is a variable with the same name as the field, as `{p}`. Underbar (`_`) means "don't-care".

Record patterns may be nested:

```
Menus{cafe: "Cafe A", dish: {name: "Bread", allergens: ["wheat"]}};
Menus{cafe: "Cafe B", dish: {name: "Salted peanuts", allergens: ["peanut"]}};
Menus{cafe: "Cafe B", dish: {name: "Cake", allergens: ["peanut", "wheat"]}};

# Find dishes that don't contain peanuts.
? Menus{cafe, dish: {name, allergens, .._}, .._}, !(allergens[_] == "peanut");
# cafe: "Cafe A", name: "Bread", allergens: ["wheat"]
```

Since unification is reversible, we can also use record patterns to build records. For example, the following `Token` predicate extracts the `tokens` list from each document, extracts each token record from the list (`tokens[i]` is the list indexing operator), and builds a relation consisting of the contents of token records and their indices.

```
Token{index, ..t} :- Documents{tokens, .._}, tokens[index] == t;
? Token{index:0, text, parent, .._};
# text: "Ellis", parent: 2
```

3.2 Weights, Values, and Aggregation

Yedalog attaches *values*, such as weights, probabilities, or counts, to facts and to expressions. For example, the `Child` predicate (Figure 2, line 6) has a *value*, attached using the `=` operator, in addition to its named parameter `{p}`. Facts in `Child` are pairs, where "`p`" is the parent and the value is the child.

Yedalog's aggregation syntax, which follows Dyna [14], groups the records of predicates by a key, and combines the values using an *aggregator*. Common aggregators include `+=` (sum), `*=` (product), `Min=` (minimum), `Max=` (maximum), and `Avg=` (mean). Users can define their own aggregators.

For example, the `JazzInfluence` predicate has a value defined by the aggregator "`+=`", rather than "`=`". Instead of returning all the separate `{mid} = count` pairs as facts, aggregation groups records by the fields in the head (in this case `mid`), and combines multiple values using the `+` operator. Here, the rule sums the counts for each `mid`.

Aggregation and recursion can be combined for computations over *semirings*, giving a succinct way of solving many dynamic programming problems. For example, given an `Edge` predicate associating graph edges with weights, the `Dist` predicate below computes the shortest path between pairs of nodes.

```
Edge{s: "a", t: "b"} = 0.3;
Dist{s, t} Min= Edge{s, t};
Dist{s, t} Min= Edge{s, t:u} + Dist{s:u, t};
```

The `Min=` aggregation and the `+` operator form a semiring. The semiring structure allows Yedalog to compute shortest paths efficiently and incrementally using a variant of Datalog's semi-naive evaluation over semirings [14].

3.3 Expressions and Relational Composition

Yedalog does not just attach values to facts: expressions also have values, allowing us to program in a "functional" style. However, in the absence of an aggregator, expressions may have multiple values and need not be functions.

For example, the expression `Child{p}` may succeed many times; the successive values are the children of parent `p`. Nested expressions correspond to relational composition. The grandchildren of `x` can simply be written as `Child{p: Child{p: x}}`. We have found this easy composition of relations to be very convenient [20]; any expression can act as a generator.

An example of a multi-valued operator, is the element-of operator, written `x[y]`. The element-of operator may be used in two directions; either to look up an element of a list with a specific index, or to iterate over all elements of a list, reminiscent of XPath [8]:

```
? Documents{..d}, t == d.tokens[0].text;
# t: "Ellis"

? Documents{..d}, t = d.tokens[idx].text;
# idx: 0, t: "Ellis"
# idx: 1, t: "was"
# ...
```

Like all parameters to a predicate, values can also be inputs. This is convenient for pattern matching: for example, we can find documents whose URLs start with `"http://"`:

```
HasSchema(s) = url :- String.StartsWith(url, s);
? Documents{url: HasSchema("http://"), size, ..._};
```

4 Combining Data-Parallelism and Computation

Working with large corpora requires a mixture of distributed and local computations. The goal of Yedalog is to combine both in one language, allowing us to move seamlessly from local computation, to ad-hoc interactive analysis of data sets too large to fit on a single machine, to large-scale batch jobs.

Most existing tools for data-parallel computation embed a sublanguage of data-parallel pipelines in a general purpose language (e.g., LINQ [27], DryadLINQ [40], Flume [6], Spark [41]), or vice-versa (Pig [31], Dremel [28], and SQL-based approaches [16, 13]). Inspired by Jaql [3], Yedalog represents both computation and data-parallel pipelines in the same language.

Using a single language allows a planner visibility into both the data-parallel and computational parts of a pipeline, allowing more opportunities for optimization. On a purely pragmatic level, unifying both data-parallelism and computation means there is no syntactic overhead to combining the two, leading to shorter, more readable code.

In addition to running code on a single machine, Yedalog has two distributed execution platforms: a batch platform (Section 4.1) built on top of Flume [6], and an interactive service (Section 4.2) inspired by Dremel [28]. Each serves a different purpose, but code written for one can run on the others with minimal modification, subject to the constraints of the platform.

To run code locally, or within a distributed worker, Yedalog transforms predicates in the surface language into pipelines of pull-based generators that communicate via shared storage. Local evaluation is relatively standard, and we do not describe it in detail here, although Section 4.3 describes some of the more important aspects of the compilation process.

Our focus has been on programmer productivity and scalability over speed. The current Yedalog implementation is an interpreter, and its single-threaded performance is an order of magnitude slower than a compiled language such as C++. For many applications, programmer productivity and scalability are more important than speed. There are many standard optimizations that we could apply, such as JIT compilation, should it prove necessary.

4.1 Batch Backend

Yedalog can run programs in batch mode by converting them into Flume [6] pipelines, that in turn execute as sequences of map-reduce jobs. The batch backend is best suited for long-running jobs that produce large outputs or intermediate results.

For example, the `Influential` predicate (Figure 2, line 17) takes a corpus of parsed documents (`Documents`), and uses the `Influence` predicate in the `PerDocument` module to count entity `mid`s that appear as dependents of the verb "influenced". The predicate sums the resulting `count`, grouping the results by the `{mid, hostname}` fields. If the corpus of documents is sufficiently small, we can simply run this rule locally. However, for a large enough corpus, or if we were doing more substantial work than simply navigating within a tree, we might want to distribute the per-document work across a cluster.

There is a natural mapping from Yedalog rules to map-reduce jobs: rules correspond to map jobs, whereas aggregators correspond to reduce tasks. For example, for the `Influential` predicate, the body of a rule corresponds to a map task – for each document, do some per-document computation, yielding key-value pairs of the form (`{mid, hostname}`, `count`). The aggregator (`+=`) corresponds to a reduce task; we are to sum the counts for each key.

We leverage this insight to compile Yedalog programs into Flume pipelines. Compilation to Flume partitions relations and operators into two coarse cardinality classes: *big*, which represent data of arbitrary size, and *small*, which are small enough to fit in a single machine's memory. Operations on big relations are compiled into Flume pipeline operators, whereas small operations are run using the single-machine execution platform, either on the head node, or by a phase of the distributed pipeline.

For example, when compiling Figure 2 to Flume, the `Documents` predicate would be inferred to be big, either automatically or using a programmer-specified annotation (Section 4.3). The planner compiles the `Influential` rule by partitioning it into two parts – a big part, whose operators are translated into distributed Flume operators, and a small per-document part, which is executed using the single-machine backend:

```
InfluentialBig{mid, hostname} += count :- Documents{tokens, hostname, ..._},
                                    InfluentialSmall{tokens, mid, count};
InfluentialSmall{tokens, mid, count} :- PerDocument{tokens} == t,
                                   t.Influence{mid} == count;
```

In this particular case, `InfluentialBig` becomes a single map-reduce job, whose mapper executes the code of `InfluentialSmall` code using the single machine backend.

4.2 Interactive Backend

Although the batch backend can scale to massive datasets, we frequently want to ask sequences of short analytical queries. For example, we might want to know the average size of a document, or the frequency of a particular construction. The batch backend is not suitable because of overheads such as high start up time.

Yedalog can compile to a high fan-in distributed service, which executes programs on a tree of shared workers, modeled after Dremel [28]. Compilation to the interactive service is very similar to compiling to the batch map-reduce backend; the interactive service is limited to queries that produce small outputs and intermediate results.

For example, the `JazzInfluence` predicate in Figure 3 takes the set of `Influential` entities produced by Figure 2, and counts entities that both come from Wikipedia and are jazz artists. When executed using on the interactive backend, each leaf worker scans over a shard of the `Influential` table, accumulating a table mapping `mid`s to `count`s. Non-leaf workers sum the counts for each `mid` from each of their children, and the root worker returns the grand totals to the client in seconds.

4.3 Declarative Code, Planning, and Physical Hints

Yedalog aims to let programmers focus on the logic of their program without laboring over the mechanics of how it is to be run. Given a Yedalog program, the key challenge is *planning*, that is, choosing a feasible and efficient operational strategy for running the program. For example, the planner must choose how information flows through rules (moding), whether code should run locally or distributed, whether to materialize a predicate as a table (materialization) and if so, which indexes to create (indexing), and which algorithms to use for joins and in which order (join planning).

It is not realistic to expect that any planner will always make optimal decisions about how to run every program. Many commercial relational database management systems include a hint language to guide the query optimizer to make better choices [32, 29]. Motivated by *physical transparency* [3], rather than hiding execution details and attempting fully-automatic optimization, we use a simple, best-effort planner to handle most cases adequately, and rely on monitoring and programmer-supplied physical annotations for the rest.

Users are offered a *monitor* tool that shows their (logical) Yedalog program overlaid with color-coded execution statistics (e.g., time spent in each predicate) and *physical annotations* encoding the choices made by the planner. The user can use the execution statistics to spot bottlenecks, and add manual physical annotations if necessary (e.g., force a different join order, or request an additional index).

For example, the `@small` annotation in Figure 3 line 5 marks the `JazzArtist` predicate as small (Section 4), implying that it fits in memory. There are only a few thousand jazz artists in Freebase, but the planner cannot know that without either an annotation or a good statistical model of `Freebase`.

As another example, there are two possible ways we might compute the `JazzArtist` predicate. One option is that we can scan over `Freebase`, finding all triples with the right predicate and object. However if `Freebase` had a suitable index, then we could instead take `mid` as an input, and look up a particular `mid` to determine whether it corresponds to a jazz artist. We can use an `@mode` annotation on the call to `Freebase` to select which of the two physical strategies we intend. If we do not specify, the planner will choose one automatically.

5 Experience

The Yedalog implementation is still at an early stage, but is used by a handful of teams at Google, for tasks ranging from ad hoc analysis of linguistic data, to production pipelines that use Yedalog to identify malware. For example:

- Teams working with natural language are using Yedalog to solve a range of different tasks, along the lines of the running example in this paper.
- Several teams are using Yedalog in pipelines that find and classify malware. The Yedalog program takes as input protocol buffers containing signals that describe, for example, a web page. Yedalog rules are used to match combinations of input signals and compute a numerical score for each candidate. Yedalog is used online in production to classify new candidates as they arrive. Yedalog is also used offline to test the efficacy of new rules by applying them to a corpus of historical data, using Yedalog's interactive and batch distributed backends. (Other authors have explored domain-specific languages and libraries for similar tasks [5, 26].)
- A team is using Yedalog to write data integration pipelines. Data about the same entities is currently spread across several different data sets. The team is using batch Yedalog pipelines to merge and transform data about an entity into a single data set. Yedalog is a good fit for batch pipelines that transform data in protocol buffer form.

5.1 Case Study

We conducted an informal Yedalog case study using a research task that classifies instances of certain linguistic constructions in a corpus of text. An existing implementation used a combination of C++, MapReduce and domain-specific libraries to find examples of the construction in the corpus, and used Dremel interactively to compute statistics about the relative frequency of particular examples.

An expert Yedalog programmer with limited domain knowledge reimplemented the task in Yedalog in a few days. The Yedalog programmer primarily worked from a specification, but consulted the C++ code to verify several details. The new implementation used 70% fewer lines of code than the original. The two implementations had similar scaling properties since both compiled to similar pipelines of MapReduce jobs. The Yedalog interpreter's single-threaded performance, while an order of magnitude lower than the C++ implementation, was acceptable for the task because of parallelism.

Initially, there were some puzzling discrepancies in the outputs of the two implementations, which turned out to have a number of different causes:

- part of the specification was miscommunicated to the Yedalog programmer,
- there were small logic bugs in the original C++ implementation; for example, the C++ implementation incorrectly handled matches that were supersets of other matches, and
- the original C++ implementation contained extra heuristics to work around a memory limitation in a piece of infrastructure that weren't present in the specification.

```
# The random jump probability.
Alpha = 0.15;

# Number of outlinks for every node.
Outlinks(j) += 1.0 :- Edge(j, _i);

# The (non-normalized) PageRank algorithm.
PageRank(i) += Alpha :- Node(i);
PageRank(i) += PageRank(j) * (1.0 - Alpha) / Outlinks(j) :- Edge(j, i);
```

Figure 4 PageRank in Yedalog. Assumes the existence of `Node` and `Edge` predicates that describe a directed graph.

Since the Yedalog code was much shorter, it was easier to check for correctness. After identifying and fixing these issues, the two systems produced identical output.

6 Research Challenges

6.1 Dynamic programming, Semirings, Recursion

Eisner et al.'s Dyna proposal [14] was one of the papers that inspired us to started working on Yedalog. Dyna demonstrates that a Datalog over weighted semirings can express a wide range of AI tasks elegantly, ranging from natural language parsing to constraint satisfaction. For example, Figure 4 expresses the classic PageRank algorithm [33] in just a few lines of Yedalog code.

Declaratively, this states the set of simultaneous equations whose solution is the rank of the web's pages. Operationally, this also expresses how to compute the exact answer recursively, by propagating updates along the graph until the values reach numeric convergence. Many other algorithms, such as CKY parsing and machine translation decoding can be expressed in equally elegant fashion.

However, computing the exact PageRank of the web would be both expensive and pointless. Practical approaches require significant operational control. Often we don't want precise solutions, but heuristic approximations that can be computed cheaply.

- We might stop early, well before actual convergence.
- We might opportunistically drop some updates – by unsynchronized writes to shared memory [35] or by ignoring stragglers [28] – in order to process other updates faster.
- We often need precise control over how search is done. For example, in a natural language parser, or in a machine translation decoder, we frequently use a beam search to find an approximate solution quickly, instead of an exact dynamic programming search.

Although we have expressed beam search "declaratively" in Yedalog, we found we had sacrificed the brevity and clarity benefits promised by declarative programming. A challenge for future research is to preserve the clear declarative statement of the result to be approximated, while separately stating the means of approximating it and how much distortion is implied [36].

Neither of our distributed backends are optimized for "big" recursive queries over "big" data sets. The approaches explained by [1, 37] are particularly promising.

```
# Predicate that translates English to French by making an RPC.
ToFrench(english) = french :-
  RPC.Call{
    host: "... elided ...", service: "TranslationService", method: "Translate",
    input: {text: english, src_lang: 'ENGLISH, tgt_lang: 'FRENCH},
    output: {translated_text: french, ..._}};

EnglishCorpus("Shall I compare thee to a summer's day?");
EnglishCorpus("Thou art more lovely and more temperate:");

FrenchCorpus(french) :- Corpus(english), french == ToFrench(english);

? FrenchCorpus(french);
# french: "Dois-je te comparer à un jour d'été?"
# french: "Tu es plus belle et plus tempéré."
```

Figure 5 Interacting with services from Yedalog.

6.2 Interacting with Services

We often want to interact not just with tables of data on disk, but also with services running within a datacenter. Datacenter services are often provided using a remote procedure call (RPC) abstraction, via frameworks such as Thrift [15] or gRPC [12].

Since predicates are the common abstraction of both data and computation in a logic language, we represent query-only service APIs in Yedalog as predicates. Google's gRPC library [12] is based around protocol buffers; in Yedalog a remote procedure call is modeled as a predicate that inputs and outputs Yedalog records. Figure 5 shows an example of Yedalog code that uses Google Translate to translate a corpus of documents.

While it is convenient to represent services as predicates in Yedalog, it also poses a number of challenges. When interacting with remote services, it is important to hide latency via parallelism or batching. For example, if each translation call is synchronous and has, say, 10ms of network latency overhead, translating a corpus of a thousand sentences will take at least 10 seconds. If we made all the requests simultaneously, the latency overhead would be only 10ms.

It should be the obligation of the programming language to hide latency automatically. Yedalog has preliminary support for hiding RPC latency via batching, and we intend to explore this further in future work.

6.3 Incremental and Streaming Computation

A natural extension of our work would be to use Yedalog to maintain views of data incrementally and efficiently. For example, in the example of Section 2, we should be able to maintain the output statistics efficiently in the presence of additions or removals of documents from the input corpus.

There has been an immense body of work on maintaining incremental views in the database community (e.g., [21], [41], [30], [37], and many others). We believe that Datalog variants with weights over ring and semiring structures are well suited to the automatic and efficient computations of incremental views.

7 Related Work

Yedalog builds upon and incorporates many ideas from the research literature.

7.1 Datalog

Dyna's pioneering work on the elegant expression of natural language processing and machine learning algorithms [14] helped inspire our project. Yedalog uses a number of ideas from Dyna, including attaching values to facts, the syntax for aggregators, the use of aggregators for weighted deduction from noisy data, and first-class modules (Dynabases).

Yedalog builds on a long history of work on Datalog and deductive databases. Like Coral [34], Yedalog mixes top-down and bottom-up evaluation strategies. Yedalog's system of modes and compilation approach is inspired by Mercury [39]. A number of authors have also used Datalog as a language for data parallel computations. Shaw et al. [38] explored compiling Datalog programs into map-reduce jobs, focusing more on efficient recursive map-reduce computations. Socialite [37] is a distributed Datalog for recursive queries over social graphs.

7.2 Languages for Data-Parallelism

Existing tools for data-parallel computations, e.g., Flume itself [6], Pig [31], Spark [41], Dremel [28], LINQ [27], DryadLINQ [40], T-LINQ [7] and approaches that integrate MapReduce into SQL (e.g., [16, 13]) either embed a computation language into a language of data-parallel pipelines, or vice-versa. For example, in Flume, the user builds data-parallel pipelines using a library of pipeline operators, and expresses local computation by subclassing function objects. In Dremel, the large-scale structure of a query is expressed using a SQL dialect; if the user desires complex computation, they must write foreign functions in a distinct language. Jaql [3] and Yedalog represent both computation and data-parallelism in one language.

7.3 Database and Persistent Programming Languages

Object database systems seek to blur the boundary between set-oriented (readily parallelizable) database operations and general computation, along with persistence. The GemStone system [10], an early influential object-database system, extends Smalltalk's object and computational model smoothly to include shared, persistent, query-able data. Likewise, O2 [2] and ObjectStore [24], are object-database systems with a bridge to typed C++ objects. Yedalog, like most deductive database systems, naturally only expresses query-only computation, mostly sidestepping issues of persistence and consistency.

7.4 Semi-structured Data

There is a large body of work on working with semi-structured data, in particular data in XML and JSON formats. The canonical tools for working with XML data are XPath [8] and XQuery [9], which can be viewed as an adaptation of SQL-style queries to semi-structured data. Yedalog can be viewed as an adaptation of the Datalog family of database query languages to querying data in protocol buffer form. Jaql [3] is one of the more notable tools for querying large collections of JSON data and shares many goals with our work, although the two languages have many differences.

8 Conclusion

We have presented Yedalog, a declarative language for analyzing and transforming semi-structured data. Yedalog allows programmers to write both computation and data-parallel pipelines in the same formalism; both are first-class. Yedalog programs can run locally, and on interactive and batch distributed platforms. Yedalog hides operational details while allowing programmers to override the tools' choices. Despite being a young language, Yedalog's unique combination of features mean that it is already being used in production applications.

References

1 Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *Proceedings of the 15th International Conference on Extending Database Technology EDBT*, pages 132–143, 2012.

2 François Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-oriented Database System: The Story of O2*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

3 Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*, 2011.

4 Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1247–1250, New York, NY, USA, 2008. ACM.

5 Louis Brandy. Fighting spam with pure functions. `https://www.facebook.com/notes/facebook-engineering/fighting-spam-with-pure-functions/10151254986618920`, 2013.

6 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'10, pages 363–375, New York, NY, USA, 2010. ACM.

7 James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *ACM SIGPLAN Notices*, volume 48, pages 403–416. ACM, 2013.

8 World Wide Web consortium. XML path language (XPath). `http://www.w3.org/TR/xpath-30/`, 2014.

9 World Wide Web consortium. XML query (XQuery). `http://www.w3.org/XML/Query/`, 2014.

10 George Copeland and David Maier. Making Smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD'84, pages 316–325, New York, NY, USA, 1984. ACM.

11 Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

12 Google Developers. Introducing gRPC, a new open source HTTP/2 RPC framework. `http://googledevelopers.blogspot.com/2015/02/introducing-grpc-new-open-source-http2.html`, 2015.

13 David J DeWitt, Alan Halverson, Rimma Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. Split query processing in Polybase. In *Proceedings of the 2013 International Conference on Management of Data*, pages 1255–1266. ACM, 2013.

14 Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 181–220. Springer Berlin Heidelberg, 2011.

15 Apache Software Foundation. Apache Thrift. `https://thrift.apache.org/`, 2015.

16 Eric Friedman, Peter Pawlowski, and John Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, 2(2):1402–1413, 2009.

17 Google. Introducing the Knowledge Graph: things, not strings. `http://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html`, 2012.

18 Google. 11 billion clues in 800 million documents: A web research corpus annotated with Freebase concepts. `http://googleresearch.blogspot.com/2013/07/11-billion-clues-in-800-million.html`, 2013.

19 Google. Protocol buffers. `https://developers.google.com/protocol-buffers/`, 2014.

20 Ralph E Griswold and Madge T Griswold. *The Icon programming language*, volume 30. Prentice-Hall Englewood Cliffs, NJ, 1983.

21 Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Record*, volume 22:2, pages 157–166. ACM, 1993.

22 Johannes Hoffart, Mohamed Amir Yosef, Ilaria Bordino, Hagen Fürstenau, Manfred Pinkal, Marc Spaniol, Bilyana Taneva, Stefan Thater, and Gerhard Weikum. Robust disambiguation of named entities in text. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 782–792. Association for Computational Linguistics, 2011.

23 Martin Kay. Functional grammar. In *5th Annual Meeting of the Berkeley Linguistic Society*, 1979.

24 Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, 1991.

25 Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014.

26 Simon Marlow and Jon Purdy. Open-sourcing Haxl, a library for Haskell. `https://code.facebook.com/posts/302060973291128/open-sourcing-haxl-a-library-for-haskell/`, 2014.

27 Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD'06, pages 706–706, New York, NY, USA, 2006. ACM.

28 Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

29 Microsoft. Query hints (Transact-SQL). `https://msdn.microsoft.com/en-us/library/ms181714.aspx`, 2014.

30 Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 439–455, New York, NY, USA, 2013. ACM.

31 Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, New York, NY, USA, 2008. ACM.

32 Oracle. Database performance tuning guide. `http://docs.oracle.com/cd/B19306_01/server.102/b14211/hintsref.htm#i8327`, 2015.

33 Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. 1999.

34 Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL deductive system. *The VLDB Journal*, 3(2):161–210, April 1994.

35 Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

36 Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 324–334. ACM, 2006.

37 Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. Distributed Socialite: a Datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 6(14):1906–1917, 2013.

38 Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. Optimizing large-scale semi-naïve Datalog evaluation in Hadoop. In Pablo Barceló and Reinhard Pichler, editors, *Datalog in Academia and Industry*, volume 7494 of *Lecture Notes in Computer Science*, pages 165–176. Springer Berlin Heidelberg, 2012.

39 Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1):17–64, 1996.

40 Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, volume 8, pages 1–14, 2008.

41 Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in Cloud Computing*, pages 10–10, 2010.