Towards Holistic Query Optimization for Datalog

Nick Rassau
supervised by Prof. Dr. Felix Schuhknecht
Johannes Gutenberg-University
Mainz, Germany
{rassau|schuhknecht}@uni-mainz.de

ABSTRACT

The logic programming language Datalog is today used in a wide array of applications, which led to the rise of various specialized Datalog engines such as Soufflé, DDlog or LogicBlox, where all of them have their reason to be. While some of them implement basic forms of query optimization, the applied optimizations are unfortunately always deeply hard-wired into the specific engine, making a transfer of a technique from one system to another one or the extension of an engine with a new technique a cumbersome and manual engineering process.

Consequently, in this work, we propose to optimize Datalog engines holistically. Instead of having the entire optimization pipeline materialized within the Datalog engine, we generate an optimized general-purpose representation of the query plan outside of the engine. Then, we translate this general-purpose plan into the representation of the target engine, while applying as many optimizations as possible. With this approach, our goal is to make query optimization in Datalog more sustainable by optimizing a broad range of Datalog engines, rather than focusing on just one.

VLDB Workshop Reference Format:
Nick Rassau. Towards Holistic Query Optimization for Datalog. VLDB 2024 Workshop: VLDB Ph.D. Workshop.

1 INTRODUCTION

Datalog is a carefully restricted logic programming language that has seen a surge in popularity in recent years. Originally, Datalog was conceived as a database query language that operates on finite sets only [7]. Nowadays, Datalog is being used in a wide array of applications, from program analysis [11, 14] to network monitoring [1], distributed computing [2] and distributed storage [8]. What makes Datalog so popular is that it is designed as a *declarative* programming language, where programmers specify what a computation should achieve rather than how that result can be achieved computationally.

While originally, Datalog queries were rather small in size, modern (often synthesized) Datalog queries involve large and complex computations over structured data, easily consisting of hundreds of lines of Datalog code [6]. In this context, generating a naive execution plan and running it often reaches its limits and leads to poor performance. As a consequence, popular engines such as Soufflé [11] allow users to manually optimize the execution by annotating their queries. This has two problems: (1) For the user, it is notoriously difficult to come up with reasonable physical design decisions, as they require a deep understanding of the internals of the engines as well as the requirements of the dataset and the query. (2) It breaks with the declarative nature that Datalog was supposed to inhabit.

So why do Datalog engines lack automatic query optimization, like it has been implemented in relational systems since decades? The reason lies in one significant difference between relational and Datalog processing: While in the relational world, the database is known up-front and the queries arrive on-the-fly, in the Datalog world, the situation is typically the other way around. For example, consider static program analysis performed by an IDE: A particular analysis, such as type checking, is encoded as a Datalog query up-front. However, the database in form of the program to analyze arrives on-the-fly as it constantly changes under user inputs. This means that in contrast to a relational engine, a Datalog engine cannot simply calculate statistics of the database, such as size, cardinality, domain, and data distribution, up-front and use it to steer the query optimization.

Nevertheless, the first steps have been carried out to introduce automatic query optimization to the Datalog world. To identify cost-optimal join orders, Soufflé now implements a so-called *feedback-directed* optimization strategy [3]. It works as follows: First, the engine performs a profiling run using the query's naive execution plan on some sort of representative database to gather statistics about the execution. To perform join order optimization, the engine builds join cardinality estimates from these statistics and uses them to compute a cost-optimal join order. This join order is then worked-in during a recompilation pass to produce an optimized execution plan, which hopefully performs well on the actual database.

While this implementation of feedback-driven optimization is a step in the right direction, its downside is that the entire optimization process is currently highly baked into a single specific engine, namely Soufflé. Integrating an optimization into a specific engine is (a) a complicated and lengthy task, and (b) optimizes only one specific engine. This is unfortunate, considering the large amount of existing Datalog engines, which often have been designed with a different use-case and domain in mind. For instance, while Soufflé has been mainly design for the aforementioned static program analysis, `DDlog` [10] targets incremental computation scenarios. A mixture of both is `IncA` [13, 14] which is used for whole-program analysis in an incremental way. Another appearance is `Dedalus` [2], a logic programming language that is used to specify distributed services and protocols. Lastly, we mention a commercial Datalog engine named `LogicBlox` [4] that is used for sophisticated analytical applications and predictive analytics. It is clear that all these

Proceedings of the VLDB Endowment. ISSN 2150-8097.

systems have their reason to be. Hence, our goal is to develop a framework that optimizes *all* of them, not just a single one.

1.1 Towards Holistic Query Optimization

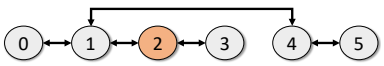
As a consequence, we advocate that automatic query optimization of Datalog should happen in a *holistic* fashion. Instead of hard-wiring the entire optimization pipeline into a specific engine, it should be split into two parts: In the first part, we gather runtime statistics of the not fully optimized execution on a representative database. This still happens *within* one particular engine, such as Soufflé. The second part happens *outside* a particular engine and uses the gathered information to optimize an intermediate representation (IR) of the execution plan. Such an IR can be produced by a front-end, either from a given Datalog query or directly synthesized. Then, we compile the IR containing the applied optimizations to the concrete physical execution plan of a specific engine. Again, by this, a particular optimization becomes available for a large number of systems, not just for a single one.

2 STATE-OF-THE-ART

Before diving deeper into our design vision, we will walk through a brief introduction to Datalog and the Datalog engine Soufflé. We then discuss the current feedback-driven optimization strategy implemented within Soufflé and its limitations.

2.1 Datalog

To get a better understanding of Datalog, let's consider a simple example. Suppose we want to visit 6 sightseeing attractions, enumerated from 0 to 5. Knowing the train connections between those and that we arrive at attraction 2, we want to compute whether we can visit all of them. The situation could look like this:



As initial knowledge we have a set of train connections $\{0 \leftrightarrow 1, 1 \leftrightarrow 2, 1 \leftrightarrow 4, 2 \leftrightarrow 3, 4 \leftrightarrow 5\}$ and a set of visited sightseeing attractions, which only contains $\{2\}$ at the beginning. These so-called *facts* are simply materialized as tuples in corresponding tables `train_connection = {(0,1), (1,2), (1,4) (2,3), (4,5)}` and `visited = {(2)}`. To compute how to travel from one to another sightseeing attraction, we next define a Datalog *program*, which consists of a set of *rules*. The repeated execution of these rules on the existing facts produces new facts, which are inserted again in the table(s). This is done until no new facts are generated anymore and a fix point is reached. For our travelling example, the following two rules populate the `visited` table with new facts:

```
visited(x) :- visited(y), train_connection(y,x)
visited(x) :- visited(y), train_connection(x,y)
```

The first rule states that if we visited a sightseeing attraction y and there is a train connection from y to x, we can visit x as well and therefore x should be inserted in the `visited` table. The second rule is the same as the first but in the other direction. Let us look at the syntax of a rule: A rule consists of a head and a body, separated by `:-`. To evaluate a rule, Datalog starts at the first predicate in the body, `visited(y)` for the first rule, and queries all tuples in the `visited` table. For each tuple, the program checks if the second predicate in the body, `train_connection(y,x)`, is true, by querying all tuples in the `train_connection` table that have y as their first element. If so, the head tells what will happen: the tuple `(x)` is inserted into the

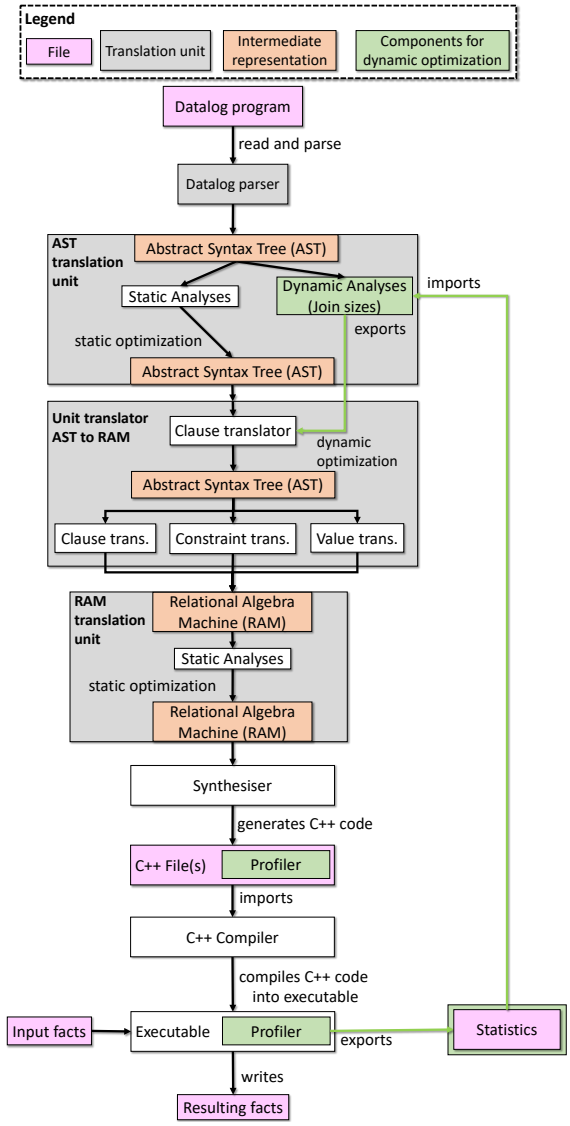


Figure 1: Soufflé architecture and its optimizations.

`visited` table. Initially, this is the case for `(1)` and `(3)`, neighboring `(2)`. But those three attractions are not all that can be visited, thus the rules are evaluated a second time, resulting in `visited` to contain $\{2, 1, 3, 0, 4\}$, which is the fixpoint.

2.2 Optimizations in Soufflé's Architecture

Datalog programs can be large and complex, specialized and highly tuned engines are required to manage the data and to perform the recursive rule evaluation efficiently. Ideally, this includes optimizing the execution. One such engine is Soufflé, which will play a special role in our following design but other engines would also be possible. The reason why we choose Soufflé is that it is open-source, actively developed, has a large user base, and was used often as a research target [3, 9, 15]. Also, despite being often used for static program analysis, its engine is designed in a rather general-purpose fashion. The high-level architecture of Soufflé is shown in Figure 1. Soufflé, using a multi-step compilation, first parses the Datalog program into an `Abstract Syntax Tree` (AST), which is a

tree-style representation of the input program, where each node represents an element or statement. The AST is for instance used for semantic analysis to check the program for errors. The AST is then transformed into a so-called `relational algebra machine` (RAM). Precisely, the RAM consists of a sequence of relational algebra operations, relation management statements, and control flow constructs with parallelism, like `Scan`, `Project`, `Filter`, `Aggregate`, `Join` and `Union`. Finally, based on the RAM, Soufflé starts to synthesize C++ code, which is then compiled into a binary that can be executed to evaluate the logic of the Datalog program.

During this multi-step compilation, Soufflé applies two classes of optimizations: The first class are *static optimizations*, which only operate on the representations of the Datalog program (i.e., AST and RAM), but without factoring in any concrete database (as it is typically not available when the compilation happens). On the AST, Soufflé applies magic set transformation [5], removing empty and redundant tables, or resolving aliases. On RAM, it applies only a handful of static optimizations, such as converting index scan operations to filter/existence checks, transforming consecutive filters into a single filter containing a conjunction, or pushing one aggregate as far up the loop nest as possible.

The second class of optimizations are *dynamic optimizations*, which also factor in a concrete database. Join order is the only dynamic optimization currently implemented in Soufflé and is based on the previously mentioned feedback-directed optimization pipeline. For join ordering, this pipeline entails (1) a profiling part, which instruments the generated code to collect join cardinality estimates, (2) an optimization step, which applies Selinger's algorithm [3, 12] on the statistics to determine the cost-optimal join order, and (3) an application phase, which materializes the determined join order in the AST for the next round of compilation. Unfortunately, all three steps are heavily inter-weaved with various components of the engine, rendering any modification of the optimization process or the extension of the system with a new type of optimization extremely cumbersome. Further, due to this deep inter-weaving, the transfer of a successfully applied optimization (such as join ordering) from one Datalog engine to another Datalog engine requires again careful and manual engineering.

3 TOWARDS HOLISTIC QUERY OPTIMIZATION

To overcome these limitations, we advocate to optimize holistically. Instead of hard-wiring the entire optimization pipeline into a single specific engine, we decouple both the optimization step (2) and the application step (3) from a concrete Datalog engine. Only the profiling step (1) is required to happen locally in a specific engine.

*Consequently, our **research goal** is to lay the foundation of such a holistic query optimization approach for Datalog. The developed methods are aimed to be (a) practical in use, (b) compatible with a large number of different engines, and (c) robust against a variety of workloads.*

Our vision of an architecture approaching this goal is illustrated in Figure 2. Analogue to the original feedback-directed optimization, our design also requires a way of executing step (1), namely performing profiling runs to collect statistics on a concrete database. This happens in one dedicated *profiling engine*, which must be able to capture and output relevant statistics of runs. We use Soufflé for this purpose, as it (a) already incorporates a statistics collector

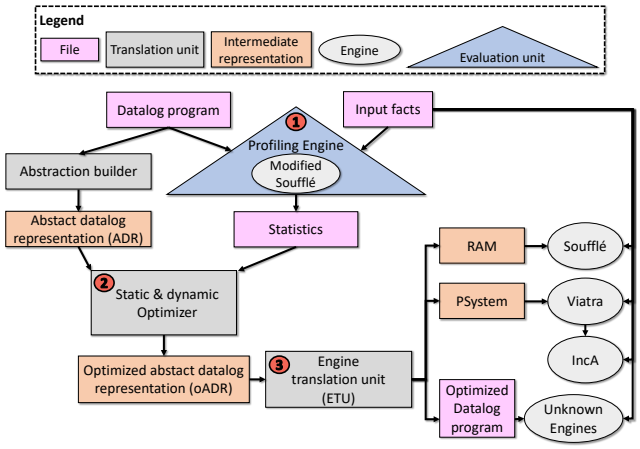


Figure 2: Vision of a holistic query optimization framework.

for its join order optimization and (b) is open-source. After having gathered statistics, the entire remaining pipeline happens outside a particular engine. It starts with the translation of the Datalog program into an *abstract Datalog representation* (ADR) by the *abstraction builder*. ADR is designed to capture the syntax, structure and semantics of the Datalog program – on a conceptual level, it resembles the AST representation of Soufflé but is not specialized for any specific engine. Next, the ADR is handed over to an *optimizer*, which is responsible for applying both static optimizations, such as removing empty/redundant tables, and dynamic optimizations, such as join ordering. To perform the latter, the optimizer imports the previously collected statistics. The optimizer then outputs an *optimized abstract Datalog representation* (oADR), which contains all applied optimizations either directly (e.g., if the syntax tree was reorganized) or indirectly in the form of annotations (e.g., whether a table should be materialized in a specific physical layout). This oADR then goes into the *engine translation unit* (ETU), which is responsible for producing a representation that a particular target engine can understand and import. For example, for Soufflé, this representation would be RAM, for Viatra, it would be PSystem. Note that the ETU must contain a specific translation scheme for each engine, as not all systems support the same feature set, data structures, and operations. In general, the ETU tries to work in as many of the oADR's indirect optimizations as possible during translation. If an optimization is not supported by a target engine, it will be ignored. As a fallback, if the ETU does not know the target engine, it will produce optimized Datalog code in a file.

3.1 Example

For a better understanding, let us go through our optimization pipeline in Figure 3, showing the example we previously used. Precisely, we show an optimization that decides on whether to materialize the `train_connection` table as a clustered index or rather as an unclustered index. As clustered indexes are faster but also consume more space than unclustered indexes, the decision depends on the space constraints and selectivity of the accesses (for high selectivies, an unclustered index might provide sufficient performance, while for low selectivities, a clustered index is essentially mandatory). Consequently, in the first step, we use our profiling engine to determine the *selectivity* of accessing `train_connection`, i.e., the ratio

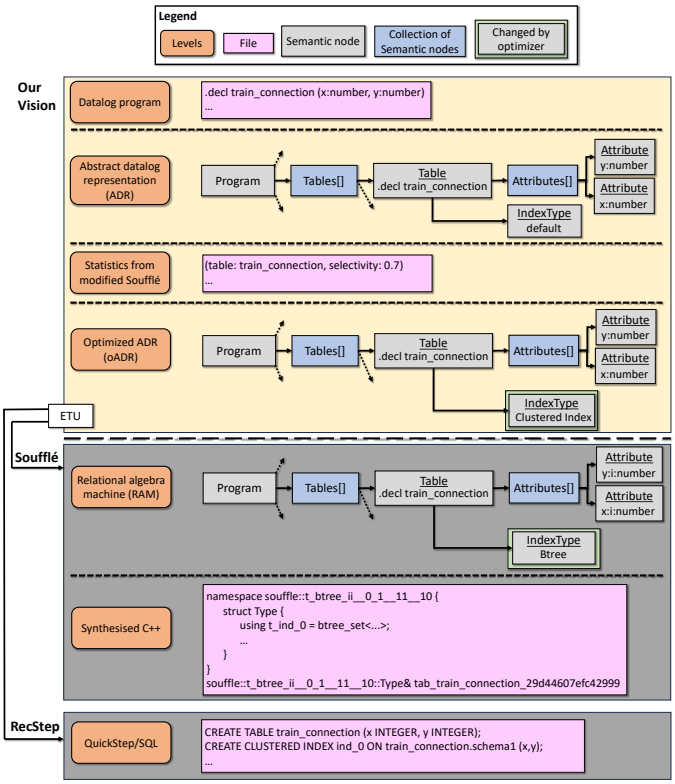


Figure 3: Example for the visioned architecture.

between the resulting size of a range query and the total size of the table. Assuming this selectivity is rather low with w=0.7, the desired outcome is to use a clustered index. Let us see the individual steps of reaching this: In the **Datalog program**, the table `train_connection` is simply declared without any physical information. From this, we generate the **ADR**, which mirrors the declaration closely but adds an `IndexType` to the table, which is left to *default*. This means that if no further optimization information is applied, the ETU generates the default representation for this table. However, as **statistics** are available, the optimizer decides to translate the ADR into the **oADR** which is annotated with the hint to generate a clustered index for this table. From the oADR, the concrete representation of the target engine(s) can be produced by the **ETU**. In the example, we show this for Soufflé and RecStep[9]. For Soufflé, the ETU will translate the hint to generate a clustered index by encoding a request to generate a B-tree for this table (which is a clustered index in Soufflé). Finally, the internal code generation of Soufflé will translate this into the corresponding **C++** code, where we can see that `train_connection` has now an index `t_ind_0`, which is a `B-tree`. For RecStep, which uses QuickStep as an RDBMS based on SQL, this translation is different. We produce two SQL statements, one for the table itself and the other one originates from the optimization, leading to the generation of a clustered index.

4 RESEARCH CHALLENGES AND QUESTIONS

Let us conclude with the high-level research challenges of this project. First, step (1) requires to execute a Datalog program on a concrete database to produce statistics. However, since the program cannot be dynamically optimized for this run, it might perform poorly. Consequently, this run has to be performed on a database that is small but also representative for the actual workload. Identifying such a feasible representative workload automatically is an open challenge, that we will have to tackle early on. Also collecting meaningful statistics for Datalog is significantly more difficult than in the relational world, since new facts are recursively produced. This requires a careful collection, aggregation and interpretation of statistical data during the profiling run. Second, step (2) optimizes the program holistically based on statistics gathered in a specific engine in order to optimize for other engines as well. It remains an open research question how sensitive optimizations are to be transferred from one engine to another, as typically, they are hard-coded for a specific engine and the engine itself is tailored towards a particular domain. Further, we will investigate different classes of optimizations (e.g., physical design decisions, access paths, join ordering) under workloads from different domains, while we anticipate that their performance will vary significantly in this regard. Third, in step (3) where the translation into the specific representation of the individual engine happens, we have to carefully map each optimization in the oADR to the corresponding engine construct. This mapping might be ambiguous (e.g., the hint to create a clustered index could be translated into a clustered B+tree or clustered radix tree).

ACKNOWLEDGMENTS

This research is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project 508316729.

REFERENCES

[1] Serge Abiteboul et al. 2005. Diagnosis of asynchronous discrete event systems: datalog to the rescue! *(ACM SIGMOD PODS '05)*. https://doi.org/10.1145/1065167.1065214
[2] Peter Alvaro et al. 2010. Dedalus: datalog in time and space. In *Proceedings of the First International Conference on Datalog Reloaded (Datalog'10)*. https://doi.org/10.1007/978-3-642-24206-9_16
[3] Samuel Arch et al. 2022. Building a Join Optimizer for Soufflé *(LOPSTR 2022)*, Alicia Villanueva (Ed.), Vol. 13474. https://doi.org/10.1007/978-3-031-16767-6_5
[4] Molham Aref et al. 2015. Design and Implementation of the LogicBlox System *(ACM SIGMOD '15)*. https://doi.org/10.1145/2723372.2742796
[5] I. Balbin et al. 1991. Efficient bottom-up computation of queries on stratified databases. *Journal of Logic Programming* 11 (1991). https://doi.org/10.1016/0743-1066(91)90030-S
[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses *(ACM SIGPLAN Conference OOPSLA '09)*. https://doi.org/10.1145/1640089.1640108
[7] S. Ceri, G. Gottlob, and L. Tanca. 1989. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering* (1989). https://doi.org/10.1109/69.43410
[8] Datomic. 2024. Datomic. https://www.datomic.com/ Accessed: May 23, 2024.
[9] Zhiwei Fan et al. 2018. Scaling-Up In-Memory Datalog Processing: Observations and Techniques. *CoRR* (2018). http://arxiv.org/abs/1812.03975
[10] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog*. https://api.semanticscholar.org/CorpusID:169040311
[11] Bernhard Scholz et al. 2016. On fast large-scale program analysis in Datalog *(ACM Compiler Construction 2016)*. https://doi.org/10.1145/2892208.2892226
[12] P. Griffiths Selinger et al. 1979. Access path selection in a relational database management system *(ACM SIGMOD '79)*. https://doi.org/10.1145/582095.582099
[13] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices *(ACM SIGPLAN International Conference on PLDI 2021)*. https://doi.org/10.1145/3453483.3454026
[14] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses *(IEEE/ACM International Conference on ASE '16)*. https://doi.org/10.1145/2970276.2970298
[15] Jiacheng Wu, Jin Wang, and Carlo Zaniolo. 2022. Optimizing Parallel Recursive Datalog Evaluation on Multicore Machines *(ACM SIGMOD '22)*. https://doi.org/10.1145/3514221.3517853