Modern Datalog Engine Flavors

Bruno Rucy Carneiro Alves de Lima
University of Tartu.
Narva maantee 18, Tartu, Estonia 51090
bruno.rucy.carneiro.alves.de.lima@ut.ee

Abstract—This article is both a theoretical overview of the state-of-the-art-distributed-and-parallel implementations of datalog[1] engines, and a practical attempt at specializing a novel data structure to semi-naive evaluation[2], datalog’s most popular evaluation mechanism. We managed to describe all major data processing frameworks’ datalog engines, and successfully moved towards the formal definition, and implementation, of a new novel data structure for heterogenous fact data.

I. INTRODUCTION

Datalog and SQL[3] are the two biggest, and oldest, both having their origins in the 70’s, mainstream domain-specific languages for interacting with databases, with Datalog’s focus being inference and SQL’s, as of recent times, shifting more to an overall database management language. It is relevant to point out that even though both of these languages intersect, with respect to querying, their foundations are substantially different, first-order logic and relational algebra, thus, both can be optimized in wildly different ways, which leads to datalog always having at least some relevancy, due to its very big expressivity and not very complex evaluation semantics.

Datalog first came about with attempts to connect prolog to databases, in order to succinctly express inference rules over knowledge bases modeled as relational datastores, however, this proved to be considerably unwieldy and inefficient[4] back then, and one of the obviously major issues is that Prolog is highly expressive and undecidable, hence, a natural solution to this problem is to create a non-turing-complete subset of prolog, with its’ functionality specifically sculpted to fit the needs of a query engine.

II. DATALOG’S SYNTAX AND SEMANTICS

A. Syntax and Semantics

A datalog program \(P\) inherits much of its syntax and semantics from first-order logic. \(P\) consists of rules \(r\), which are written as horn clauses.

Horn clause is a specific type of clause, a sequence of logically interconnected atomic formulas, also called simply atoms, in which “atomic” means that the formula does not have logical connectives, and thus it is nothing but a predicate, interchangeably known as fact, with terms, which can be semantically understood as relations.

An example of a binary predicate is, within the context of a directed graph, the relation:

\[\text{HasEdgeTo}(x, y)\] (1)

that is true given that there indeed is an edge from node \(x\) to node \(y\).

Terms are either variables or constant values, hence in the previous atom both terms are variables. However, there can also be predicates which have constants, such as:

\[\text{ConnectedTo}(x, \text{NodeSix})\] (2)

in which NodeSix is a constant.

Given that we’ve informally established what clauses, atomic formulas and terms are, we can show the syntax of \(r\), with \(f_i\) representing the \(i\)-th atomic formula:

\[r \rightarrow f_1, \ldots, f_n\] (3)

This can be understood as: if \(f_1, \ldots, f_n\) holds, then \(r_1\) holds.

In the example above it is possible to see the constituents of a rule, in which \(r_1\) is the head, and \(f_1, \ldots, f_n\) is the body.

Programs have specific conditions which must be met in order to assure the non-turing-completeness of Datalog, such as:

1. All predicates must consist exclusively of ground terms, those which are constant.
2. A variable that occurs in the head of a rule must also occur in its body.

With programs defined, it is important to understand the difference between EDB, the Extensial Database, and IDB, the Intensional one. EDB refers to the database from which datalog will get its set of relations, in order to generate new rules and facts, which are said to belong to the IDB.

As with first-order logic, programs are only true with respect to some interpretation, which is built based on facts arisen from the EDB. Formally, Herbrand Base is the set of all ground formulas derivable from the Herbrand Universe, that is the minimal set of ground terms. As an example, let us define a datalog program[5]:

B. Example

Rule:

\[\text{IsConnected}(x) \rightarrow \text{HasEdgeTo}(_-, x)\] (4)

Ground Facts:

\[\text{IsConnected}(1), \text{HasEdgeTo}(2, 3)\] (5)

We have that the Herbrand Universe is: \(\{1, 2, 3\}\), and the base is “all derivable possibilities”: 

\[\text{HasEdgeTo}(1, 2), \text{HasEdgeTo}(2, 3), \text{HasEdgeTo}(3, 1)\]
Possibilities in order for $\text{IsConnected}(1)$:

HasEdgeTo(1, 1), HasEdgeTo(2, 1), HasEdgeTo(3, 1)  \hspace{1cm} (6)

Possibilities in order for $\text{IsConnected}(2)$:

HasEdgeTo(1, 2), HasEdgeTo(2, 2), HasEdgeTo(3, 2)  \hspace{1cm} (7)

Possibilities in order for $\text{IsConnected}(3)$:

HasEdgeTo(1, 3), HasEdgeTo(2, 3), HasEdgeTo(3, 3)  \hspace{1cm} (8)

The union of all of these possibilities is the Herbrand Base. Coming back to the interpretation, we can now define it as a subset of the Herbrand Base, and from that, models can be built, which are interpretations in which all ground facts are true.

From interpretations and models, we can define what exactly is a datalog program: the minimal model of a Herbrand Base. Thus, the minimal model of the given base is:

\[
\mathcal{M}_B = \begin{cases}
\text{IsConnected}(1), \\
\text{IsConnected}(2), \\
\text{HasEdgeTo}(2, 3), \\
\end{cases}
\hspace{1cm} (9)
\]

In which IsConnected(2) is derived from HasEdgeTo(2, 3).

III. SEMI-NAIVE EVALUATION

A. Naive evaluation

So far the only aspect of datalog programs that has been described has been what it is through the lens of model-theoretic semantics: the minimal model of an interpretation.

Now it is time to be explicit as to how to computationally evaluate a program.

The straightforward approach to do so is called Naive, and it does so by:

1. Start out with all EDB facts
2. Iterate through all of the EDB facts and infer new facts
3. Iterate through all of the EDB facts plus the newly inferred IDB ones
4. Repeat 3 until a fixpoint is reached

It is very ineffective and trivially of exponential complexity.

B. Semi-naive evaluation

A linear-time solution to this problem is called Semi-Naive, and works as follows:

1. Start out with all EDB facts
2. Iterate through all of the EDB facts and infer new facts
3. Iterate through the new facts and infer new facts
4. Repeat 3 until a fixpoint is reached

This is far more effective than naïve evaluation because no fact will be evaluated more than once.

Semi-naive evaluation happens to be the main evaluation method used by all of the state-of-the-art engines to be presented here, and a more concise overview on why it is so efficient can be found in[6].

IV. DECOMPOSABILITY

One of the first approaches to distributed and parallel Datalog, that are still relevant, has been on attempting to investigate the decomposition of databases and its relationship with datalog programs[7].

Decomposability takes advantage of the inherent graph structure that certain programs might be able to take advantage of. A datalog program $P$ is said to be disjoint distributive if $P(D_1 \cup D_2) = P(D_1) \cup P(D_2)$ so long as $D_1 \cap D_2 = \emptyset$.

An obvious example of a program that satisfies this constraint is the classical example, the transitive closure of a graph, which can be defined through the following composite rules:

\[
\text{Reachable}(x, y) \rightarrow \text{Edge}(x, y)  \hspace{1cm} (10a)
\]

\[
\text{Reachable}(x, z) \rightarrow \text{Reachable}(x, y), \text{Edge}(x, z)  \hspace{1cm} (10b)
\]

This is embarrassingly parallelizable due to its strong connectivity. All variables are connected to each other through common formulas, that is, if there are three variables, then there exists at least one variable such that it appears in formulas with both other variables.

A practical example of taking advantage of program connectivity is of partitioning the database by strongly connected components of the underlying graph.

Another type of distribute-able program is that of bounded distributive, which are those that are bounded recursive, such as the following one:

\[
wants(x, y) \rightarrow \text{likes}(x, y)  \hspace{1cm} (11a)
\]

\[
wants(x, y) \rightarrow \text{trendy}(x), \text{wants}(z, y)  \hspace{1cm} (11b)
\]

In this case, the decomposition of databases would be done such that they would all at least have a number of facts equal to the level of bounded recursiveness of the program.

V. SOUFFLE

A. The engine

Program analysis has been one of the leading use-cases of datalog, and the most industrial-quality open-source engine has been geared to provide considerable support for that specific use case.

Souffle[8] is a project that compiles datalog programs to optimized C++ code with custom parallel data structures, with their locking schemes specialized to the constraints of semi-naive evaluation. One of Souffle’s biggest enterprise sponsors is Oracle, which uses it to run static analyses on the entire openJDK, a massive project with a million variables, with the total program runtime lasting less than a minute[8].

The main difference, in a nutshell, from Souffle to most other state-of-the-art datalog projects, is that while Souffle specializes the program to a specification instead of executing it, while other engines directly execute the parsed datalog, which in sum means that Souffle is a datalog compiler, as
previously mentioned, and that the execution happens by feeding the EDB to the generated C++ program.

From Figure 1 we can see that the first step is parsing the Datalog syntax into an Abstract Syntax Tree. During this process a heap of optimizations are applied to the program, such as removing irrelevant rules, facts, aliases and propagating constants across rules.

The second step consists of RAM, Relational Algebra Machine, however, before it happens the AST must use semi-naive evaluation in order to generate the to-be-optimized-further code.

While the AST deals with syntactic improvements, RAM focuses on using relational operations over facts in order to provide heavier optimizations, such as swapping traversals for range queries, adding implicit joins, avoiding data races and removing non-recursive rules from the fixpoint calculation[9].

The transitive closure program, as defined previously, would look as follows, as an optimized RAM program:

```cpp
// populate fact tuples
project (1, 2) into edge;
project (2, 3) into edge;

// rule: path(X, Y) := edge(X, Y).
insert search edge do
  project (edge[0], edge[1]) into path;

// create delta knowledge for first iteration
merge path into delta_path;

// fixed-point loop
loop
  // reset helper relation
  purge new_path;
  // rule: path(X, Z) := edge(X, Y), path(Y, Z).
  insert search edge do
    search delta_path where {
      (edge[0] = delta_path[0]) and
      (edge[1] = delta_path[1]) not in path;
    }
    project (edge[0], delta_path[1]) into new_path;

  // fixpoint reached?
  exit counttuples(new_path) == 0;
  // book-keeping
  merge new_path into path;
  swap new_path and delta_path;
endloop
```

The optimizations that happen during that stage are mostly centered on transpiling the RAM program’s relational algebra operations to templatized C++ programs with much focus on inserts and lookup operations, which are those that dominate the run time of semi-naive evaluation.

One of the use cases of templating is to automatically select the most proper data structure to be used depending on the input facts, which can vary from a very specialized B-Tree, to be discussed next, and another also-very-specialized Trie, that handles facts with a large arity.

### B. The main data structure

#### B.1 The B-Tree:

For a considerable amount of time Datalog was seen as an expressive, but slow[10], tool for program analysis, largely due to the lack of a datalog implementation with strong focus on the lowest level and sufficient specificity for the context.

The following code on Figure 3 exhibits the final output, in a more humanly-readable way, of Souffle’s compilation.

```cpp
using Tuple = array<string, 2>;
using Relation = set<Tuple>;

Relation relation = edge, deltaPath = edge;
while (!deltaPath.empty()) {
  Relation path = edge, deltaPath = edge;
  while (!deltaPath.empty()) {
    Relation newPath, deltaPath;
    for (const auto & edge : deltaPath)
      auto l = edge.lower_bound((tt[1], 0));
      auto u = edge.upper_bound((tt[1], 1, 0));
      for (auto i = l; i != u; ++i)
        auto & t = i;
    Tuple r(t1[1], t2[1]);
    if (path.find(r) == path.end())
      newPath.insert(t);
  }
  path.insert(newPath.begin(), newPath.end());
  deltaPath.swap(newPath);
}
return path;
```

Fig. 3. Compiled representation [8]

Important aspects that a low-level data structure for datalog ought to have, stemming from the code example, are:

- the capability of being generic over any type with a strict weak order, since facts, relations in this code, can have virtually any arity, as it can be seen from the first two lines of code.
- efficient in-order traversal with range queries, $O(\log n + r)$, with $r$ as range, complexity, in order to skip previously computed facts
- strong and specific synchronization schemes, to be described next
- $O(\log n)$ general complexity

The most optimal type of data structure that could reasonably satisfy these conditions, according to the authors of Souffle[8], would be a search tree, in this case, most specifically a B-tree, and a throughout treatise on it can be found on[11], on Figure 4:

In short, the major difference between a regular Binary Search Tree and a B-Tree is that the latter has a sorted container of items instead of only one item, in each tree...
node, and even though theoretically this seems unoptimal, in practice, due to spatial, and possibly temporal, locality, it ends up being much more optimal.

**B.2 The B-Tree’s locking scheme:** However, given how computationally expensive semi-naive evaluation is, any additional overhead that the data structure might incur, such as locking schemes, can dramatically slow down the whole process, making net-beneficial parallelization a hard task, thus, Souffle’s B-Tree’s contribution is centered on a highly-specific-and-minimal scheme that, since general ones would add too much overhead, enjoys considerable performance benefits from having more cores.

The scheme takes advantage of the fact that there’s two distinct parts, one in which facts are queried, and another in which they are written. There’s never a situation in which a fact is queried as it is written, which implies that the locking mechanism ought to provide only write synchronization in specific parts of the tree.

Souffle’s implementation is based on a locking scheme, Figure 5, that satisfies the aforementioned constraints, seqlock[12].

A throughout description of Seqlocks is outside of the scope of this article, however, in essence, it achieves synchronization by assigning temporal attributes to data that will be recorded once the thread take-over starts, and confirmed once it ended, and in case there’s been a change on the attribute, the evaluation restarts.

The main difference between the tree implementation’s take on seqlock and the actual algorithm, is that while seqlock has threads for ensuring the locks of read and write operations, the tree combines both of these in one thread, with finer-grained non-blocking operations, other than write, that attempt to keep the lock over the Node for the minimum amount of time as possible.

**B.3 Search Fingers:** Temporal locality refers to using the same memory locations within a reasonably short amount of time, which opens up considerable caching possibilities that can yield substantial performance gains.

Datalog programs often exhibit temporal locality, since during semi-naive evaluation the same relations are often reused.

This behavior is often taken advantage of through the usage of search fingers, which are references to certain points within a sequence, that are usually set through some kind of logic that takes into account temporal locality.

The usage of fingers in Souffle is done by checking whether a fact falls within the range of any of the facts currently stored as fingers. This is a very low impact operation that can yield massive speedups in programs that are highly connected.

**B.4 Souffle Benchmarks:** The following, rather disappointing, benchmarks compare Souffle to other datalog projects, most specifically to the ones that initially inspired it: bddbddb[13] and µZ[14] alongside a SQLite[15]-based approach.

This benchmark, Figure 6, is the only one that showcases the same datalog program being ran both against other engines and against Souffle using other data structures.

While Souffle appears to be fast with considerably less memory usage than all other options, it is important to point out that, at the time of writing that article, the engines in that benchmark are far from being the state-of-the-art datalog engines, for instance, bddbddb’s website shows it’s been last updated on September 18th 2008.

It is important to point out that that’s not the only benchmark showcased on Souffle’s paper, but virtually all of them compare it to these engines, providing a not-state-of-the-art-relevant comparison.

**VI. BigDatalog**

**A. The Engine**

BigDatalog[16] is a datalog engine built with, not on-top-of, spark[17], and it’s main features are the possibility of writing declarative datalog programs that enable the effective evaluation of recursive and non recursive rules with aggregations.

A relevant issue with spark is that one requires much specific knowledge about it in order to put together non-trivial programs, leading to the development of official libraries such as GraphX[18], that ease the complexity for writing graph algorithms and analytics.

Most specifically, the following steps would be necessary in order to write a manual transitive closure spark program:

1. explicitly distribute the graph edges over the spark cluster
2. invert the edges in order to make them join-able
3. cache the RDDs in order for memory to not blow up
4. write the fixpoint loop

GraphX efficiently leverages spark for its main goal making writing programs a considerably less complicated endeavor, albeit still far more verbose than datalog, compared to the previously given example, however, it still suffers from the issue that Spark itself just isn’t suitable for recursive iterative queries, such as transitive closure, due to the fact that a new job is spawned for every iteration.

B. Spark extensions

BigDatalog solves the recursion issue by directly forking spark and making fundamental changes to its scheduling and SQL engines, such as the extension of spark’s sql planner, to make use of some form of cyclical plans and a less rigid scheduler, given datalog programs are guaranteed to finish.

The centerpiece of the engine is DeALS[19] which, given a database schema and sets of rules, provides a logical plan that is mapped to SQL, to be optimized and evaluated by BigDatalog, in a pipeline that is quite similar to Souffle, except that souffle compiles to c++ code while Spark immediately executes the program.

DeALS stems from DeAL[20], Deductive Application Language, which is a datalog flavor geared towards analytics, hence BigDatalog’s focus on providing aggregations. DeALS extends DeAL with datalog extensions that are a necessity when dealing with real-life data, such as negation[21], subsumption stratification[22] and the aggregation capabilities.

![Recursion Operator](image)

Fig. 7. Transitive Closure Program plan[16]

The output of DeALS is a tree which reasons whether a rule is recursive or non-recursive, and how ought their relation-algebra translation be. Figure 7 showcases the transitive closure program’s logical structure, where each branch represents a rule, out of which exit rules are the non-recursive, and non-exit represent fixpoint steps.

The next step is to create the physical execution plan, a translation of the generated logic tree into proper spark SQL, as it’s shown on Figure 8, with Z, from the transitive closure program, the common connecting edge, being the join argument. There are some non-obvious aspects of the translation that the authors felt the necessity to point out:

- All joins are translated to series of binary ones, left-to-right
- Shuffle operations are added according to Catalyst

BigDatalog executes the physical plan with not only SQL optimizations, but lower level ones targetted specifically at its parallel semi-naive evaluation, in order to ensure that it will indeed benefit speed-wise from it.

Similarly to Souffle, it puts considerable effort in finding the most suitable data structure, which in this situation is the one which would avoid one of the slowest spark operations, shuffling.

SetRDD’s are a spark abstraction that assume that the RDD is properly partitioned within the memory, not preemptively shuffling it, providing a considerably big performance boost, alongside with convenient set operators, such as diff and union.

In order to make sure that the RDD is correctly partitioned in memory, BigDatalog makes use of a specific partitioning scheme, which is to do it based on the first predicate. Looking at the physical plan on Figure 8 it would be the left node of the recursive rule. Further reasoning as to why this is optimal is digressed in [16].

Decomposability, the assurance that a Datalog program could partition its EDB and join the results in parallel as opposed to computing over the entire DB, as seen on section 4 of this article, is also an optimization target for BigDatalog. Decomposable programs are identified with a technique called Generalized Pivoting[23], that is run on the logical level, and they are executed as a different plan than shuffle-based programs.

The last relevant optimization is the scheduler one, which deals directly with spark’s limitation of creating one job per iteration. In order to enable the fixpoint iterations to have awareness of the other, a FixpointStage is added that contains fixpointjobs, that are jobs in which their iteration stems from the previous’ SetRDD up until it is empty.

C. Benchmarks

BigDatalog’s benchmarks are considerably more relevant than Souffle’s, with its comparisons being against much more relevant engines.

For this article, given that BigDatalog claims to be superior to spark and GraphX, the most relevant benchmark is the one
that compares the performance of a real-world query translated to native spark, GraphX and datalog.

The graph on Figure 9 clearly shows that BigDatalog either performs at the same level or better than its direct competitors, however, it isn’t that much better.

Needlessly to say, the biggest advantage is not performance but expressivity.

The following program, Figure 10 refers to the MLM query:

```
r1. networkTC(M, N) ← sponsor(M, ...).
  r2. networkTC(M, N) ← sponsor(..., N).
  r3. networkTC(M, N2) ← networkTC(M, M1), sponsor(M1, N2).
  r4. memberTotalSales(M, sum(S)) ← networkTC(M, NM),
    memberSales(NM, S).
  r5. memberBonusSelf(M, B) ← memberSales(M, ST),
    memberTotalSales(M, S), schedule(LS, RS, BP),
    S >= LS, S < RS, B = ST + BP.
  r6. memberBonusFrontline(M, sum(B)) ← sponsor(M, NM),
    memberTotalSales(NM, S), schedule(LS, RS, BP),
    S >= LS, S < RS, B = S + BP.
  r7. bonus(sum(B)) ← memberBonusSelf(M, B1),
    memberBonusFrontline(M, B2), B = B1 + B2.
  r8. grossProfit(sum(P)) ← sales(..., P).
  r9. netProfit(NP) ← grossProfit(P), bonus(B), NP = P - B.
```

In sum, this program calculates the profit of a multi-level marketing company after handing out its due payments to its members. The first three rules compute the transitive closure, the fourth computes all sales, that is comprised of rule five plus rule six. rule seven computes the total bonus paid by the compute, rule 8 the gross profit and at last, rule 9 yields the net profit[16].

It is obvious that writing the same in Spark would incur either a considerable amount of low-level RDD code or many-more-than-twenty lines of a SparkSQL aggregated query.

The next query from the benchmark is people you may know, that could be used to directly recommend new connections on a social network.

```
r1. uarc(X, Y) ← arc(X, Y).
  r2. uarc(Y, X) ← arc(Y, X).
  r3. cnt(Y, Z, count(X)) ← uarc(X, Y), uarc(Z, X), Y!= Z, ~uarc(Y, Z).
  r4. pynk(X, W9, topk(10, Z)) ← cnt(X, SID, Z), pages(X, W2, ..., W9).
```

This query, Figure 11, builds an undirected graph using the first two rules, and then counts the number of shared connections between edges which are not directly connected with r3, to then use an aggregation in order to recommend the top 10 shared connections.

In sum, BigDatalog not only provides realistic benchmarks, but also rare examples of useful analytics datalog queries.

VII. Cog

A. The Engine

Cog[24] is a recent attempt at replicating the most basic form of BigDatalog, positive datalog without aggregation and stratification, on a more modern framework, flink[25]. Similarly to spark, it is a distributed dataflow platform, however, it is friendlier to datalog semantics since it has first-class support for iterations, that is, by properly implementing them as bounded cyclical dataflows, while spark, as previously seen on BigDatalog, does not provide specific constructs for such specific iteration and instead relies on directly extending the scheduler, which requires both substantial work and knowledge of the platform’s internals.

Cyclic dataflows in Flink almost exactly mirror fixpoint semantics in semi-naive evaluation, by feeding the next iteration with the previous one, avoiding spark’s one-job-per-iteration approach.

An extensive comparison between spark and flink can be found in [26], with benchmarks in the end of the paper showing that flink is on average faster than spark, however, it provides significantly less fine-grained control than the former.

B. The Point of Using Flink

Cog replicates its’ competitor’s program compilation process: first creating a logical plan, then a physical one.

The logical plan is done with Apache Calcite[27], instead of DeALS, a framework that provides useful database querying primitives, such as a SQL parser and relational algebra expressions.

```
RepeatUnion

Recursive predicate
```

```
Project
arc
Join
Seed node subtree

δ tc

Iterative node subtree
```

```
Project
Project
arc

Fig. 12. Cog Logical Plan[24]
```
As it can be seen, this, Figure 12, and BigDatalog’s plan are equivalent, since both of them represent a tree with all nodes sharing the same relational operators.

The physical plan translates the relational algebra expressions unto flink’s DataSet API, that in turn makes use of the \texttt{iterateDelta} operator for the computation, however, there is not much detail as to how Cog’s physical plan is optimized other than which generic functions are used, making it seem that none of BigDatalog’s platform-independent optimizations, such as decomposibility for instance, happen at all.

In order to further compare spark and flink, within this context, BigDatalog and Cog’s authors wrote generic queries that attempt to show how datalog abstracts all underlying implementation details.

The two following figures not only showcase the low level details of the dataflow graph, but the recursion approach.

The results are not very good, with BigDatalog beating Cog in a considerable number of datasets, despite being much more expressive and on a heavier platform.

Another flaw of Cog’s paper is that there’s no comparison against raw expert-written flink queries, which would happen to be the benchmark with the most practical value.

\section*{VIII. DDLOG}

\subsection*{A. The Engine}

Differential Datalog\cite{DDlog}, DDlog, is more akin to Souffle than BigDatalog and Cog, in the sense that it compiles to very effective code written in a high-performance language, Rust.

Unlike Souffle, that focuses on highly optimized and parallel C++, DDlog’s generated code makes use of a very experimental, and obtuse, dataflow paradigm that attempts to somewhat generalize the problem of distributed streaming computation, Timely Dataflow\cite{TimelyDataflow}.

One of the unique aspects of Timely Dataflow is that not only is it parallel, but it fully supports cyclic dataflows in a streaming context, going considerably beyond what flink can offer, with the added benefit of the implementation language being much more performant and memory safe\cite{Rust}.
Another striking difference is that its’ datalog language is very rich, with highly strongly-typed and expressive constructs, such as aggregations, a powerful type system, loops and pure functions being, as it seems, much more expressive than BigDatalog’s DeALS.

B. Compilation process

DDlog’s usage of Timely Dataflow actually happens through a slightly higher-level interface, Differential Dataflow[31], that provides map-reduce-like operators, similar to those used in spark and flink, while at the same time bringing the benefit of making sure that all computations are always up-to-date, effectively killing one of the most quintessential issues with materialized views.

The core compilation pipeline of Datalog, explained in less detail than Souffle’s, is on Figure 16

![Fig. 16. DDlog compilation pipeline][28]

The first step is parsing, which happens on a complex parser, written in Haskell, gears towards memory effectiveness, such as attempting to share indexes, reference counting and minimizing the usage of expensive operations as much as possible.

The second part is the rust compilation, where the generated differential dataflow program, similarly to Souffle, is a native binary.

C. Benchmark

Out of all datalog flavors introduced here, DDlog is the only one that is used in production software in such a way in which it is the software’s main language.

OVN[32] is a high-profile networking project that is often used in cloud platforms in order to decouple the logical and physical network, however, it often struggles with very big dynamic network topologies where all the node data is re-evaluated whenever change happens.

The authors of DDlog rewrote a subset of OVN, northdb and not only achieved a substantial speedup, but managed to do so in less than half as many lines of code as the original C implementation.

This benchmark, Figure 17, does not refer to OVN, but a different software, a Firewall manager, that was implemented both in DDlog and in an expert-crafted Java, with iterative computations.

It is unsurprising that DDlog is faster than Java, given that it is executing Rust code, a much more lower level language, nevertheless, the results are still impressive.

![Fig. 17. DDlog Benchmarks][28]

THEORETICAL CONCLUSION

We can summarize this article so far on Table I, that succinctly classifies all mentioned datalog engines.

<table>
<thead>
<tr>
<th>Name</th>
<th>Last release</th>
<th>Distributed and/or Parallel</th>
<th>Language</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>mZ</td>
<td>Jan 2020</td>
<td>Distributed and/or Parallel</td>
<td>C++</td>
<td>Z3</td>
</tr>
<tr>
<td>bddbddb</td>
<td>Aug 2007</td>
<td>Neither</td>
<td>Java</td>
<td>bddbddb</td>
</tr>
<tr>
<td>Souffle</td>
<td>Sep 2020</td>
<td>Parallel</td>
<td>C++</td>
<td>Souffle</td>
</tr>
<tr>
<td>BigDatalog</td>
<td>Oct 2016</td>
<td>Distributed and Parallel</td>
<td>Scala</td>
<td>Spark</td>
</tr>
<tr>
<td>Cog</td>
<td>Apr 2020</td>
<td>Distributed and Parallel</td>
<td>Scala</td>
<td>Flink</td>
</tr>
<tr>
<td>DDlog</td>
<td>Mar 2020</td>
<td>Distributed and Parallel</td>
<td>Haskell/Rust</td>
<td>Diff. Dataflow</td>
</tr>
</tbody>
</table>

THE SPLIT LIST

A. The Macro-structure

In this section we develop a prototypical data structure that would be suitable as an index to store heterogeneous fact data, to be used in the semi-anive evaluation step of a datalog reasoner.

The Split List is a semi-ordered specific in-development data structure, inspired by the Skip List[33] and the B-Tree, whose main goal is to be a concurrent low-memory-consumption-and-not-compressed index for storing datalog facts, or virtually any type of data that has a strict weak order.

The data structure aims to satisfy these requirements by exploring the Skip List’s usage of the geometric distribution, in order to simulate a balanced search tree, alongside B-Tree’s clever array bucketing to take the most advantage of possible referential locality.

Figure 18 pictures the top-level of the list, that holds pointers to arrays of geometrically distributed arrays, with the innermost of them called buckets that have a globally defined constant fixed size. We say that the inner arrays of the list are “geometrically distributed” on the sense that the amount of data that each array holds is given by the following formula, with $h$ as height, the number of sublists, $|S_i|$ as the length of list of rank $i$, and $|S|$ as $\sum_{i=1}^{h} |S_i|$ as

$$|S_i| = |S| \cdot \frac{1}{2^i}$$

(12)
B. The Insertion Operation

Insertion starts by a procedure that will ensure the geometric distribution of the list, in which the to-be-added datapoint will be assigned a random list that it will be added to:

$$\text{height()} := \text{int}(\text{abs}(\text{log}_2(\text{rng}(0, 1))))$$  \hspace{1cm} (13)

This function has $O(1)$ complexity and successfully approximates the geometric distribution, thus, if 1 million elements are added, the maximum value would be around 20.

In case a datapoint is assigned a height that is bigger than the current maximum height, then the difference between the current and new height will be added, vertically extending the list.

After the datapoint’s height is assigned, a binary search will occur over the height’s list’s maximum, that will contain buckets of some fixed size. The following Figure, 19, illustrates how do the buckets inside a height look like.

Thus we can see that inside each height we will have buckets that not only are they internally ordered, but are so relatively to each other as well.

The worst-case insertion complexity is as follows:

1. Randomly assigning the datapoint to a height: $O(1)$
2. Bisecting the maximum of the buckets inside the biggest height, $S_1$: $O(\log_2 \left( \frac{|S_1|}{B} \right))$ with $B$ being the bucket size
3. Bisecting the bucket to find where the datapoint will be inserted: $O(\log_2 B)$
4. Inserting in the middle of the bucket: $O(B)$

Worst case complexity:

$$O(\log_2 \left( \frac{|S_1|}{B} \right)) + O(B + \log_2 B) \approx O(B)$$ \hspace{1cm} (14)

roughly constant time. It’s very clear that, due to the bucketing, inserting a new element will only impose a local lock, on the bucket, leaving $n - B$ of the data structure free for concurrent insertions.

However, given that the buckets have a fixed size, there ought to be some kind of periodic balance routine that will make sure that no buckets are overflowing.

C. The Find Operation

Finding some value is a trivial operation:

1. Iterating through all height lists: $O(h)$
2. Bisecting all height’s lists’ buckets maximums: $O(\sum_{i=1}^{h} \log_2 \left( \frac{|S_i|}{B} \right))$
3. Bisecting one bucket in each height’s lists: $O(h \cdot \log_2 B)$

Worst case complexity:
\[ O(h) + O\left(\sum_{i=1}^{h} \log_2 \left( \frac{|S_i|}{B} \right) \right) + O(h \cdot \log_2 B) \] (15)

This operation could be significantly sped up at a very small performance, but a high complexity, cost by adding pointers to the buckets to an interval tree.

This would change the find complexity to, assuming that the tree is balanced, worst case, in which \( r \) is the amount of overlaps:

\[ O(\log_2 \left( \frac{|S_i|}{B} \right)) + O(r \cdot \log_2 B) \] (16)

**D. Benchmarks**

In order to compare the effectiveness of the split list for semi-naive evaluation, the benchmarks emulate the process by only comparing operations that are relevant during it, and against other data structures that are able to support generic key types. Google’s reference B-Tree implementation was selected as the B-Tree[34] and The skip list implementation with most stars was picked[35]. All code is in golang[36].

![Fig. 21. Random insertions benchmark](image)

On random insertions, Figure 21, we can see that the Split List is considerably faster than its direct predecessor, the split list, and manages to steer close to the B Tree, providing close to state-of-the-art results.

Random lookups, Figure 22, due to the unoptimized complexity, provide a more worrisome result, since while insertions seemed to have scaled up in a linear manner, lookups in the Split List are growing at a faster rate than the B-Tree, being closer to the Skip List.

The last benchmark, Figure 23, shows that the less-than-ideal Split List is already competitive in terms of heap memory allocation with regards to the Skip List, being twice as better, and at the same level as the B-Tree, once again strengthening the argument that it is memory efficient.

![Fig. 22. Random lookups benchmark](image)

**CONCLUSION**

From the experiments promising results were obtained, showing that the Split List is a worth-pursuing idea that provides not only decent results speed-wise, taking into account that it is heavily unoptimized, but memory-wise too, with it being the most resourceful data structure tested on the benchmarks, being up to 2.5 times more memory-efficient than the Skip List.
REFERENCES

[23] - Jürgen Seib and Georg Lausen. 2020. Distributed Graph Analytics with Datalog Queries in Flink. DOI:https://doi.org/10.1007/978-3-030-61133-0_6


