Optimizing Nested Recursive Queries

AMIR SHAIKHHA, University of Edinburgh, United Kingdom
DAN SUCIU, University of Washington, USA
MAXIMILIAN SCHLEICH, RelationalAI, USA
HUNG NGO, RelationalAI, USA

Datalog is a declarative programming language that has gained popularity in various domains due to its simplicity, expressiveness, and efficiency. But "pure" Datalog is limited to monotone queries, and cannot be used in most practical applications. For that reason, newer systems are relaxing the language by allowing non-monotone queries to be freely combined with recursion. But by departing from the elegant fixpoint semantics of pure datalog, these systems often result in inefficient query execution, for example they perform redundant computations, or use redundant storage. In this paper, we propose Temporel, a system that allows recursion to be freely combined with non-monotone operators. Temporel optimizes the program by compiling it into a novel intermediate representation that we call TempoDL. Our experimental results show that our system outperforms a state-of-the-art Datalog engine as well as a vectorized and a compiled in-memory database system for a wide range of applications from machine learning to graph processing.

CCS Concepts: • **Information systems → Query languages**; **Database query processing**; • **Software and its engineering** → *Compilers*; *Constraint and logic languages.*

Additional Key Words and Phrases: Declarative programming, nested recursion, iterative algorithms, demand transformation.

ACM Reference Format:
Amir Shaikhha, Dan Suciu, Maximilian Schleich, and Hung Ngo. 2024. Optimizing Nested Recursive Queries. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 16 (February 2024), 27 pages. https://doi.org/10.1145/3639271

1 INTRODUCTION

Datalog is a declarative programming language that has gained popularity in various domains due to its simplicity, expressiveness, and efficiency. Apart from the conventional usecases in databases, for representing and manipulating large amounts of data efficiently, it has also found applications in declarative networking, artificial intelligence, and machine learning [20, 41, 69]. Additionally, Datalog has been used in program analysis and verification, enabling developers to analyze and verify the high-level [5, 28] and low-level [17, 42, 52] properties of their programs [13, 28, 33].

Most existing Datalog systems only support monotone Datalog programs. Monotone datalog has elegant semantics, in the form of a minimal model, which leads to several desired properties, for example, every monotone datalog program is guaranteed to terminate in polynomial time. In order to use non-monotone operators, the programmer needs to write stratified datalog programs. For example, Soufflé does not allow aggregations inside recursion and one needs to express the program using stratified Datalog. This is sometimes not possible, and other times it results in creating

Authors' addresses: Amir Shaikhha, University of Edinburgh, Edinburgh, United Kingdom, amir.shaikhha@ed.ac.uk; Dan Suciu, University of Washington, Seattle, WA, USA, suciu@cs.washington.edu; Maximilian Schleich, RelationalAI, CA, Berkeley, USA, maximilian.schleich@relational.ai; Hung Ngo, RelationalAI, Berkeley, CA, USA, hung.ngo@relational.ai.

2836-6573/2024/2-ART16 $15.00
https://doi.org/10.1145/3639271

unnecessary intermediate results through recursion followed by negation/aggregation [21]. While some recent theoretical [36, 75] and systems [37, 70, 71] advances have gone beyond the traditional monotone Datalog, they are still limited, and cannot capture important classes of problems in graph processing and machine learning.

A few commercial systems and research projects have adopted a more liberal view of datalog, and, more generally, to programs that extend relational queries with iteration, by allowing recursion and non-monotone operators to be intertwined freely. For example [32, 50] propose extensions of SQL with unrestricted iteration constructs, [31] describes how to compile PL/SQL UDFs with arbitrary iterations into SQL99 recursive queries, while [24, 25] describe a dataflow engine where iteration can be applied freely to relational queries. Startups like Logicblox [4] and RelationalAI [1] allow the free use of recursion and non-monotone operators, thus departing from the elegant minimal model semantics of "pure" datalog. In these systems, the iteration is performed *as given*, using a loop-based operational semantics [32], i.e., the iteration is performed repeatedly, as stated in the program, until some termination condition is satisfied.

Such "free" datalog systems are very powerful, as they often lead to a Turing-complete language, and thus can be used to express any desired problem in graph computation or machine learning. This clearly make these systems desirable in practice. However, their *what-you-write-is-what-you-get* semantics means that the optimizer is limited to firing the queries in exactly the order prescribed by the programmer. Computations performed at iteration t are performed again at iteration $t + 1$, because that is what the semantics dictates. For example, a program as simple as computing the prefix sum of an array $p[i] := p[i - 1] + v[i]$ written in any of these systems will take time $O(n^2)$ instead of $O(n)$, because the entire array p is computed at each iteration. Another common source of inefficiency stems from the fact that users often need only the last value of the iteration, for example they only need $p[n]$, which is the sum of the entire array, and do not need the other intermediate results $p[i]$. Systems restricted to monotone datalog can benefit from semi-naive evaluation [7] and magic set rewriting [8, 43, 44] in order to remove these redundant computations, but these techniques are not available when recursion and non-monotone operators are freely intermixed.

In this paper, we propose Temporel, a system that optimizes general recursive programs, where non-monotone operators such as aggregates and negation are intertwined with recursion; its architecture is described in Sec. 3. We allow recursion and non-monotone operators to be freely intertwined, and call the language FreeDatalog. Temporel converts FreeDatalog into an optimized intermediate representation that consists of nested loops (Sec. 4), which enables two further optimizations: subsumption and temporal elimination (Sec. 5). On one hand, Temporel eliminates the limitations of stratified datalog, by supporting FreeDatalog where recursion and non-monotone operators can be used freely, on the other hand it removes the redundancies inherent in other systems that allow such freedom.

Our starting observation is that programs written using unrestricted iteration often use temporal variables in order control the order in which to apply the relational operators. For example, Fig. 1a shows how a user would typically write a batch-gradient-descent program for linear regression in FreeDatalog; she would write a similar program in any of the other extensions mentioned above. She uses the variable t as a time stamp that allows her to control the order of the recursion and aggregation. We found the use of these temporal variables to be wide-spread in several benchmarks that we discuss in Sec. 6. Some systems, such as Bloom [33] or Dedalus [29], even mandate the use of temporal variables. In FreeDatalog we do not mandate them, but rely on the user to provide them were necessary. For example a monotone program may have no temporal variables, while complex programs may use 1, or 2 or more temporal variables per predicate, in order to simulate nested loops. Temporel starts by performing a static analysis on the program in order to identify the temporal

```
model(t, c, p) :-
  (t = 0), model0(c, p).
predict(t, id, sum(y)) :-
  model(t, c, p), xtrain(id, c, v),
  y = v * p.
gradient(t,c,sum(g)) :-
  ytrain(id,y), predict(t,id,y'),
  xtrain(id,c,v), g=2*(y'-y)*v.
model(t+1, c, p') :-
  model(t, c, p), gradient(t, c, g),
  (t<MAX_ITER), p'=p-(α*g/N).
query(c, p) :- model(last, c, p).
```

(a) The input FreeDatalog program.

Iter #0

t	c	p
0	0	22
0	1	43

Iter #1

t	c	p
0	0	22
0	1	43
1	0	38
1	1	49

...

Iter #10

t	c	p
0	0	22
0	1	43
...	...	...
10	0	88
10	1	99

Iter #0

t	c	p
0	0	22
0	1	43

Iter #1

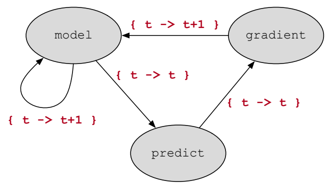
t	c	p
0	0	22
0	1	43
1	0	38
1	1	49

...

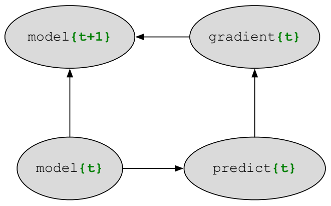
Iter #10

t	c	p
0	0	22
0	1	43
...	...	...
10	0	88
10	1	99

(b) Intermediate values of model in naïve vs. temporal stratified evaluations. The updated elements have a gray background.



(c) Constructed Annotated Program-Dependence Graph (APDG).



(d) The transformed APDG after temporalization.

```
model{0}(t, c, p) := (t = 0), model0(c, p).
for t from 0
  predict{t}(t, id, sum(y)) +=
    model{t}(t, c, p), vtrain(id,c,v,_),
    y = v * p.
  gradient{t}(t, c, sum(g)) +=
    predict{t}(t,id,y'), ytrain(id, y),
    xtrain(id,c,v), g=2*(y'-y)*v.
  model{t+1}(t+1, c, p') +=
    model{t}(t, c, p), gradient{t}(t, c, g),
    (t<MAX_ITER), p'=p-(α*g/N).
end
query(c, p) := model{last}(last, c, p).
return query(c, p).
```

(e) Nested temporal stratified representation in TempoDL.

Fig. 1. The Batch-Gradient Decent (BGD) for a linear regression model.

variables, and checks that the program is *T-stratified*, meaning that it is stratified by the temporal variables; the system returns an error if the program is not T-stratified. Next, the predicates of the program are assigned explicit time stamps, through a process that we call *temporalization*, which essentially creates multiple versions of each predicate. Finally, the program is compiled into a novel intermediate language, TempoDL, which is based on nested loops. Temporel postprocesses TempoDL in order to perform two additional optimizations. *Subsumption* removes the temporal variable when that is implied by the version number, and *temporal elimination* removes the version when the system determines that it suffices to store only the last version.

Example 1.1. We will use as running example a program computing Batch-Gradient Decent (BGD), shown in Figure 1. This formulation is inspired by the previous work [69]. The input relations (called EDBs in datalog) are the following: `xtrain` corresponds to the features, `ytrain` are the labels, and `model0` to initial model parameters. The computed relations (called IDBs in datalog) are `model` that keeps the learned parameters, `gradient` holds the parameters' gradient, and `predict` keeps the predicted targets. Intuitively, the FreeDatalog program performs the standard loop in BGD, for $t = 0, 1, \ldots,$ MAX_ITER, and ensures that the model at iteration $t + 1$ is computed from model and gradient at iteration t (rule 4). A naïve evaluation of this program would recompute the values at time t at all future steps $t + 1, t + 2, \ldots$, as shown on the left of Fig. 1b. Temporel does not evaluate the program naïvely, but instead starts by performing a static analysis of the program where it identifies the temporal variable, and constructs an *annotated program dependency graph* (APDG), Fig. 1c, which represents explicitly how the temporal variable is updated. Next, it performs a temporalization of the program, leading to Fig. 1d, and checks that the program is T-stratified by verifying that the temporalized APDG is acyclic. Finally, Temporel converts the program into TempoDL, which is an intermediate language with explicit loops, Fig. 1e. The explicit loops in TempoDL have a fixed-point semantics and keep track of time stamps; they remove the redundant computations, because at each iteration t only the data values with that time stamp are processed: its execution is illustrated on the right of Fig. 1b. Finally, the value computed at the last timestamp of the loop is accessed using the **`last`** keyword. Further optimizations are described in Section 5.

Several recent projects discuss optimization and evaluation techniques for iterative relational programs: [50] and [32] propose extensions of SQL to overcome the current limitations of WITH RECURSIVE, [30, 31] describe how to compile python programs and PL/SQL programs respectively into WITH RECURSIVE queries, while [36, 71] extend the semantics of datalog to semirings. Our work is orthogonal to these systems: we propose the exploitation of temporal variables and introduce a new intermediate language TempoDL to optimize queries where iteration and non-monotone operators are used freely. In summary, our paper makes the following contributions:

- We present Temporel, a system for optimizing general recursive programs (Section 3). Users write the input programs in FreeDatalog, a language for expressing recursion and non-monotone operators. Temporel uses a novel intermediate language, TempoDL, that supports iterations as first-class citizens to capture different existing recursion evaluation strategies such as naïve evaluation, (generalized) semi-naïve evaluation, and XY-stratification, as well as a novel evaluation proposed in this work.
- We present the compilation of FreeDatalog to TempoDL (Section 4). Temporel accepts a T-stratified Datalog programs, a class of Datalog programs that capture a wide range of iterative algorithms.
- We present the optimizations required to further improve the performance of the translated TempoDL programs, including subsumption, temporal elimination, and code generation (Section 5).
- We experimentally show that Temporel can express programs that are either not representable in existing Datalog and relational database systems or are asymptotically slower in comparison with Temporel (Section 6).

2 BACKGROUND

Datalog. We briefly review Datalog here and refer to [3] for details. A Datalog program consists of a set of rules, where each rule itself consists of a head and a body. The body consists of the conjunction of a set of atoms separated comma (or, equivalently, by $\wedge$), while the head consists of a single atom. Each predicate that occurs in the head of a rule is called an IDB (Intensional Database) predicate, all others are called EDB (Extensional Database) predicates. When multiple rules have the same predicate in the head then they are combined with $\vee$, and we often write this explicitly, for example by replacing the two rules on the left with the rule on the right:

```
h(x,y) :- b1(x,y).
h(x,y) :- b2(x,y).     ⇝   h(x,y) :- b1(x,y) ∨ b2(x,y).
```

The semantics of a datalog program is defined as its minimal model. The *naïve evaluation algorithm* computes the minimal model by computing the fix-point of the Immediate Consequence Operator (ICO): it evaluates all the rules on the current state of the IDBs in order to obtain the next state of the IDBs, and stops when there is no more change. The *semi-naïve evaluation algorithm* improves over the naive algorithm, by avoiding recomputations across iterations. We illustrate these concepts on a classic example.

Example 2.1 (Reachability). Assuming `edge(x, y)` specifies an edge in between nodes x and y in a graph, the reachability is expressed as the following Datalog program:

```
reach(x, y) :- edge(x, y).
reach(x, y) :- reach(x, z) ∧ edge(z, y).
```

The first rule specifies the reachability of two connected nodes and the second rule specifies its transitive closure. The naïve recursive evaluation of the reachability query, starts by $\texttt{reach}_0\texttt{(x, y)} = \emptyset$, and at the iteration t, updates $\texttt{reach}_{t+1}$ based on `edge` and $\texttt{reach}_t$, until we reach a fix-point:

```
reach_0(x, y) = ∅; t = 0;
fixpoint loop
  reach_{t+1}(x, y) = edge(x,y)∨(∃z reach_t(x, z) ∧ edge(z,y))
```

The semi-naïve algorithm keeps track of the difference δ`reach` between two consecutive states:

```
δreach_0(x, y) = edge(x,y); reach(x, y) = ∅; t = 0;
fixpoint loop
  δreach_{t+1}(x, y) =
       (∃z δreach_t(x, z) ∧ edge(z,y)) \ reach_t(x, y)
  reach_{t+1}(x, y) = reach_t(x, y) ∨ δreach_{t+1}(x, y)
```

Negation & Aggregation. Pure Datalog includes neither negation nor aggregations. Datalog is often extended with negation and aggregates, by allowing atoms in the body to be negated and allowing aggregate operators to occur in the head. However, such an extension no longer has a unique minimal model, which creates a problem. One popular solution is to restrict datalog programs to be stratified, defined as follows.

Definition 2.2. The *Predicate Dependence Graph* (PDG) of a datalog program is the graph whose vertices are the IDBs, and whose edges are pairs of IDBs (R_b, R_h) s.t. there exists a rule where R_b occrs in the body and R_h occurs in the head. We say that the edge is negative if R_h contains an aggregate operator, or R_b occurs negated in the body; otherwise we say that the edge is positive. The datalog program is *stratified* if no cycle contains a negative edge.

Several Datalog systems including Soufflé [34] require the program to be stratified. We illustrate with a simple stratified program.

Example 2.3 (Complement of Reachability). The program below computes the complement of the transitive closure of an undirected graph specified by the nodes `node(x)` and edges `edge(x, y)`:

```
reach(x, y) :- edge(x, y).
reach(x, y) :- reach(x, z) ∧ edge(z, y).
comp_reach(x, y) :- node(x),node(y), not(reach(x, y)).
```

This is a stratified Datalog program with two strata: (1) the first two rules compute `reach`, and (2) the last rule computes `comp_reach`. The first stratum is computed using the semi-naïve algorithm, and the second stratum is an anti-join.

Local Stratification. Unfortunately, many applications cannot be written using stratified datalog. For example, the BGD program in Fig. 1a is not stratified, because the IDBs `model`, `predict`, `gradient` are mutually recursive and also involve the aggregate operator **`sum`**. A more general condition is

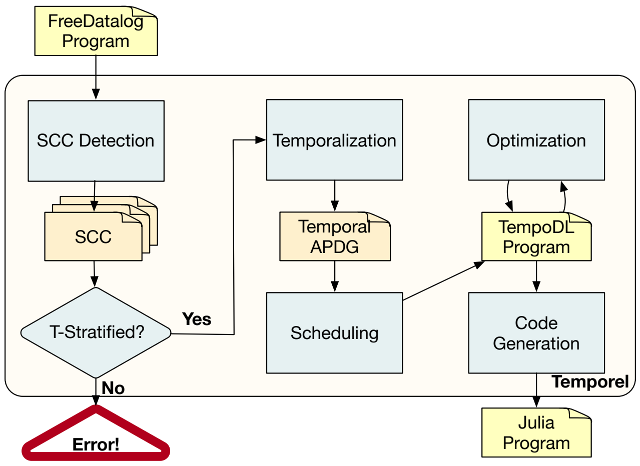


Fig. 2. The high-level workflow of Temporel.

that of a *locally stratified* program, where stratification is checked only at runtime, however, it has been shown that local stratification is undecidable in general [14, 49]. Thus, there has been efforts in identifying subsets of locally stratified programs that are decidable [48, 54].

One such restriction is XY-stratification [74]. All the rules of an XY-stratified program have a distinguished parameter called a temporal parameter `t`, which must be used under certain restrictions. However, XY-stratified programs are restrictive. For example, it is not possible to express nested recursive computations as XY-stratified programs.

Both semi-naïve evaluation and XY-stratification leverage temporal variables; the former uses it as a temporal index, while the latter identifies it in the program. Next, we show how our proposed framework subsumes both.

3 TEMPOREL

In this section, we describe the architecture of our system Temporel, which supports an extension of Datalog called FreeDatalog, where recursion and aggregation can be intertwined. In order to optimize such programs we propose a new intermediate language called TempoDL. Our system compiles FreeDatalog into TempoDL, optimizes the latter, then executes it in Julia.

3.1 Architecture

We show the architecture of Temporel in Figure 2. The light yellow boxes correspond to the input FreeDatalog program, intermediate programs in TempoDL, and the generated code in Julia. The yellow-gold boxes correspond to the data structures required for the compilation process. Finally, the blue boxes correspond to the components of the compiler.

First, the cluster of all mutually recursive IDBs of the input Datalog program is specified by the Strongly Connected Component (SCC) Detector. Then, for each SCC, we test to see if it is T-stratified. If this is not the case, Temporel rejects the program. Otherwise, it compiles the FreeDatalog program to an intermediate TempoDL program through the temporalization and scheduling procedures. After applying optimizations on this program (cf. Section 5), Temporel finally generates a low-level specialized engine. We currently use Julia for our engine, however, one could use C/C++ similar to Soufflé or LLVM similar to several query compilers.

FreeDatalog				
Prog.	p	::=	$r \mid r\ p$	List of rules.
Rule	r	::=	H :- B.	Head and body.
Head	H	::=	$A \mid R(\overline{x}, \overline{\texttt{agg}(y)})$	Access with(/out) aggs.
Body	B	::=	$e \mid e,\ B$	Conjunction of expressions.
Expr.	e	::=	$a\ \theta\ a \mid A$	Comparison (θ), access,
		|	**not** A	negation.
Access	A	::=	$R(\overline{x})$	Rel. access without aggs.
Atom	R	::=	X	Relation name.
Arith.	a	::=	$x \mid c \mid a \diamond a$	Variable, const., bin. ops. ($\diamond$).

TempoDL				
Prog.	p	::=	s **return** A.	Top-level program.
Stat.	s	::=	$r \mid$ **decl** $\overline{R} \mid s\ s$	Rule, decl., list of stmts.,
		|	**for** x **from** $c\ s$ **end**	& for loop.
Rule	r	::=	$H := B. \mid H \mathrel{+}= B.$	Assignment and update.
Atom	R	::=	$X \mid X\{\overline{t}\}$	Rel. name with(/out) strata.
Temp.	t	::=	$x \mid c \mid x{+}c \mid$ **last**	Variable, const., offset, last.
Arith.	a	::=	... | **last**	Arith. exprs. and last.

Fig. 3. The grammar of FreeDatalog and TempoDL.

3.2 FreeDatalog

FreeDatalog is standard Datalog extended with negation and aggregates. We show the grammar in Fig. 3. An example of FreeDatalog is the BGD program in Fig. 1a.

3.3 TempoDL

In this paper, we introduce a novel intermediate language called TempoDL, where the recursive evaluation strategy in FreeDatalog is made explicit by two main additions. First, TempoDL consists of for-loop iterations that implement directly a fixpoint computation. Second, time-indices are first-class citizens of TempoDL. Every iteration exposes a time index, and each nested iteration has a sequence of time indices that corresponds to a nested stratum. To avoid repeated computations, TempoDL uses versioned IDBs, by annotating them with a time index: a versioned IDB `idb` is represented as `idb{t}`. Versioning allows TempoDL to avoid redundant computations, by restricting each computation to only the necessary version (stratum).

Example 3.1 (Reachability in TempoDL). Figure 4 shows the representation of the previously mentioned evaluation strategies of Reachability in TempoDL (Example 2.1). Figure 4a corresponds to the naïve evaluation. Figure 4b shows the semi-naïvely evaluated using a **for**-loop updating the IDB `reach` and its changes δ`reach`.

4 COMPILING FREEDATALOG TO TEMPODL

We describe in this section the main part of Temporel: compiling a FreeDatalog program into the intermediate language TempoDL. This consists of four phases. First, Temporel breaks the input FreeDatalog program into a set of SCCs, each one specifying a set of mutually recursive IDBs. Then for each SCC, it identifies the temporal variables and checks if the program is T-stratified; if not, then it returns an error message. Afterwards, it rewrites the FreeDatalog program into a

```
reach{0}(x, y) := ∅.
for t from 0
  reach{t+1}(x, y) := edge(x, y).
  reach{t+1}(x, y) += reach{t}(x, z}, edge(z, y).
end
return reach{last}(x, y).
```

(a) Naïve evaluation.

```
δreach{0}(x, y) := edge(x, y). reach{0}(x, y) := ∅.
for t from 0
  δreach{t+1}(x, y) += reach{t}(x, z), edge(z, y),
                               not(reach{t}(x, y)).
  reach{t+1}(x, y) := reach{t}(x, y).
  reach{t+1}(x, y) += δreach{t}(x, y}.
end
return reach{last}(x, y).
```

(b) Semi-Naïve evaluation.

Fig. 4. Evaluation strategies of Reachability expressed in TempoDL.

```
1: function COMPILE($FreeDatalogProg$)
2:     $SCCs \leftarrow$ SCCDETECTION($FreeDatalogProg$)
3:     $TempoDLProg \leftarrow$ ""
4:     for $SCC \leftarrow SCCs$ do
5:         $TempAttrs \leftarrow$ CANDIDATETEMPORALATTRS($SCC$)
6:         $TempVars, SCC \leftarrow$ NORMALIZESCC($SCC, TempAttrs$)
7:         $PDG \leftarrow$ EXTRACTPDG($SCC$)
8:         $APDG \leftarrow$ ANNOTATEPDG($SCC, PDG, TempVars$)
9:         $TemporalTree \leftarrow$ TEMPORALPLAN($APDG$)
10:        for $Var \leftarrow$ PREORDER($TemporalTree$) do
11:            $APDG \leftarrow$ TEMPORALIZE($APDG, Var$)
12:        end for
13:        $Exp \leftarrow$ SCHEDULE($SCC, APDG, TemporalTree$)
14:        $TempoDLProg \leftarrow$ "$\$TempoDLProg$; $\$Exp$"
15:    end for
16: end function
```

Algorithm 1. The compilation of FreeDatalog to TempoDL.

temporalized program, where the temporal strata are annotated with symbolic variables. Finally, it compiles the temporalized program into TempoDL.

4.1 SCC Detection

As the first step, the rules are broken into a sequence of Strongly Connected Components (SCC) of the PDG (Def. 2.2). Each SCC involves a set of rules that define mutually recursive IDBs. The SCC Detection component is responsible for creating a topological order for the SCCs. The next stages of compilation are applied over each SCC one-by-one.

4.2 The T-Stratification Test

When users combine recursion with non-monotone operators like summation or negation, they usually control the order of operations by using temporal variables. The BDG program in Example 1.1 is a typical illustration: the variable t in Fig. 1a is a temporal variable, which ensures that the values computed by **sum** are used only at the next time stamp. Temporel detects automatically the temporal variables, then checks that the program is *T-stratified* (defined below); if the programs is not T-stratified, then Temporel returns an error message.

If R is IDB relation, then we denote by $R.1, R.2, \ldots$ its attributes. To detect temporal variables, Temporel starts by constructing the *Attributes Graph*, AG, whose nodes are the attributes of the IDB relations, and whose edges are pairs $(R_b.i, R_h.j)$ for any rule where R_b occurs in the body and has some variable `x` on position i, and R_h occurs in the head with an expression `x+c` on attribute j, where c is a constant, possibly 0. A *T-equivalence class* is a Strongly Connected Component in the AG.

Example 4.1. Continuing Example 1.1, we have the following T-equivalence classes:

- `[gradient.1, predict.1, model.1]`: The variable `t` in the first rule connects `gradient.1` with `predict.1` and in the second rule connects `predict.1` with `model.1`. Furthermore, this variable connects `model.1` and `gradient.1` in the third rule as it is a constant offset (`t+1`).
- `[gradient.2, model.2]`: The variable `c` in the third rule connects `gradient.2` with `model.2`.

Note that there is no T-equivalence class related to the variable `p`. This is because in the third rule, `model.3` is connected with itself through a complicated arithmetic expression.

Definition 4.2 (Candidate Temporal Attributes). The attributes of each IDB that are in the same T-equivalence class are called candidate temporal attributes if at least in one of the rules there is a constant offset (e.g., `t+c`, for `c > 0`).

Identifying Temporal Variables. Temporel uses a union-find data structure to construct the disjoint sets of T-equivalence classes. Afterwards, Temporel associates a distinct temporal variable to each T-equivalence class; this is achieved by renaming the occurrences of candidate temporal attributes in an SCC into a single name (by calling NormalizeSCC in Algorithm 1 and returning $TempVars$).

In our running example, the variable `t` is the candidate temporal variable in the first three rules. However, `c` is not a candidate temporal variable as there is no constant offset in any of the rules.

Definition 4.3 (Annotated Predicate Dependence Graph (APDG)). An APDG is a graph is a PDG (Def. 2.2) $G =< V, E >$, where each edge is annotated with a mapping $\{t \longrightarrow t + \delta\}$, where t specifies a candidate temporal variable, and δ specifies its increment in the corresponding rule.

In our running example, the corresponding APDG is shown in Figure 1c. For example, the label `{t -> t+1}` on the edge between `gradient` and `model` corresponds to the fourth rule; `gradient(t,...)` appears in the body whereas `model(t+1, ...)` is the head.

Definition 4.4 (T-Stratified Program). A Datalog program is T-stratified if for any cycle in the APDG graph there exists an edge labeled $t \longrightarrow t + \delta$, with $\delta > 0$.

In the APGD of the BGD program (e.g., Figure 1c), there are two cycles, and for both cycles, we have the temporal variable `t` and (1) all the edges include `t`, and (2) there is an edge with `{ t -> t+1 }`. Thus, this program is T-Stratified.

```
function TEMPORALIZE($APDG$, $t$)
    $APDG' \leftarrow$ EMPTYGRAPH
    $Edges \leftarrow$ EXTRACTEDGES($APDG$)
    for $E \leftarrow Edges$ do
        if $t$ -> $t + \delta \in E.label$ then
            $E'.label \leftarrow E.label - \{t$ -> $t + \delta\}$
            if $\delta = 0 \in E.label$ then
                $E'.src \leftarrow E.src$ {t}
                $E'.dst \leftarrow E.dst$ {t}
            else
                $E'.src \leftarrow E.src$ {t}
                $E'.dst \leftarrow E.dst$ {t+1}
            end if
            $APDG' \leftarrow$ ADDEDGE($APDG'$, $E'$)
        else
            $APDG' \leftarrow$ ADDEDGE($APDG'$, $E$)
        end if
    end for
end function
```
Algorithm 2. Temporalization Algorithm.

4.3 Temporalization

In the next stage, we need to associate a stratum for each recursive IDB. This is achieved in a process called temporalization. In this process, the nodes in the APDG are transformed into ones with strata, and the temporal increment labels on the edges are removed.

Algorithm 2 shows the process for transforming an input APDG for a given temporal variable. In line 3, we extract the list of edges. In the case that the label does not contain the temporal variable (lines 15-16) the edge is unchanged. Otherwise, we remove the associated temporal increment from the label (line 6). If the delta value is zero (lines 7-9) we use `{t}` as the stratum for both IDBs. Otherwise, if the delta value is 1 (line 10-12) we use `{t+1}` for the IDB appearing in the head and `{t}` for the IDB in the body.

Example 4.5 (BGD, Cont.). Figure 1d shows the temporalized APDG of our running example (Example 1.1). The edge from `gradient` to `model` is transformed into an edge from `gradient{t}` to `model{t+1}`, and the self-loop on `model` is transformed into an edge from `model{t}` to `model{t+1}`. The other two edges from `model` to `predict` and from `predict` to `gradient` are transformed into two edges from `model{t}` to `predict{t}` and from `predict{t}` to `gradient{t}`, respectively.

In the case of nested recursions, Algorithm 2 is invoked multiple times by a pre-order traversal over the temporal variables. At each invocation, the APDG is partially temporalized for the edges with the relevant temporal increments.

Example 4.6 (BGD with Backtracking Search (BGD-BTS)). The BGD algorithm shown earlier assumes a fixed learning rate. One can improve this algorithm by deciding on how much to move towards the direction of gradient (Figure 5a). This requires a nested loop responsible for doing a search for a learning rate that results in better cost than the current prediction. Thus, one needs to compute the cost both in the outer loop (`cost`) and the inner loop (`cost_inner`), and the program uses two temporal variables for this purpose, `t` and `k`.

Figure 6 shows the temporalized APDG of the aforementioned program. At first, the TEMPORALIZE function is invoked for the temporal variable `t`. At this stage, all the nodes are rewritten to include a stratum of `t` (either `{t}` or `{t+1}`) and the temporal increments of `t` are removed from their

```
model(t, c, p) :- (t = 0), model0(c, p).
predict(t, id, sum(y)) :- model(t, c, p), xtrain(id, c, v), y = v * p.
gradient(t,c,sum(g)) :- ytrain(id,y), predict(t,id,y'), xtrain(id,c,v), g=2*(y'-y)*v.
cost(t, sum(f)) :- ytrain(id,y), predict(t,id,y'), f=(y'-y)^2.
alpha(t, k, lr) :- cost(t, x), k = 0, lr = α.
alpha(t, k+1, lr) :- cost(t,f1), cost_inner(t,k,f2), f1>=f2, k<MAX_INNER,
  alpha(t, k, lr2), lr=lr2*0.9.
model_inner(t,k,c,p') :- model(t,c,p), alpha(t,k,a), gradient(t, c, g),
  (t<MAX_ITER), p'=p-(α*g/N).
predict_inner(t,k,id,sum(y)) :- model_inner(t,k,c,p), xtrain(id, c, v), (y = v * p).
cost_inner(t, k, sum(f)) :- ytrain(id, y), predict_inner(t, k, id, yp), f = (yp-y)^2.
alpha_final(t, lr) :- alpha(t, last, lr).
model(t+1, c, p') :- model(t,c,p), alpha_final(t,lr), gradient(t, c, g),
  (t<MAX_ITER), p'=p-(lr*g/N).
query(c, p) :- model(last, c, p).
```

(a) The input FreeDatalog program.

```
model{0}(t, c, p) := (t = 0), model0(c, p).
for t from 0
  predict{t}(t, id, sum(y)) += vtrain(id, c, v, _), model{t}(t, c, p), y = v * p.
  gradient{t}(t, c, sum(g)) += ytrain(id, y), predict{t}(t, id, y'),
    xtrain(id, c, v), g = 2*(y'-y)*v.
  cost{t}(t, sum(f)) += ytrain(id,y), predict{t}(t,id,y'), f=(y'-y)^2.
  alpha{t, 0}(t, k, lr) += cost{t}(t, x), k = 0, lr = α.
  for k from 0
    model_inner{t,k}(t,k,c,p') += model{t}(t,c,p), alpha{t,k}(t,k,a),
      gradient{t}(t, c, g), (t<MAX_ITER), p'=p-(α*g/N).
    predict_inner{t,k}(t,k,id,sum(y)) += model_inner{t,k}(t,k,c,p),
      xtrain(id, c, v), (y = v * p).
    cost_inner{t,k}(t, k, sum(f)) += ytrain(id, y),
      predict_inner{t,k}(t, k, id, yp), f = (yp-y)^2.
    alpha{t,k+1}(t,k+1,lr) += cost{t}(t,f1), cost_inner{t,k}(t,k,f2), f1>=f2,
      k<MAX_INNER, alpha{t,k}(t,k,lr2), lr=lr2*0.9.
  end
  alpha_final{t}(t, lr) += alpha{t,last}(t, last, lr).
  model{t+1}(t+1, c, p') += model{t}(t,c,p), alpha_final{t}(t,lr),
    gradient{t}(t, c, g), (t<MAX_ITER), p'=p-(lr*g/N).
end
query(c, p) := model{last}(last, c, p).
return query(c, p).
```

(b) Nested temporal stratified representation in TempoDL.

Fig. 5. The compilation procedure of Batch-Gradient Decent (BGD) with backtracking search.

connecting edges. Next, the same function is invoked for the temporal variable k. At this stage, only the dark gray nodes and their connecting edges are transformed.

Theorem 4.7. *If the Datalog program is T-stratified, then its temporalized APDG is acyclic.*

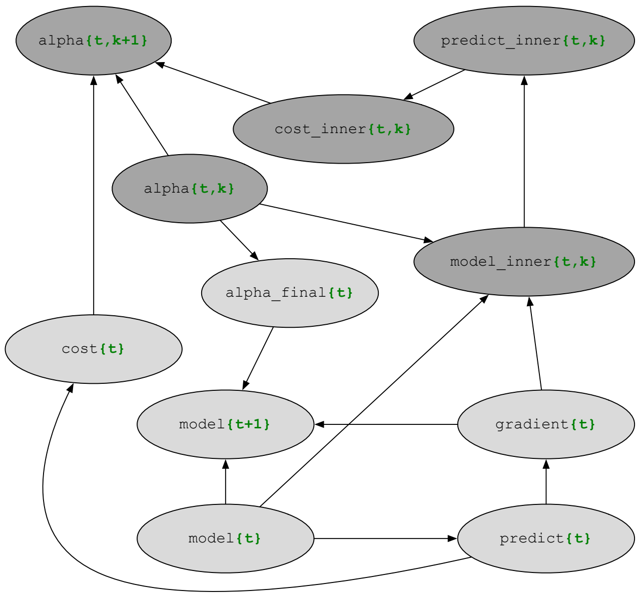


Fig. 6. Temporalized APDG of BGD with backtracking search.

Proof. We prove the previous theorem by contradiction. Let us assume that temporalized APDG has a cycle. As Algorithm 2 does not increase the number of cycles, the cycle existing on the output of this algorithm can be traced back to a cycle from the input APDG. By the definition of a T-stratifiable program, this cycle contains at least one edge with label $\{t\text{->}t+\delta\}$ with $\delta > 0$. However, this cycle is already broken by Algorithm 2 applied to the associated temporal variable. □

The previous theorem results in an efficient way of scheduling the tempralized APDG which is presented next.

4.4 Scheduling

As the final stage, the scheduling process transforms the temporalized APDG into a TempoDL expression that exploits the nested strata information. Rather than following the naïve evaluation that involves many recomputations, the temporal stratified evaluation carefully keeps track of the previous versions and reuses the values stored in them.

Algorithm 3 shows the process of generating TempoDL programs for a given SCC of rules. The function ScheduleSingle returns the following three. First, it returns the statements that are invariant to the loop and thus can be scheduled before executing the loop. Second, the statements that are dependent on the loop and thus required to be put inside it. The ScheduleSingle function uses the dependency information in the APDG in order to schedule the inner rules. Finally, it returns the start value for the temporal variable. In the case of a single temporal variable, the generated TempoDL program simply glues together these values (lines 18-21).

Example 4.8 (BGD, Cont.). The TempoDL representation of the BGD program is shown in Figure 1e. The program consists of two SCCs, and here we focus on the scheduling for the first SCC that is T-stratifiable. The first rule is invariant to the loop and is thus generated outside the loop. The rest of the three rules are scheduled inside the loop. The three rules are scheduled based on the

```
function SCHEDULE($SCC$, $APDG$, $TemporalTree$)
    $Var \leftarrow TemporalTree.var$
    $Outer, Inner, Start \leftarrow$ SCHEDULESINGLE($SCC$, $APDG$, $Var$)
    if ISLEAF($TemporalTree$) then
        $LoopBody \leftarrow Inner$
    else
        $LoopBody \leftarrow$ ""
        for $SubTree \leftarrow TemporalTree$ do
            $SCC', APDG' \leftarrow$ FILTER($SCC, APDG, SubTree$)
            $InnerLoop \leftarrow$ SCHEDULE($SCC', APDG', SubTree$)
            $PreLoop \leftarrow$ DEPENDENTSTATEMENTS($Inner, InnerLoop$)
            $LoopBody \leftarrow$ "$\$LoopBody$
                               $\$PreLoop$
                               $\$InnerLoop$"
            $Inner \leftarrow$ REMOVESTATEMENTS($Inner, PreLoop$)
            $Inner \leftarrow$ REMOVESTATEMENTS($Inner, InnerLoop$)
        end for
        $LoopBody \leftarrow$ "$\$LoopBody$
                           $\$Inner$"
    end if
    $Exp \leftarrow$ "$\$Outer$
             for $\$Var$ from $\$Start$"
                   $\$LoopBody$
             end"
end function
```

Algorithm 3. The scheduling algorithm.

dependencies of their heads: `predict{t} -> gradient{t} -> model{t+1}`. Note that all statements inside the loop are updating the head rather than replacing it.

Let us consider the case of a hierarchy of temporal variables, that corresponds to nested loops. We need to invoke the scheduling algorithm for each of the subtrees recursively. For each one, we need only to consider the relevant set of rules (*SCC′*) and APDG (*APDG′*). After computing the expression for the inner loop, we need to compute the list statements of the outer loop that current inner loop is dependent on (*PreLoop*). Afterwards, we append the dependent statements as well as the inner loop statement to the current list of statements (*LoopBody*) and update the remaining list of statements. Finally, once we are done with all the children temporal variables, we append the remaining statements to the inner loops statements.

Example 4.9 (BGD-BTS, Cont.). Figure 5b shows the scheduled TempoDL expression for BGD-BTS. The temporal variables form a tree structure with `t` as the root and `k` as the leaf. First, SCHEDULESINGLE is called for the temporal variable `t` returning the rule with `model{0}` as the outer statement, and the rest of the rules as the inner statements. Then, it goes to the else branch iterating the subtree, i.e., the temporal variable `k`. It only considers the dark-gray sub-APDG in Figure 6 and its associated SCC (*APDG′* and *SCC′*) when recursively invoking the scheduling algorithm for `k`. This results in the inner loop and its preceding statement with the head `alpha{t,0}` (*InnerLoop*). Afterwards, among the transformed statements of the outer loop, the ones on which the inner loop is dependent are assigned to *PreLoop*, which includes the rules with `predict{t}`, `gradient{t}`, and `cost{t}` as the head. These statements are put before the inner loop statements based on their dependencies extracted from the temporalized APDG, and are appended to *LoopBody*, which was

```
model{0}(c, p) := model0(c, p).
for t from 0
  predict{t}(id, sum(y)) := vtrain(id, c, v, _),
    model{t}(c, p), y = v * p.
  gradient{t}(c, sum(g)) := ytrain(id, y),
    predict{t}(id, y'), xtrain(id, c, v), g = 2*(y'-y)*v.
  model{t+1}(c, p') := model{t}(c, p),
    gradient{t}(c, g), t<MAX_ITER, p' = p-(α*g/N).
end
query(c, p) := model{last}(c, p).
return query(c, p).
```

(a) The TempoDL program after subsumption.

```
model'(c, p) := model0(c, p).
for t from 0
  model(c, p) := model'(c, p).
  predict(id, sum(y)) := vtrain(id,c,v,_), model(c,p), y=v*p.
  gradient(c, sum(g)) := ytrain(id, y),
    predict(id, y'), xtrain(id, c, v), g = 2*(y'-y)*v.
  model'(c, p') := model(c, p),
    gradient(c, g), t<MAX_ITER, p' = p-(α*g/N).
end
query(c, p) := model'(c, p).
return query(c, p).
```

(b) The TempoDL program after temporal elimination.

Fig. 7. The optimization of the BGD example.

empty initially and are removed from the list of the statements of the body of the outer loop. Finally, the remaining statements of the body of the outer loop (with the heads `alpha_final{t}` and `model{t+1}`) are appended after the inner loop.

5 OPTIMIZATIONS

Up to now, we have seen the process of translating FreeDatalog programs (with implicit/inherent iterations) to TempoDL programs (with **for** loops). In this section, we observe how we can optimize further the generated TempoDL programs.

5.1 Subsumption

The generated TempoDL expression appends the result of intermediate strata to the final result. However, in many cases, we are only *directly* interested in the result computed in the last stratum; the intermediate results are only *indirectly* required to compute the last result.

In such cases, we can leverage the fact that the result of each iteration *subsumes* [38] its previous iterations. This has two implications. First, there is no more need to keep track of the stratum as an attribute in the relation, i.e., one can push the projection in the recursion. Second, the result of each iteration replaces the result of its prior ones.

Performing subsumption involves an analysis phase followed by a transformation. The analysis pass performs a backward analysis on the TempoDL program to detect the attributes and the related

IDBs that we demand their **last** value. If there is nothing apart from the **last** value, the analysis pass marks the IDBs and the temporal attribute as the candidate for subsumption.

The transformation pass removes the temporal attribute(s) from the candidate IDBs; inside the loop the occurrences of the temporal variable is removed and outside the loop the occurrences of the base case constant and **last** are eliminated. Furthermore, the first instance of update rules in the loops are substituted by a replacement. This is because we are no longer interested in appending all intermediate strata to the final result; instead we are only interested in the result of the last stratum.

Example 5.1 (BGD, Cont.). Figure 7a shows the result of applying subsumption to TempoDL representation of the BGD example. Note that the intermediate IDBs now have an arity of 2 instead of 3. Furthermore, the rules in the body of the loop are replacement (`:=`) instead of addition (+=).

5.2 Temporal Elimination

Even though with subsumption we removed the need to keep track of the intermediate results at the IDB level, we still keep all the intermediate levels at the stratum level. In many cases, each iteration only needs two versions: the result from the previous iteration and the result after this iteration.

The temporal elimination transformation removes the intermediate strata of IDBs by replacing each IDB with two versions: its old version (with the same name, e.g., `idb`) and its new version (the primed name, e.g., `idb'`). The transformation is as follows. For the rules inside the loop, each occurrence of `idb{t}` is replaced with `idb` and `idb{t+1}` with `idb'`. In the rules outside the loop the IDBs (`idb{basecase}` and `idb{last}`) are replaced with `idb'`. Finally, the IDBs that appear with next stratum in the head (e.g., `idb{t+1}`), we add a replace statement to copy the content of the IDB computed in the previous iteration (`idb := idb'`).

Example 5.2 (BGD, Cont.). The result of applying temporal elimination to the BGD example is shown in Figure 7b. All the occurrences of `model{t}`, `predict{t}`, and `gradient{t}` are replaced with `model`, `predict`, and `gradient`, respectively. The rest of the cases for `model` are replaced with `model'`. Finally, as the first statement of the loop, we copy the content computed from the previous iterations (`model := model'`).

5.3 Code Generation

As the final step, we generate a specialized engine from the optimized TempoDL program. This involves a two-stage process. First, the TempoDL program is compiled into SDQL [59, 61], a recently introduced intermediate representation for query processing. Then, the SDQL expression is compiled down to low-level Julia code [56].

To handle **for** loops in TempoDL, SDQL is extended with temporal fix-point recursion. This additional construct performs a **for**-loop-like iteration where the termination is achieved when the inner IDBs involved in the loop have reached a fix-point.

Each rule is first converted into a physical query plan. Similar to Soufflé we currently do not consider changing the join order and rely on the order specified by the programmer. However, Temporel uses the primary-key/foreign-key information to leverage efficient physical query operator implementations. Then, the query plan is compiled to SDQL by using a push-based approach [47, 60].

6 EXPERIMENTS

In this section, we empirically evaluate the performance of Temporel. More specifically we answer the following research questions:

Table 1. The datasets used in the experiments

Name	Parameters	Description
DS-M-N	M: # of elements N: # of features	Training dataset for machine learning
PNTS-N	N: # of points	Random 2D points
HOSP-N	N: # of hospitals & residents	Residents/Hospital preference data
RMAT-N	N: # of nodes $10N$: # of edges	Synthetic graphs generated by [6]
VECT-N	N: # of elements	Vector of numbers

- How well does Temporel work in comparison with the state-of-the-art systems on various workloads?
- How much is the benefit of temporal stratification over standard recursive evaluation strategies for iterative programs?
- What is the impact of each individual optimization for TempoDL programs?

6.1 Experimental Setup

We conducted our experiments on an AWS t2.2xlarge instance with 8 vCPUs, 32 GBs of RAM, and Ubuntu 22.04 LTS. Temporel runs using Julia 1.8.5. As the competitors, we consider Datalog engines as well as database systems. For the former, we consider the Soufflé engine, while for the latter, we consider the DuckDB and HyPer systems. There are many Datalog systems including Soufflé [34], SociaLite [58], Myria [68], the DeALS family of systems (DeALS [65], BigDatalog [64], and RaDlog [27]), and RecStep [21]. We consider Soufflé (version 2.3) because of its state-of-the-art performance for stratified Datalog programs. RecStep and the DeALS family have efficient support for aggregates in recursion in comparison with previous research [64]. However, neither RaDlog, as the open source representative of the DeALS family, nor RecStep supported any of our workloads.

Most DBMSes with support for recursion (e.g., HyPer and Postgres), only expect a single recursive relation. DuckDB supports a wider range by inlining mutual recursive functions. We use DuckDB version 0.8.1, which supports nested recursion as well as *LIMIT* inside a recursive query, a feature missing from prior versions and necessary for the workloads that use the `choice` construct. We also compare it against HyPer, an in-memory DBMS that, similarly to Souffle, employs code generation [47, 62, 63]. We use Tableau's publicly available HyPer API version 0.0.17537. However, its support for recursion is more limited than DuckDB and thus supports a subset of our workloads. We do not use other database systems such as PostgreSQL, because they are more limited than DuckDB for dealing with mutual recursion. Other previous work [19, 30, 31] are orthogonal as they provide source-to-source translation from PL/SQL and Python UDFs to SQL code and use PostgreSQL as the backend.

6.2 Workloads

We consider nine workloads: (1) BGD for linear regression (BGD-LR), (2) BGD with backtracking search for linear regression (BGD-BTS-LR), (3) the stable matching problem (SMP), (4) prim's algorithm for computing the minimum spanning tree (MST-PRIM), (5) computing window sums (WIN-SUM), (6) computing window z-scores (WIN-ZSCORE), (7) k-means clustering (K-MEANS), (8) the page-rank algorithm, and (9) computing the graph diameter (GRAPH-DIAM). We have already covered the first two in the previous sections. Below we present the remaining applications.

```
hprefers(h,s1,s2) :- hosp(h),stud(s1),
  stud(s2),hpref(h,s1,r1),
  hpref(h,s2,r2), r1 < r2.
hpairs(t,h,s) :- (t=0), (h=1), (s=1).
sfree(t, s) :- (t = 0), stud(s).
hfree(t, h) :- (t = 0), hosp(h).
next_stud(t, min(s)) :- sfree(t, s).
next_hfree_rank(t, min(r)) :-
  next_stud(t,s),hfree(t,h),
  spref(s, h, r).
next_hpref_rank(t, min(r)) :-
  next_stud(t, s1), hpairs(t, h, s2),
  hprefers(h, s1, s2), spref(s1,h,r).
next_hosp_rank(t, r) :-
  next_hfree_rank(t,r1),
  next_hpref_rank(t,r2), r=min(r1,r2).
next_hosp_rank(t, r1) :-
  next_hfree_rank(t, r1),
  not(next_hpref_rank(t, r2)).
next_hosp_rank(t, r2) :-
  next_hpref_rank(t, r2),
  not(next_hfree_rank(t, r1)).
next_hosp_id(t, h) :- next_stud(t, s),
  next_hosp_rank(t, r), spref(s,h,r).
hpairs(t+1, h, s) :- hpairs(t, h, s),
  not(next_hosp_id(t, h)).
hpairs(t+1, h, s) :- next_stud(t, s),
  next_hosp_id(t, h).
query(h, s) :- hpairs(last, h, s).
```

(a) Stable Matching Problem

```
uedge(x, y, w) :- edge(x, y, w).
uedge(x, y, w) :- edge(y, x, w).
min_edge_init(min(w)) :- uedge(x, y, w).
tree_edge(t,x,y) :- t=0, min_edge_init(w),
  uedge(x, y, w), choose((), (x, y)).
tree_node(t, x) :- tree_edge(t, x, y).
tree_node(t, x) :- tree_edge(t, y, x).
min_edge_weight(t,min(w)) :- uedge(x,y,w),
  tree_node(t, x), not(tree_node(t, y)).
min_edge(t, x, y) :- min_edge_weight(t, w),
  uedge(x, y, w), tree_node(t, x),
  not(tree_node(t, y)), choose((), (x, y)).
tree_edge(t+1, x, y) :- tree_edge(t, x, y).
tree_edge(t+1, x, y) :- min_edge(t, x, y).
query(x, y) :- tree_edge(last, x, y).
```

(b) Prim's algorithm for MST.

```
winsum(i, a) :- i = 0, a = 0.
winsum(i+1, s) :- i>=0, i<W, winsum(i,a'),
  vec(i+1,a), s=a+a'.
winsum(i+1, s) :- i>=W, winsum(i, a'),
  vec(i+1,a1), vec(i-W+1,a2), s=a'+a1-a2.
winsumsq(i, a) :- i = 0, a = 0.
winsumsq(i+1, s) :- (i >= 0), (i < W),
  winsumsq(i, a'), vec(i+1, a), s = a+a'.
winsumsq(i+1, s) :- (i>=W),winsumsq(i,a'),
  vec(i+1,a1),vec(i-W+1,a2),s=a'+a1^2-a2^2.
winzscore(i,r) :- winsum(i,w),winsumsq(i,s),
  vec(i,v), g=(s-w*w/W)/W,r=sqrt((v-w/W)/g).
```

(c) Window Z-Score (W is the window size).

Fig. 8. Applications used in the experiments.

Stable Matching Problem (SMP). This problem has many applications including matching resident students to hospitals, job market matching, resource allocation, and task assignment. We consider the EDBs `stud(s)` and `hosp(h)` showing the IDs of students and hospitals, and the EDBs `spref(s, h, r)` and `hpref(h, s, r)` showing the ranking of a particular hospital for a student and vice versa. This program (cf. Figure 8a) consists of two SCCs: the first one is non-recursive, and the second one is T-stratified.

Minimum Spanning Tree (MST-PRIM). Computing the spanning tree of a graph with minimum sum of weights has many applications in network design, circuit design, and data mining. For an input undirected weighted graph specified by the EDB `edge(x, y, w)` the Datalog program of Figure 8b uses the Prim's algorithm to compute its MST. This program is T-stratified.

Window Sum (WIN-SUM). The window sum of elements of a vector by window size specified by the parameter `W` is presented in the top part of Figure 8c. This program is T-stratified.

Window Z-score (WIN-ZSCORE). The rolling (window) Z-score is a statistical technique with applications in financial analysis, anomaly detection, and time series analysis. The FreeDatalog

Table 2. Performance comparison of different systems on various recursive programs. TO: Timeout after 15 minutes, NS: Not Supported, NP: Not Possible (to express the stratified version).

Query	Dataset	Temporel	Soufflé	DuckDB	HyPer
BGD-LR	DS-8-1K	< 0.1 s	NP	0.1 s	NS
	DS-8-8K	0.1 s	NP	0.6 s	NS
	DS-8-64K	1.2 s	NP	5.6 s	NS
	DS-8-512K	15.4 s	NP	50.3 s	NS
	DS-8-4M	208.5 s	NP	421.8 s	NS
BGD-BTS-LR	DS-8-1K	< 0.1 s	NP	0.8 s	NS
	DS-8-8K	0.1 s	NP	5.1 s	NS
	DS-8-64K	2.1 s	NP	41.1 s	NS
	DS-8-512K	23.8 s	NP	341.7 s	NS
	DS-8-4M	333.5 s	NP	TO	NS
K-MEANS	PNTS-1K	< 0.1 s	NP	0.3 s	0.1 s
	PNTS-8K	0.2 s	NP	1.5 s	0.6 s
	PNTS-64K	3.4 s	NP	10.8 s	7.5 s
	PNTS-512K	32.0 s	NP	110.6 s	64.8 s
	PNTS-4M	898.1 s	NP	TO	TO
SMP	HOSP-16	< 0.1 s	NP	14.3 s	NS
	HOSP-32	< 0.1 s	NP	101.2 s	NS
	HOSP-64	0.4 s	NP	TO	NS
	HOSP-128	6.2 s	NP	TO	NS
	HOSP-256	93.3 s	NP	TO	NS
PAGE-RANK	RMAT-1K	0.8 s	NP	0.9 s	0.4 s
	RMAT-2K	1.8 s	NP	1.9 s	0.8 s
	RMAT-4K	4.4 s	NP	4.0 s	1.8 s
	RMAT-8K	9.8 s	NP	10.2 s	4.2 s
	RMAT-16K	25.1 s	NP	21.6 s	11.2 s
MST-PRIM	RMAT-128	< 0.1 s	5.1 s	0.6 s	< 0.1 s
	RMAT-256	0.4 s	61.0 s	3.3 s	0.3 s
	RMAT-512	1.6 s	750.6 s	19.8 s	2.4 s
	RMAT-1K	7.7 s	TO	133.2 s	19.7 s
	RMAT-2K	31.8 s	TO	TO	160.8 s
GRAPH-DIAM	RMAT-256	0.4 s	NP	1.5 s	0.6 s
	RMAT-512	2.5 s	NP	12.4 s	6.0 s
	RMAT-1K	11.3 s	NP	126.4 s	81.4 s
	RMAT-2K	68.0 s	NP	TO	TO
	RMAT-4K	370.3 s	NP	TO	TO
WIN-SUM (W = 30)	VECT-512	< 0.1 s	< 0.1 s	0.3 s	< 0.1 s
	VECT-2K	< 0.1 s	< 0.1 s	1.4 s	< 0.1 s
	VECT-8K	0.4 s	1.3 s	10.1 s	0.4 s
	VECT-32K	6.3 s	22.7 s	104.8 s	7.8 s
	VECT-128K	92.8 s	350.1 s	TO	131.4 s
WIN-ZSCORE (W = 10)	VECT-512	< 0.1 s	< 0.1 s	0.6 s	< 0.1 s
	VECT-2K	< 0.1 s	0.1 s	3.0 s	< 0.1 s
	VECT-8K	0.8 s	2.8 s	20.7 s	0.9 s
	VECT-32K	11.9 s	45.2 s	209.0 s	15.5 s
	VECT-128K	180.1 s	720.3 s	TO	265.2 s
WIN-ZSCORE (W = 30)	VECT-512	< 0.1 s	< 0.1 s	0.6 s	< 0.1 s
	VECT-2K	< 0.1 s	0.1 s	3.0 s	< 0.1 s
	VECT-8K	0.8 s	2.8 s	20.6 s	0.9 s
	VECT-32K	11.9 s	45.2 s	208.6 s	15.5 s
	VECT-128K	175.9 s	718.6 s	TO	264.7 s

program for computing the rolling z-score of elements of a vector by window size specified by the parameter `W` is shown in Figure 8c. This program has three SCCs: the first two are T-stratified and the last one is non-recursive.

K-Means. Clustering a dataset of unlabeled elements is one of the main tasks in unsupervised machine learning with applications in document clustering, image segmentation, and lossy data compression. This algorithm has been already encoded in recursive SQL [57] and can be encoded as a T-stratifiable Datalog program.

Page-Rank. This algorithm was originally used by Google to rank webpages in the search results. In addition, this algorithm can be used for recommendation systems, social network analysis, and fraud detection. This algorithm has also already been encoded as a recursive SQL program [57] and we encode a T-stratifiable version of it in Datalog.

Graph Diameter Computation (GRAPH-DIAM). The longest shortest path between any two edges in a graph is referred to as the graph diameter. This measure can be used for network bottleneck analysis, transportation optimization, and community identification in social networks. This problem can be encoded as a nested T-stratifiable problem; the outer loop is responsible for iterating over source nodes and the inner loop computes the single-source shortest path (SSSP). At the end, the maximum of SSSPs is computed.

6.3 Datasets

Depending on the workload type we use different datasets (cf. Table 1). For the machine learning training workloads, we use a randomly generated dataset named DS-M-N where M and N specify the number of features and elements, respectively. For K-Means, we use randomly generated 2D points named as PNTS-N where N specifies the number of points. For SMP we generate random preferences for N residents/hospitals as the dataset HOSP-N. For the graph problems, we use the RMAT dataset with a slight modification compared with the earlier work [64]. RMAT-N has initially N nodes and $10N$ edges. We modify it by connecting each node to its next node by maximum weight in order to make the graphs fully connected. Finally, for the windowed statistics workloads we use the vector dataset VEC-N where N specifies the size of the vector.

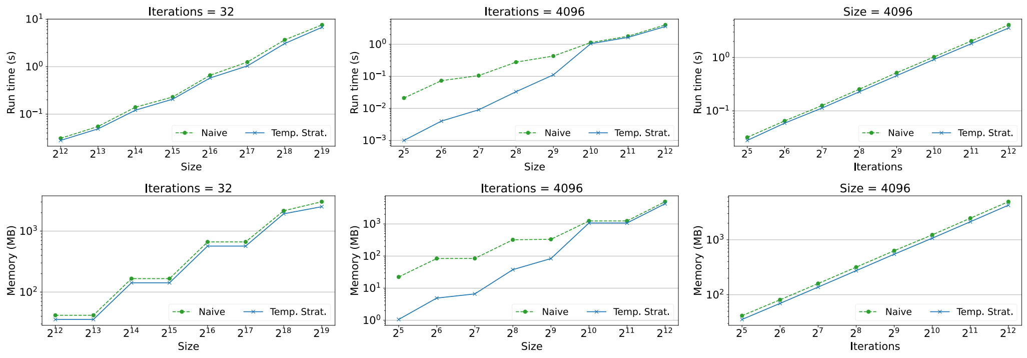


Fig. 9. Run time and memory consumption of Temporel for the iterative Hadamard product kernel using naïve evaluation and temporal stratification by varying the size of data and the number of iterations.

6.4 Benchmarking Recursive Programs

In this section, we show the comparison between the performance of Soufflé, DuckDB, HyPer, and Temporel for a wide range of applications. Table 2 shows the results for different dataset configurations on these systems.

In terms of expressiveness, Soufflé is more limited than other engines. As it only handles stratified aggregations, it does not support most workloads. For MST-PRIM, there is an implementation provided by the developers of Soufflé with higher computational complexity, as can be observed in the run times. The GRAPH-DIAM needs to be implemented by first computing a reachability path query, followed by computing a min-group-by-aggregate query to compute the all-pairs shortest path (APSP), then computing the single-source shortest path (SSSP) by a min-group-by-aggregate query and finally the maximum over the SSSP by a max-aggregate query. Finally, for the WIN-SUM and WIN-ZSCORE, as the queries are monotone, we observe that Soufflé can express them and handle them efficiently using semi-naïve evaluation. The performance is significantly better than DuckDB and HyPer, however, it is still worse than Temporel.

DuckDB has more expressive power than Soufflé and HyPer; the SQL CTE recursion support of DuckDB is sufficient to support all workloads. HyPer cannot support workloads with mutual recursion: BGD, BGD-BTS-LR, and SMP. For PAGE-RANK and K-MEANS we used non-mutually recursive SQL implementations [57]. For the GRAPH-DIAM workload, in all engines except Temporel, we compute the APSP using aggregates inside recursion and then compute the SSSP and graph diameter similar to Soufflé.

Overall, there are two important factors in the performance of DuckDB in comparison with Temporel. First, the higher the number of iterations, the more important becomes the impact of subsumption and temporal elimination in removing redundant computations. Second, workloads with more recursive IDBs will also show better the impact of reducing redundant storage and computations. For BGD-LR, K-MEANS, and PAGE-RANK, DuckDB and HyPer are either competitive or better in comparison with Temporel. This is because these workloads only involve ten iterations, and a few recursive IDBs, which makes the optimized hash-join implementation of in-memory DBMSes more important. The workloads with more iterations such as WIN-SUM and WIN-ZSCORE, and more iterations and more recursive IDBs such as SMP and MST-PRIM show 10×-100× speedup; in some cases, DuckDB timed out. HyPer cannot support SMP, and for MST-PRIM scales worse than Temporel due to not supporting subsumption.

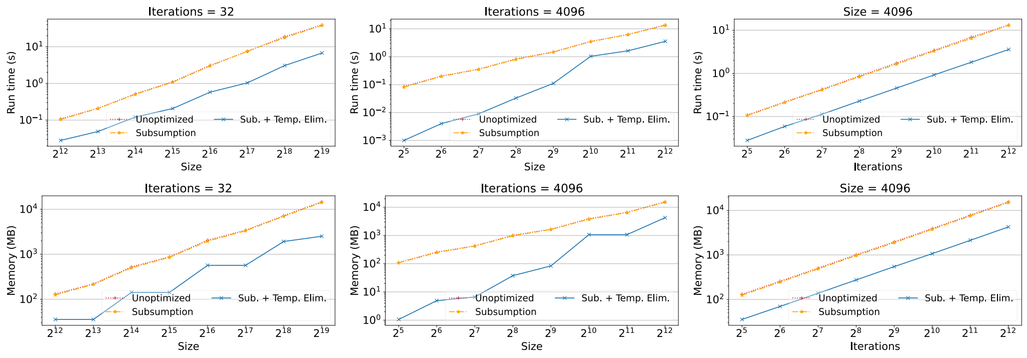


Fig. 10. The impact of different optimizations on the run time and memory consumption of Temporel.

6.5 Effect of Temporal Stratification

In this section, we show the impact of temporal stratification in comparison with a naïve implementation of recursion. For this, we micro-benchmark a synthetic kernel. This kernel performs an iterative Hadamard product of two vectors. We vary the size of the vectors and the number of iterations. The FreeDatalog representation of this kernel is as follows:

```
V2(t, i, v) :- (t = 0), V0(i, v).
V2(t+1, i, v1 * v2) :-
  V1(i, v1), V2(t, i, v2), (t < MAX_ITER).
query(i, v) :- V2(last, i, v).
```

The TempoDL representation for the naïve recursive evaluation of this program is as follows:

```
V2(t, i, v) := (t = 0), V0(i, v).
for iter from 0
  V2(t+1, i, v1 * v2) :=
      V1(i, v1), V2(t, i, v2), (t < MAX_ITER).
query(i, v) := V2(last, i, v).
```

Note that the iteration variable `iter` does not appear in the body of the loop. At each iteration, we need to recreate the results for all intermediate `t`s from scratch (cf. Figure 1b).

Figure 9 shows the run time and memory consumption comparison between naïve recursion and temporal stratified evaluation. As we increase the number of iterations, the gap between the performance widens. Crucially, for small vectors with many iterations (cf. the middle figure), the performance improvement can be one order of magnitude. As the size of the vector gets closer to the number of iterations, the performance improvement decreases to 1.1×.

Similarly, memory consumption is also significantly improved thanks to removing the temporal attribute from the intermediate computations. In all cases, the memory consumption improvement is similar to the run time.

6.6 Effect of Optimizations

Finally, we show the impact of individual optimizations over temporal stratified programs. Similar to the previous section, we only consider the iterative Hadamard product of two vectors using different configurations. We consider three variants of the generated TempoDL program. The unoptimized variant corresponds to TempoDL program produced immediately after compilation from FreeDatalog (similar to Figure 1e). The “Subsumption” variant, which is the result of applying the subsumption optimization to the compiled TempoDL expression (similar to Figure 7a). Finally,

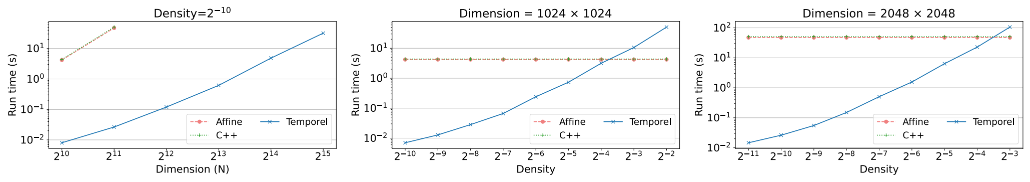


Fig. 11. The comparison of Temporel, the MLIR Affine dialect, and a C++ implementation of a variant of power iteration.

the "Sub. + Temp. Elim." variant corresponds to the fully optimized TempoDL program (similar to Figure 7b).

Figure 10 shows the run time and memory consumption of these three variants. Overall, the improvement trends are similar to the case of naïve recursion and temporal stratified recursion. One can observe that the impact of temporal elimination is more significant than subsumption. This is because even with subsumption, we still keep the results of the intermediate strata; it is only after the temporal stratification combined with subsumption that we completely remove the intermediate history.

6.7 Discussion: Polyhedral Frameworks

The techniques for handling nested recursion with temporal constraints have a close connection to polyhedral frameworks [9] such as Halide [51] and the Affine dialect in the MLIR framework. In this section, we compare Temporel with polyhedral frameworks.

Qualitative Comparison. In general, TempoDL is more generic and more expressive; it allows for data-dependent control flows and arbitrary types of attributes, whereas the polyhedral model only supports *static control programs* [22] over multi-dimensional dense arrays. The static control programs can be summarized as programs with two features [9, 72]. First, its control statements are for-loops with affine bounds and if-statements with affine conditions. Second, these affine bounds and conditions can only depend on constant values and outer loop counters.

This makes the polyhedral model appropriate for applications dealing with multi-dimensional dense array computations such as image processing (the main target domain of Halide). Furthermore, the polyhedral model can capture the temporal loops where the termination can be statically determined. However, for the cases where the termination is dependent on the value of data (e.g., BGD where the termination condition can be specified by a threshold or MST-PRIM where the termination is data dependent) the polyhedral model is not applicable. In addition, the inner loops that implement selections or joins also will involve control flows that are data-dependent and thus go beyond the static control programs. Finally, polyhedral-based optimizers can be used as the backend (for dense cases), however, the high-level transformations to extract temporal information from Datalog (e.g., translation from Figure 1a to Figure 1e) are beyond the scope of polyhedral techniques.

Quantitative comparison. Let us consider an example that both frameworks support. We consider a variant of the power iteration method, a numerical method for computing eigenvalues and eigenvectors of a square matrix with applications in Principal Component Analysis (PCA), Quantum Mechanics, etc. This method is an iterative algorithm that at each iteration involves a matrix-vector multiplication followed by normalization. We remove the normalization inside the loop and perform matrix-matrix multiplication instead of matrix-vector multiplication. The TempoDL program is represented as follows:

```
M2(t, i, j, a) :- (t = 0), M0(i, j, a).
M2(t+1, i, j, sum(m1 * m2)) :-
  M1(i, k, m1), M2(t, k, j, m2), (t < MAX_ITER).
```

```
query(i, j, r) :- M2(last, i, j, r).
```

Figure 11 shows the performance of the code generated by Temporel compared to a version written in the MLIR framework [39] and a C++ implementation. MLIR provides polyhedral optimizations through the Affine dialect [18]. The MLIR Affine implementation is faster than the C++ version thanks to the polyhedral optimizations. However, due to the constraints of the static control flow, it is impossible to express algorithms that leverage the sparsity of matrices. Thus, as the matrices get sparser, which is the case for most relational and graph data, Temporel shows better performance.

7 RELATED WORK

Most efforts on extending Datalog with negation and aggregation in the literature can be classified into two categories [37]. The first category involves restricting the recursion to not allow negation and aggregation inside it. This is achieved by stratified Datalog [45]. The second category supports monotone aggregates in recursion by defining an ordering for the aggregation operators [16, 36, 55, 71]. The key limitation of both approaches is their limited expressive power. For example, neither approach can express the first seven workloads we used in our experiments. Most other approaches in the literature suffer from a similar expressiveness issue [23, 35, 40].

There have been temporal extensions to Datalog either as explicit arguments such as Datalog$_{1S}$ [15, 53, 73], or approaches inspired by temporal logic such as TempLog [2], DatalogLite [26], and DatalogMTL [10, 12, 67]. The former approach admits XY stratification, enabling efficient evaluation of iterative programs without nested recursion [74]. Our approach generalizes this idea by supporting nested recursion. Furthermore, we use program analysis to *automatically* detect temporal argument(s), rather than enforcing the programmer to put them as a distinguished (e.g., first) attribute.

The SQL 1999 standard introduced the `WITH RECURSIVE` keyword to express recursion. There has been recent interest in converting PL/SQL UDFs and Python code into pure SQL recursive queries [19, 30, 31]. Also, the recursive SQL queries have been used to express Machine Learning [11] and Data Mining [57] algorithms. However, the standard SQL recursion is limited to monotonic fixpoint iterations; expressing non-monotonic iterations requires workarounds with additional run time and storage complexities [25]. As a partial remedy, recent proposals on extensions for SQL give more control over recursive evaluation [32, 50]. However, these additional constructs contradict the declarative nature of query languages [32]; rather than specifying *what* the recursive computation needs to do, the programmers are forced to specify *how* it should be performed. Temporel solves this issue by exposing a declarative language, nevertheless, it uses program analysis to determine the execution strategy for T-stratified programs. Timely Dataflows [46] can serve as an alternative backend for Temporel after performing optimizations on TempoDL. Alternatively, one can also target Polyhedral frameworks [9, 22, 51, 72] as the backend for a subset of Datalog programs that can be translated to static control programs, i.e., programs with affine bound controls (cf. Section 6.7). Another promising backend is Sparse Polyhedral Frameworks [66], however, they have limited support for selection predicates and aggregations such as maximum/minimum.

To the best of our knowledge, no other query processing system has an optimization similar to subsumption. Even though we chose Datalog because of its clean, well-understood semantics, our techniques could, in principle, be extended to SQL. The "physical recursive plan" produced by an SQL engine is similar to TempoDL. For example, consider the following recursive SQL query:

```
WITH RECURSIVE presum(j, p) AS (
  SELECT 0, 0
    UNION ALL
  SELECT presum.j + 1, presum.p + vector.p
  FROM presum, vector
  WHERE presum.j < 10 AND presum.j = vector.j )
SELECT presum.p FROM presum
WHERE presum.j = (SELECT MAX(presum.j) FROM presum)
```

In this query, we are only interested in the result computed in the last iteration. By integrating our technique in DBMSes, this query will be rewritten into a recursive physical plan that uses the idea of subsumption to just keep the last state.

8 CONCLUSION

In this paper, we present Temporel, a Datalog engine for efficiently executing recursive programs. We identified a class of Datalog programs called T-stratified programs that can express nested iterative programs as well as semi-naïve evaluation of monotone Datalog programs. We proposed a transformation, called nested temporal stratification, along with an intermediate language, called TempoDL, to capture an efficient evaluation of T-stratified Datalog programs. We have shown empirically that Temporel can express a wide range of iterative workloads, with better performance than the state-of-the-art Datalog and in-memory database systems.

ACKNOWLEDGEMENT

The authors would like to thank Denis Hirn and Torsten Grust for their helpful comments on recursion support in DuckDB. Shaikhha would like to thank Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh. Suciu was partially supported by NSF IIS 2314527, NSF SHF 2312195, and NSF-BSF 2109922. This project was partially supported by RelationalAI.

REFERENCES

[1] [n.d.]. Recursion - RAI Documentation. https://docs.relational.ai/rel/concepts/recursion. Accessed: 2023-03-30.

[2] Martín Abadi and Zohar Manna. 1987. Temporal Logic Programming. In *Proceedings of the 1987 Symposium on Logic Programming, San Francisco, California, USA, August 31 - September 4, 1987.* IEEE-CS, 4–16.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley. http://webdam.inria.fr/Alice/

[4] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1371–1382. https://doi.org/10.1145/2723372.2742796

[5] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[6] David A Bader and Kamesh Madduri. 2006. Gtgraph: A synthetic graph generator suite. (2006).

[7] Isaac Balbin and Kotagiri Ramamohanarao. 1987. A Generalization of the Differential Approach to Recursive Query Evaluation. *J. Log. Program.* 4, 3 (1987), 259–262. https://doi.org/10.1016/0743-1066(87)90004-5

[8] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*, Avi Silberschatz (Ed.). ACM, 1–15. https://doi.org/10.1145/6012.15399

[9] Cédric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France.* IEEE Computer Society, 7–16. https://doi.org/10.1109/PACT.2004.10018

[10] Luigi Bellomarini, Markus Nissl, and Emanuel Sallinger. 2021. Monotonic Aggregation for Temporal Datalog. In *Proceedings of the 15th International Rule Challenge, 7th Industry Track, and 5th Doctoral Consortium @ RuleML+RR 2021 co-located with 17th Reasoning Web Summer School (RW 2021) and 13th DecisionCAMP 2021 as part of Declarative AI*

2021, Leuven, Belgium (virtual due to Covid-19 pandemic), 8 - 15 September, 2021 (CEUR Workshop Proceedings), Ahmet Soylu, Alireza Tamaddoni-Nezhad, Nikolay Nikolov, Ioan Toma, Anna Fensel, and Joost Vennekens (Eds.), Vol. 2956. CEUR-WS.org. http://ceur-ws.org/Vol-2956/paper30.pdf

[11] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Viktor Leis. 2022. Machine Learning, Linear Algebra, and More: Is SQL All You Need?. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p17-blacher.pdf

[12] Sebastian Brandt, Elem Güzel Kalayci, Roman Kontchakov, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. 2017. Ontology-Based Data Access with a Horn Fragment of Metric Temporal Logic. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, Satinder Singh and Shaul Markovitch (Eds.). AAAI Press, 1070–1076. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14881

[13] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 243–262. https://doi.org/10.1145/1640089.1640108

[14] Peter Cholak and Howard A. Blair. 1994. The Complexity of Local Stratification. *Fundam. Informaticae* 21, 4 (1994), 333–344. https://doi.org/10.3233/FI-1994-2144

[15] Jan Chomicki. 1990. Polynomial Time Query Processing in Temporal Deductive Databases. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*, Daniel J. Rosenkrantz and Yehoshua Sagiv (Eds.). ACM Press, 379–391. https://doi.org/10.1145/298514.298589

[16] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, Michael J. Carey and Steven Hand (Eds.). ACM, 1. https://doi.org/10.1145/2391229.2391230

[17] Steven Dawson, C. R. Ramakrishnan, and David Scott Warren. 1996. Practical Program Analysis Using General Purpose Logic Programming Systems - A Case Study. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 117–126. https://doi.org/10.1145/231379.231399

[18] MLIR Developers. 2023. MLIR affine dialect. https://mlir.llvm.org/docs/Dialects/Affine/. Accessed: 2023-10-03.

[19] Christian Duta and Torsten Grust. 2020. Functional-Style SQL UDFs With a Capital 'F'. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1273–1287. https://doi.org/10.1145/3318464.3389707

[20] Jason Eisner and Nathaniel Wesley Filardo. 2010. Dyna: Extending Datalog for Modern AI. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.), Vol. 6702. Springer, 181–220. https://doi.org/10.1007/978-3-642-24206-9_11

[21] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. 2019. Scaling-Up In-Memory Datalog Processing: Observations and Techniques. *Proc. VLDB Endow.* 12, 6 (2019), 695–708. https://doi.org/10.14778/3311880.3311886

[22] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Program.* 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

[23] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. 1991. Minimum and Maximum Predicates in Logic Programming. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 29-31, 1991, Denver, Colorado, USA*, Daniel J. Rosenkrantz (Ed.). ACM Press, 154–163. https://doi.org/10.1145/113413.113427

[24] Gábor E. Gévay, Tilmann Rabl, Sebastian Breß, Lorand Madai-Tahy, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Imperative or Functional Control Flow Handling: Why not the Best of Both Worlds? *SIGMOD Rec.* 51, 1 (2022), 60–67. https://doi.org/10.1145/3542700.3542715

[25] Gábor E. Gévay, Juan Soto, and Volker Markl. 2022. Handling Iterations in Distributed Dataflow Systems. *ACM Comput. Surv.* 54, 9 (2022), 199:1–199:38. https://doi.org/10.1145/3477602

[26] Georg Gottlob, Erich Grädel, and Helmut Veith. 2002. Datalog LITE: a deductive query language with linear time model checking. *ACM Trans. Comput. Log.* 3, 1 (2002), 42–79. https://doi.org/10.1145/504077.504079

[27] Jiaqi Gu, Yugo H. Watanabe, William A. Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. 2019. RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 467–484. https://doi.org/10.1145/3299869.3324959

[28] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. 2006. *codeQuest:* Scalable Source Code Queries with Datalog. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*

(Lecture Notes in Computer Science), Dave Thomas (Ed.), Vol. 4067. Springer, 2–27. https://doi.org/10.1007/11785477_2

[29] Joseph M. Hellerstein. 2010. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.* 39, 1 (2010), 5–19. https://doi.org/10.1145/1860702.1860704

[30] Denis Hirn and Torsten Grust. 2020. PL/SQL Without the PL. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2677–2680. https://doi.org/10.1145/3318464.3384678

[31] Denis Hirn and Torsten Grust. 2021. One WITH RECURSIVE is Worth Many GOTOs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 723–735. https://doi.org/10.1145/3448016.3457272

[32] Denis Hirn and Torsten Grust. 2023. A Fix for the Fixation on Fixpoints. In *Proceedings of the 13th Conference on Innovative Data Systems Research.*

[33] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, 1213–1216. https://doi.org/10.1145/1989323.1989456

[34] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9780. Springer, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

[35] David B. Kemp and Peter J. Stuckey. 1991. Semantics of Logic Programs with Aggregates. In *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*, Vijay A. Saraswat and Kazunori Ueda (Eds.). MIT Press, 387–401.

[36] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. Convergence of Datalog over (Pre-) Semirings. In *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Leonid Libkin and Pablo Barceló (Eds.). ACM, 105–117. https://doi.org/10.1145/3517804.3524140

[37] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. Datalog in Wonderland. *SIGMOD Rec.* 51, 2 (2022), 6–17. https://doi.org/10.1145/3552490.3552492

[38] Gerhard Köstler, Werner Kießling, Helmut Thöne, and Ulrich Güntzer. 1995. Fixpoint iteration with subsumption in deductive databases. *Journal of Intelligent Information Systems* 4, 2 (1995), 123–148.

[39] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[40] Yanhong A. Liu and Scott D. Stoller. 2022. Recursive rules with aggregation: a simple unified semantics. *J. Log. Comput.* 32, 8 (2022), 1659–1693. https://doi.org/10.1093/logcom/exac072

[41] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative networking. *Commun. ACM* 52, 11 (2009), 87–95. https://doi.org/10.1145/1592761.1592785

[42] Magnus Madsen and Ondrej Lhoták. 2018. Safe and sound program analysis with Flix. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 38–48. https://doi.org/10.1145/3213846.3213847

[43] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. Magic is Relevant. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 247–258. https://doi.org/10.1145/93597.98734

[44] Inderpal Singh Mumick and Hamid Pirahesh. 1994. Implementation of Magic-sets in a Relational Database System. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994*, Richard T. Snodgrass and Marianne Winslett (Eds.). ACM Press, 103–114. https://doi.org/10.1145/191839.191860

[45] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. 1990. The Magic of Duplicates and Aggregates. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek (Eds.). Morgan Kaufmann, 264–277. http://www.vldb.org/conf/1990/P264.PDF

[46] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* 439–455.

[47] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[48] Christos Nomikos, Panos Rondogiannis, and Manolis Gergatsoulis. 2005. Temporal stratification tests for linear and branching-time deductive databases. *Theor. Comput. Sci.* 342, 2-3 (2005), 382–415. https://doi.org/10.1016/j.tcs.2005.05.014

[49] Luigi Palopoli. 1992. Testing Logic Programs for Local Stratification. *Theor. Comput. Sci.* 103, 2 (1992), 205–234. https://doi.org/10.1016/0304-3975(92)90013-6

[50] Linnea Passing, Manuel Then, Nina C. Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß (Eds.). OpenProceedings.org, 84–95. https://doi.org/10.5441/002/edbt.2017.09

[51] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2018. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* 61, 1 (2018), 106–115. https://doi.org/10.1145/3150211

[52] Thomas W. Reps. 1993. Demand Interprocedural Program Analysis Using Logic Databases. In *Applications of Logic Databases (The Kluwer International Series in Engineering and Computer Science 296)*, Raghu Ramakrishnan (Ed.). Kluwer, 163–196.

[53] Alessandro Ronca, Mark Kaminski, Bernardo Cuenca Grau, Boris Motik, and Ian Horrocks. 2018. Stream Reasoning in Temporal Datalog. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 1941–1948. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16182

[54] Kenneth A. Ross. 1994. A Syntactic Stratification Condition Using Constraints. In *Logic Programming, Proceedings of the 1994 International Symposium, Ithaca, New York, USA, November 13-17, 1994*, Maurice Bruynooghe (Ed.). MIT Press, 76–90.

[55] Kenneth A. Ross and Yehoshua Sagiv. 1992. Monotonic Aggregation in Deductive Databases. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*, Moshe Y. Vardi and Paris C. Kanellakis (Eds.). ACM Press, 114–126. https://doi.org/10.1145/137097.137852

[56] Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2023. Optimizing Tensor Programs on Flexible Storage. *Proc. ACM Manag. Data* 1, 1 (2023), 37:1–37:27. https://doi.org/10.1145/3588717

[57] Maximilian Emanuel Schüle, Alfons Kemper, and Thomas Neumann. 2022. Recursive SQL for Data Mining. In *SSDBM 2022: 34th International Conference on Scientific and Statistical Database Management, Copenhagen, Denmark, July 6 - 8, 2022*, Elaheh Pourabbas, Yongluan Zhou, Yuchen Li, and Bin Yang (Eds.). ACM, 21:1–21:4. https://doi.org/10.1145/3538712.3538746

[58] Jiwon Seo, Stephen Guo, and Monica S. Lam. 2015. SociaLite: An Efficient Graph Query Language Based on Datalog. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1824–1837. https://doi.org/10.1109/TKDE.2015.2405562

[59] Hesam Shahrokhi and Amir Shaikhha. 2023. Building a Compiled Query Engine in Python. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, Montréal, QC, Canada, February 25-26, 2023*, Clark Verbrugge, Ondrej Lhoták, and Xipeng Shen (Eds.). ACM, 180–190. https://doi.org/10.1145/3578360.3580264

[60] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2018. Push versus pull-based loop fusion in query engines. *J. Funct. Program.* 28 (2018), e10. https://doi.org/10.1017/S0956796818000102

[61] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–33. https://doi.org/10.1145/3527333

[62] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. 2018. Building Efficient Query Engines in a High-Level Language. *ACM Trans. Database Syst.* 43, 1 (2018), 4:1–4:45. https://doi.org/10.1145/3183653

[63] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1907–1922. https://doi.org/10.1145/2882903.2915244

[64] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1135–1149. https://doi.org/10.1145/2882903.2915229

[65] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*,

Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). IEEE Computer Society, 867–878. https://doi.org/10.1109/ICDE.2015.7113340

[66] Michelle Mills Strout, Mary W. Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934. https://doi.org/10.1109/JPROC.2018.2857721

[67] Przemyslaw Andrzej Walega, Bernardo Cuenca Grau, Mark Kaminski, and Egor V. Kostylev. 2019. DatalogMTL: Computational Complexity and Expressive Power. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 1886–1892. https://doi.org/10.24963/ijcai.2019/261

[68] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. 2015. Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines. *Proc. VLDB Endow.* 8, 12 (2015), 1542–1553. https://doi.org/10.14778/2824032.2824052

[69] Jin Wang, Jiacheng Wu, Mingda Li, Jiaqi Gu, Ariyam Das, and Carlo Zaniolo. 2021. Formal semantics and high performance in declarative machine learning using Datalog. *VLDB J.* 30, 5 (2021), 859–881. https://doi.org/10.1007/s00778-021-00665-6

[70] Qiange Wang, Yanfeng Zhang, Hao Wang, Liang Geng, Rubao Lee, Xiaodong Zhang, and Ge Yu. 2020. Automating Incremental and Asynchronous Evaluation for Recursive Aggregate Data Processing. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2439–2454. https://doi.org/10.1145/3318464.3389712

[71] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing Recursive Queries with Program Synthesis. *CoRR* abs/2202.10390 (2022). arXiv:2202.10390 https://arxiv.org/abs/2202.10390

[72] Jingling Xue. 1996. Transformations of Nested Loops with Non-Convex Iteration Spaces. *Parallel Comput.* 22, 3 (1996), 339–368. https://doi.org/10.1016/0167-8191(95)00069-0

[73] Carlo Zaniolo. 2012. Logical Foundations of Continuous Query Languages for Data Streams. In *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings (Lecture Notes in Computer Science)*, Pablo Barceló and Reinhard Pichler (Eds.), Vol. 7494. Springer, 177–189. https://doi.org/10.1007/978-3-642-32925-8_18

[74] Carlo Zaniolo, Natraj Arni, and KayLiang Ong. 1993. Negation and Aggregates in Recursive Rules: the LDL++ Approach. In *Deductive and Object-Oriented Databases, Third International Conference, DOOD'93, Phoenix, Arizona, USA, December 6-8, 1993, Proceedings (Lecture Notes in Computer Science)*, Stefano Ceri, Katsumi Tanaka, and Shalom Tsur (Eds.), Vol. 760. Springer, 204–221. https://doi.org/10.1007/3-540-57530-8_13

[75] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. 2017. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *Theory Pract. Log. Program.* 17, 5-6 (2017), 1048–1065. https://doi.org/10.1017/S1471068417000436

Received July 2023; revised October 2023; accepted November 2023