Incremental Processing of Structured Data in Datalog

André Pacak
JGU Mainz
Germany

Tamás Szabó
GitHub
Germany

Sebastian Erdweg
JGU Mainz
Germany

Abstract

Incremental computations react to input changes by updating their outputs. Compared to a non-incremental rerun, incremental computations can provide order-of-magnitude speedups, since often small input changes trigger small output changes. One popular means for implementing incremental computations is to encode the computation in Datalog, for which efficient incremental solvers exist. However, Datalog is very restrictive in terms of the data types it can process: Atomic data organized in relations. While structured tree and graph-shaped data can be encoded in relations, a naive encoding inhibits incrementality. In this paper, we present an encoding of structured data in Datalog that supports efficient incrementality such that small input changes are expressible. We explain how to efficiently implement and integrate this encoding into an existing incremental Datalog engine, and we show how tree diffing algorithms can be used to change the encoded data.

CCS Concepts: • **Information systems** → **Relational database model**; *Semi-structured data.*

Keywords: incremental computing, Datalog, structured data, relational encoding

ACM Reference Format:
André Pacak, Tamás Szabó, and Sebastian Erdweg. 2022. Incremental Processing of Structured Data in Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3564719.3568686

1 Introduction

Incremental computations allow for dramatic performance speedups. Rather than rerunning a computation when its input changes, an incremental computation processes the change and updates its output by reusing intermediate results and only recomputes intermediate results that are effected by the change [24]. This is beneficial in many scenarios, namely when input changes are small (compared to the complete input) and they trigger output changes that are also small (ideally, proportional to the input change) which can lead to significant performance improvements over a complete re-computation. For example, incremental parsers react to the change of individual tokens to update the generated syntax tree [32], incremental program analyses react to changes of individual nodes of a syntax tree to update the analysis results [28–30], and incremental build systems react to the change of individual files to update the generated build artifacts [8, 15]. All these scenarios exploit the principle of inertia: Computations continue to yield similar outputs when their inputs change over time [10, 20].

There are different ways to realize incremental computations; this paper focuses on incremental computations realized in Datalog. Datalog is a logic programming language that was originally designed to formulate queries in deductive databases [19]. However, Datalog has become quite popular in recent years for solving a wide range of problems [13], from program analysis, to network monitoring and distributed computing. Unfortunately, most of these usages do not exploit one of Datalog's most unique features: incrementality. Datalog's incremental semantics is known since the 1990s [11], yet only few modern Datalog implementations support incrementality today [25, 29, 34]. And even when incrementality is supported, it is not obvious how users can exploit it.

The problem is that Datalog can only process relations of atomic data, while many computations operate on structured data. To process structured data in Datalog, we must encode the data in flat relations, that is sets of tuples. Unfortunately, existing encodings inhibit efficient incrementality because small changes in the structured data require relatively large changes in the relations. For example, consider we want to implement a control-flow analysis in Datalog for the program in Figure 1. The analysis operates on the parsed syntax tree of the program, which we must encode into relations.

Figure 2 illustrates a possible encoding of the unchanged syntax tree as flat relations. For example, the relation `Statement` contains tuples for each statement of all methods in the program. This encoding is similar to the Java bytecode encoding used by Doop, a state-of-the-art analysis platform implemented in Datalog [5]. We also show the derived relation `CfgEdge` that represents the control-flow graph (CFG) of the

GPCE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9920-3/22/12…$15.00
https://doi.org/10.1145/3564719.3568686

```
class App { // change 1: rename App to Application
  // change 2: replace void by int
  public void run(int i) {
    int x = i;
    while (x >= 0) {
      // change 3: insert System.out.println(x);
      if (x % 2 == 0) {
        x = x - 2;
      } else {
        x = x - 1;
}}}}
```

Figure 1. Simple Java program and 3 incremental changes.

program as computed by our Datalog analysis. However, this encoding is ill-suited for incremental computing. Consider three possible changes of the program:

1. Rename of the class `App`. Since the class name occurs as part of unique IDs for methods and statements throughout the encoding, a renaming of `App` requires almost all tuples to be modified. An incremental Datalog engine must delete all information derived about `App` including those in `CfgEdge` and then re-derive all that information again.
2. Change the return type of method `run`. Since relation `Method` contains all information about methods, we must delete the old tuple of `run` and insert a new one with the updated return type. Even though the return type does not influence the CFG, our Datalog analysis must process the replaced tuple carefully to discover that the CFG is unaffected. Generally, large compound tuples hamper incrementality because the entire tuple has to be replaced even if just one element changes.
3. Insert a `println` statement at the beginning of the while-loop (index `3`). Since the encoding uses absolute indices for statements, the indices of all subsequent statements have to be incremented. For an incremental Datalog engine, this is equivalent to deleting those statements and reinserting them at a different position. In our example, we would have to recompute all but the very first `CfgEdge` tuples.

In this paper, we present an encoding of structured data in Datalog that enables efficient incremental updates. Our encoding supports incremental changes like those described above by (i) avoiding context-sensitive unique IDs, (ii) avoiding large compound tuples, and (iii) avoiding absolute positioning. Specifically, we show how our encoding supports arbitrary tree and list data. We have implemented our encoding as part of the incremental Datalog engine IncA [28–30]: Users provide structured input data, which we translate into flat relations for IncA to process. As part of this, we developed efficient index structures for relations that encode structured data and demonstrate how to use tree diffing algorithms to trigger incremental updates of the encoded data.

```
ClassType(name) = [
  "App"
]
Method(uid, name, params, class, type) = [
  ("App.run(int)", "run", "(int)", "App", "void")
]
Statement(method, index, uid) = [
  ("App.run(int)", 1, "App.run(int)/assign/1"),
  ("App.run(int)", 2, "App.run(int)/while/1"),
  ("App.run(int)", 3, "App.run(int)/if/1"),
  ("App.run(int)", 4, "App.run(int)/assign/2"),
  ("App.run(int)", 5, "App.run(int)/else/1"),
  ("App.run(int)", 6, "App.run(int)/assign/3")
]
// derived relation
CfgEdge(method, fromStmIndex, toStmIndex) = [
  ("App.run(int)", 1, 2),
  ("App.run(int)", 2, 3),
  ("App.run(int)", 3, 4),
  ("App.run(int)", 3, 6),
  ("App.run(int)", 4, 2),
  ("App.run(int)", 6, 2)
]
```

Figure 2. Tabular encoding of the program from Figure 1.

In summary, we present the following contributions:

- An encoding of structured data in Datalog that supports efficient incremental updates (Section 3).
- An implementation of efficient indices to answer Datalog queries against encoded structured data (Section 4).
- A processor that translates tree-diffing patches into updates of the encoded data (Section 5).

Our encoding has been tested and refined for the last 6 years and various case studies showcase the efficient incrementalization our encoding enables. We report on these case studies in Section 6, but they have been previously published.

2 Background on IncA

IncA is an incremental Datalog platform implemented in Scala2 that we extend to support the efficient incremental processing of structured data. This section describes the architecture of IncA and explains how users interact with an incremental Datalog solver. This background knowledge is required for understanding where and how to support structured data in IncA.

Architecture. We show the architecture of IncA in Figure 3, which consists of a compiler frontend, a compiler backend, and a runtime system. The frontend provides different user-facing languages to describe programs with Datalog-style least-fixpoint semantics. In particular, the *constraint language* resembles traditional Datalog [30], whereas the *functional language* promotes the definition of recursive functions [21]. All frontend languages must compile to IncA's

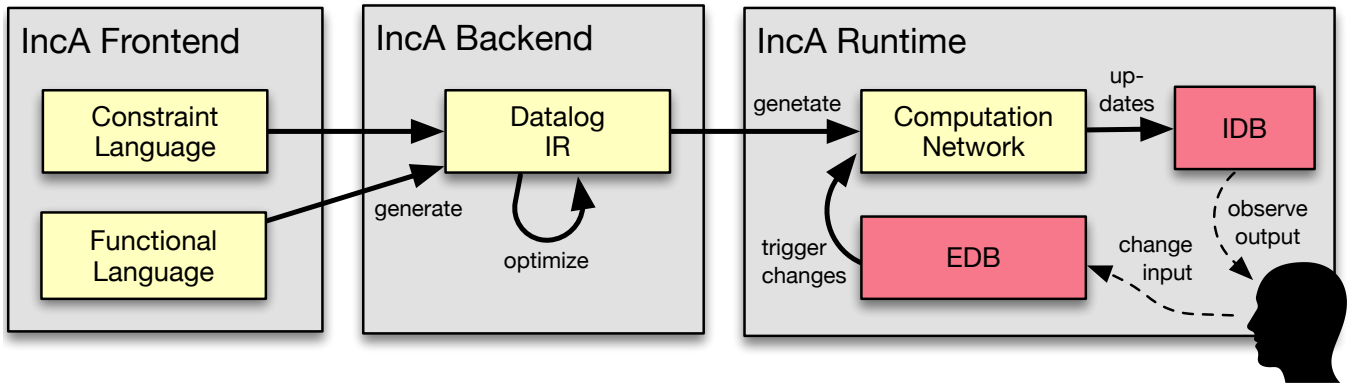

Figure 3. The architecture of IncA. Yellow boxes represent computations, red boxes represent input and output data.

intermediate representation (IR), which consists of *plain Datalog* with a few extensions: stratified negation, user-defined data types, and user-defined recursive aggregation [28]. The architecture is open for extension: For example, there also is a frontend that compiles Soufflé-style Datalog [26] to IncA's IR.

The compiler backend optimizes the Datalog IR generated by the frontends. Optimizations include constant folding, constant propagation, and inlining. After optimization, the backend generates code for a target platform to execute. The primary target platform of IncA is Viatra, which implements incremental processing through a computation network [31]. We will focus on how to provide structured data for Viatra in the remainder of this paper. However, the architecture is open and our encoding is applicable to other target platforms, for example, IncA can translate the IR to code that can be run by Soufflé.

Incremental user interaction. A Datalog program consists of logical rules that express implications. For example, consider the standard path example:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

This program computes the transitive closure `path` of a relation `edge`: (i) If there is an edge from `X` to `Y`, then there is a path from `X` to `Y`. (ii) If there is an edge from `X` to `Z` and if we already have found a path from `Z` to `Y`, then there also is path from `X` to `Y`. That is, this program computes tuples of a relation `path` given tuples of a relation `edge` as input.

In Datalog lingo, input relations define an extensional database (EDB), whereas the computed relations define an intensional database (IDB). When using a non-incremental Datalog solver, users must provide the EDB before execution and get back the IDB after execution. However, with an incremental Datalog solver, the user interactions become more interesting as shown on the right end of Figure 3.

With an incremental Datalog solver, users load a Datalog program with an initial EDB that is usually empty. For IncA, the Datalog solver instantiates a computation network that awaits user input. Users can then continuously provide EDB changes, which the computation network processes to update the IDB. The user can observe those changes in form of deletions and insertions of tuples in the IDB. For example, consider the following incremental user interactions with the `path` program (`+` is tuple insertion, `-` is tuple deletion):

```
+edge(1,2) =>  +path(1,2)
+edge(2,3) =>  +path(2,3), +path(1,3)
+edge(3,1) =>  +path(3,1), +path(2,1), +path(1,1),
               +path(2,2), +path(3,3)
-edge(3,1)
+edge(2,1) =>  -path(3,1), -path(3,3)
```

Datalog interactions are inherently relational: tuple insertions and deletions. How can users provide and change non-relational data such as structured data?

Problem statement. We want to exploit Datalog's incrementality for all kinds of computations, including those that process structured data (e.g., program analyses). To this end, we must solve two challenges. First, we must find a relational encoding of structured data so that we can represent structured data in the EDB. Second, when the structured data changes, we must translate those changes into tuple insertions and deletions in the EDB. All the while, our encoding must not inhibit the incremental performance. That is, the encoding and the change translation may not increase the computational complexity of the incremental Datalog solver.

3 Encoding of Structured Data

When encoding structured data for an incremental Datalog solver, we must submit to one rule: Small changes in the structural data must translate to small changes of the EDB. In the remainder of this section, we show how to support three kinds of structured data: tree-shaped data, optional data, and lists. We follow three design principles in our encoding:

(i) Use context-insensitive unique identifiers.
(ii) Use unary and binary relations only.
(iii) Use relative positioning for list elements.

3.1 Encoding Trees

We use the following data type (in Scala3 syntax) as a running example to explain the relational encoding of trees:

```
enum Exp:
  case Num(value: Int)
  case Add(left: Exp, right: Exp)
```

For example, the tree `Add(Num(5), Num(3))` has an addition as root node with `Num(5)` as `left` subtree. We assign each node a unique ID, so that we can refer to and update it directly without any navigation through the tree. We annotate IDs as subscripts in our examples: `Add#1(Num#2(5), Num#3(3))`.

Our unique IDs are atomic and do not contain any context-sensitive information. This is important because it allows a tree to change independently from its context and vice versa. For the Java example from Section 1, we can rename a class without requiring changes to any of the contained trees that describe fields and methods. And conversely, we can move a tree to another context (e.g., another class) without updating its unique IDs. We show an example of such update below.

Unique IDs have an additional benefit: We can use them as keys in our relational encoding. Specifically, we encode trees using two families of relations: one that enumerates all nodes of a given type, and one that enumerates all links given a type and a field name:

$$\mathsf{Node}_{type} \subseteq \mathit{UID}$$
$$\mathsf{Link}_{type.fieldname} \subseteq \mathit{UID} \times (\mathit{UID} \cup \mathit{Value})$$

For each node type T, we collect the unique IDs of nodes in Node_T. For our example data type, these are three relations Node_{Num}, Node_{Add}, and Node_{Exp}, where the latter subsumes the former two due to subtyping. Datalog programs can query these node relations to emulate pattern matching.

To encode the shape of a tree, we use a binary link relation for each field. For each node type T and *fld*, we collect the links between a node and its field value in $\mathsf{Link}_{T.fld}$. The value of a field is either a primitive *Value* such as `Int`, or another node identified by its unique ID. Datalog programs can query these link relations to traverse structured data.

Using our node and link relations, we can encode the tree `Add#1(Num#2(5), Num#3(3))` using the following Datalog tuples:

$\mathsf{Node}_{Add}(\#1)$ $\mathsf{Node}_{Num}(\#2)$ $\mathsf{Node}_{Num}(\#3)$
$\mathsf{Node}_{Exp}(\#1)$ $\mathsf{Node}_{Exp}(\#2)$ $\mathsf{Node}_{Exp}(\#3)$
$\mathsf{Link}_{Add.left}(\#1, \#2)$ $\mathsf{Link}_{Add.right}(\#1, \#3)$
$\mathsf{Link}_{Num.value}(\#2, 5)$ $\mathsf{Link}_{Num.value}(\#3, 3)$

We can query these relations in Datalog code to collect all numeric literals found in an expression tree:

```
nums(e,n) :- Node_Num(e), Link_Num.value(e,n).
nums(e,n) :- Node_Add(e), Link_Add.left(e,l), nums(l,n).
nums(e,n) :- Node_Add(e), Link_Add.right(e,r), nums(r,n).
```

This program computes a relation $\mathtt{nums} \subseteq \mathit{UID} \times \mathit{Value}$. If `e` is a *Num*, then we extract its numeric literal stored in field *value*. Otherwise, `e` is an *Add* and we extract its left (resp. right) operand and recursively collect its numeric literals. For our example, this program will derive the following tuples:

$\mathtt{nums}(\#2, 5)$ $\mathtt{nums}(\#3, 3)$ $\mathtt{nums}(\#1, 5)$ $\mathtt{nums}(\#1, 3)$

Consider we negate the left-hand operand of our example tree, resulting in `Add#1(Neg#4(Num#2(5)), Num#3(3))`. We can represent this change compactly as tuple updates:

$+\mathsf{Node}_{Neg}(\#4)$ $+\mathsf{Node}_{Exp}(\#4)$
$-\mathsf{Link}_{Add.left}(\#1, \#2)$ $+\mathsf{Link}_{Add.left}(\#1, \#4)$ $+\mathsf{Link}_{Neg.e}(\#4, \#2)$

The first, second, and last tuple change are responsible for loading the new `Neg` node. The amount of tuple changes needed for loading a new tree is proportional to its size. The other two changes manipulate *Add.left* of `Add#1` to replace the old operand by the new negation node. For updating existing nodes, the amount of tuple changes needed is proportional to the number of node changes in the structured data. Overall, this means that our encoding translates small changes in the structural data to small changes of the EDB.

3.2 Encoding Optional Data

Structured data regularly makes use of optional elements. For example, consider a lambda expression with an optional type annotation:

```
  case Lam(param: String, ty: Option[Type], body: Exp)
```

We model optional data as fields whose value may be undefined. That is, if `#1` is a *Lam* node, then $(\#1, v) \in \mathsf{Link}_{Lam.ty}$ may be undefined for all v, meaning there is no type annotation present.

There are two concerns we must discuss: (i) what is the semantics of a field being undefined, and (ii) how can a Datalog program test the definedness of a field. To illustrate these concerns, consider the following Datalog rule, which type checks a lambda expression:

```
tcheck(ctx, e) :- Node_Lam(e), Link_Lam.ty(e,t),
                  Link_Lam.param(e,x), Link_Lam.body(e,b),
                  bind(ctx, x, t, ctx'), tcheck(ctx', b).
```

The second call in the body of this rule queries the optional `ty` field, which may be undefined. Fortunately, the standard Datalog semantics dictates the correct semantics: When a subquery fails, then the entire rule is aborted and does not yield any outputs. That is, if `ty` is undefined, then the query $\mathsf{Link}_{Lam.ty}(\mathtt{e}, \mathtt{t})$ fails, and thus the entire rule fails as desired. Hence, the query of $\mathsf{Link}_{Lam.ty}$ tests for the definedness of that field. To allow users to test if a field is undefined, we add synthesized undefLink relations into the EDB:

```
tcheck(ctx, e) :- Node_Lam(e), undefLink_Lam.ty(e),
                  infer(ctx, e).
```

We synthesize undefNode and undefLink for all node types and all fields. As we will explain in Section 4, these relations can be efficiently implemented as database views, that is, without any memory overhead.

3.3 Encoding Lists

At last, we encode lists as relations. Consider an additional `Exp` case for calls with multiple arguments:

```
case Call(f: String, args: List[Exp])
```

One option for handling lists is to represent them as standard structured data using `Cons` and `Nil` nodes. However, we would have to load and incrementally maintain a `Cons` node for each list element. Moreover, the `Cons` nodes would be visible to Datalog developers, for example, when querying the parent of a node. Alternatively, we could model a list node and assign indices to the list elements, similar to an array. However, as we have discussed in Section 1, this encoding inhibits incrementality, because deleting or inserting a list element affects the indices of all subsequent elements.

We propose an encoding that behaves like a linked list, but does not require explicit `Cons` nodes. To achieve that, we introduce two special relations:

$$\#first \subseteq UID \times UID \qquad \#next \subseteq UID \times UID$$

Relation $\#first$ associates a list node to its first list element; $\#first$ is undefined if a list is empty. And for each list element, relation $\#next$ associates it to the subsequent list element. For example, we encode the program

```
Call#1("f", List#2(Num#3(5), Num#4(3), Num#5(7)))
```

as the following Datalog tuples:

$\mathsf{Node}_{Call}(\#1)$ $\mathsf{Node}_{Exp}(\#1)$ $\mathsf{Link}_{Call.f}(\#1, \text{"f"})$

```
// Node and Link tuples for Num nodes #3,#4,#5 elided
```

$\mathsf{Node}_{List[Exp]}(\#2)$ $\mathsf{Link}_{Call.args}(\#1, \#2)$ $\#first(\#2, \#3)$
$\#next(\#3, \#4)$ $\#next(\#4, \#5)$

The node with UID #2 is a list node of type $List[Exp]$. Its first element is #3 as defined by the $\#first$ fact, with subsequent elements #4 and #5. This encoding is incrementally efficient in the sense that we can replace, insert, or remove any list element in constant tuple changes, akin to the corresponding operations on linked lists. Finally, note that Datalog programs can query $\#next$ in the opposite direction to find a list element's predecessor. Thus, in fact, our encoding supports bidirectional traversals similar to doubly linked lists.

4 Querying Encoded Structured Data

In the previous section, we showed how to encode structured data in a relation format: To this end, we introduced the following relations:

- Node relations: $\mathsf{Node}_{type} \subseteq UID$
- Link relations: $\mathsf{Link}_{type.fieldname} \subseteq UID \times (UID \cup Value)$
- First link relation: $\#first \subseteq UID \times UID$
- Next link relation: $\#next \subseteq UID \times UID$
- Undefined nodes: $\mathsf{undefNode}_{type} \subseteq UID$
- Undefined links: $\mathsf{undefLink}_{type.fieldname} \subseteq UID$

The relations above serve as input (EDB) for the incremental Datalog solver and its computation network. Our encoding determines how structured data appears in the EDB. The computation network interacts with the EDB in three ways: (i) it enumerates tuples of an EDB relation, (ii) it tests the containment of a tuple in an EDB relation, (iii) it reacts to the insertion or deletion of a tuple from an EDB relation. In this section, we explain how these interactions can be supported efficiently by exploiting invariants of structured data. Specifically, we introduce data structures called *indices* for the EDB that can answer the queries of the computation network efficiently.

```
class UnarySetIndex[V]:
  private val set: mutable.Set[V]

  def entries: Iterable[V] = set
  def contains(v: V): Boolean = set.contains(v)

  def insert(v: V): Unit =
    set += v
    notify(v, true)
  def delete(v: V): Unit =
    set -= v
    notify(v, false)

  def notify(v: V, isInsert: Boolean): Unit = ...
  def addListener(l: Listener): Unit = ...
  def removeListener(l: Listener): Unit = ...
```

Figure 4. Unary set index implementation.

4.1 Node Relations

Nodes are fundamental entities of tree-shaped data. We encode nodes using node relations in the EDB as explained in Section 3. Hence, we need to enable efficient querying of node relations for the computation network. In our encoding, nodes exhibit an important invariant: each node carries a unique ID. For the node relations, this means that we do not need to consider duplicate node IDs and can use a simple set to store the UIDs of nodes per type.

Figure 4 shows the code of our unary set index. There is one instance of this class per type. In each type, we store the contents of the node relation in a mutable set. The index can answer queries such as enumerating all stored nodes (`entries`) and if a node is already indexed (`contains`).

The EDB will change over time by processing updates of the input data (details in Section 5). To this end, the index provides functions `insert` and `delete` to alter the underlying mutable set. The computation network installs listeners on EDB relations to be notified about changes. Our unary set index notifies these listeners when a node is inserted or deleted by calling `notify`. The first argument of `notify` states the value that the change is about and the second argument informs the listeners if the change was an insertion (`isInsert = true`) or a deletion (`isInsert = false`). Since nodes

```
trait BinaryIndex[K, V]:
  def entries: Iterable[(K, V)]
  def index(k: K): Iterable[V]
  def indexInverted(v: V): Iterable[K]
  def contains(k: K, v: V): Boolean =
    index(k).exists(_ == v)

  def insert(k: K, v: V): Unit
  def delete(k: K, v: V): Unit

  def notify(k: K, v: V, isInsert: Boolean): Unit = ...
  def addListener(l: Listener): Unit = ...
  def removeListener(l: Listener): Unit = ...
```

Figure 5. Generic binary index interface.

are uniquely identified, double insertions or double deletions cannot occur and do not have to be prevented.

4.2 Link Relations

We encode the shape of structured data using link relations in the EDB as explained in Section 3. Given a type and field name, the link relations associate nodes with other nodes or with primitive values. To efficiently implement indices for link relations, we need to identify invariants that we can exploit to detect and propagate changes.

As first invariant, we observe that the target of a link is always uniquely determined by the containing node. Formally, given a node n, a type T, and a field fld, then

$$|\{t \mid (n,t) \in \mathsf{Link}_{T.fld}\}| \leq 1.$$

That is, a field's value is either undefined or definite; there cannot be multiple values assigned to a single field of a node. However, the computation network may also query the link relation in reverse order: given a link target, what are the nodes that point there? In this direction, the link relations do not provide a strong invariant. For example, the following tree `Add#1(Num#2(3), Num#3(3))` contains the integer value 3 twice, as target of #2 and #3. But this is only a problem for primitive values, which are not unique.

To implement link relations efficiently, we split them apart. We distinguish *node link relations* (targeting nodes) from *value link relations* (targeting primitive values). Node link relations range over $UID \times UID$ whereas value link relations range over $UID \times Value$. This separation is always possible for well-sorted trees, where a field targets a statically known type: either a node or a value. Now we can observe a second invariant for node link relations: the target of a link uniquely determines the containing node. This invariant does not hold for value link relations.

Due to these invariants, we can implement:

- node link relations as one-to-one indices and
- value link relations as many-to-one indices.

These indices share a common interface presented in Figure 5. The interface provides the same functionality as `UnarySetIndex` such as enumerating all elements as well as registering and notifying listeners but for binary tuples. Additionally, it provides two functions that answer directional queries. The function `index` gives all values that have `k` as key whereas `indexInverted` returns all keys where `v` is the value. The concrete implementations of these two functions will directly make use of the two invariants of binary relations.

```
class OneToOneIndex[K, V] extends BinaryIndex[K, V]:
  private val biMap: mutable.BiMap[K, V]

  def entries: Iterable[(K, V)] = biMap.entries
  def index(k: K): Iterable[V] = biMap.get(k)
  def indexInverted(v: V): Iterable[K] =
    biMap.getInverse(v)

  def insert(k: K, v: V): Unit =
    biMap.put(k, v)
    notify(k, v, true)
  def delete(k: K, v: V): Unit =
    biMap.remove(k)
    notify(k, v, false)

class ManyToOneIndex[K, V] extends BinaryIndex[K, V]:
  private val forward: mutable.Map[K, V]
  private val backward: mutable.MultiMap[V, K]

  def entries: Iterable[(K, V)] = forward.entries
  def index(k: K): Iterable[V] = forward.get(k)
  def indexInverted(v: V): Iterable[K] = backward.get(v)

  def insert(k: K, v: V): Unit =
    forward.put(k, v)
    backward.put(v, k)
    notify(k, v, true)
  def delete(k: K, v: V): Unit =
    forward.remove(k)
    backward.remove(v, k)
    notify(k, v, false)
```

Figure 6. Binary one-to-one and many-to-one indices.

We show both implementations in Figure 6, utilizing the respective invariants. We implement `OneToOneIndex` that exploits the one-to-one mapping between keys and values. To exploit the invariant, we use a bidirectional map that maintains an inverse view of the data to store a one-to-one mapping efficiently. We implement `ManyToOneIndex` that exploits the many-to-one mapping between keys and values. That is, a key uniquely identifies its value, but a value may be contained by multiple keys. To exploit this invariant, we use a forward-directed map to answer `index` queries and a backwards-directed multimap for `indexInverted` queries.

4.3 Optional and List Relations

We encode optional data as links with undefined targets. We already took this into consideration when we defined the

```
class UndefNode(tyNodes: UnarySetIndex,
                allNodes: UnarySetIndex):
  def entries: Iterable[UID] =
    allNodes.entries.diff(tyNodes.entries)
  def contains(k: UID): Boolean =
    !tyNodes.contains(k)

  allNodes.addListener((v: URI, isInsertion: Boolean) =>
    if (!tyNodes.contains(v))
      notify(v, isInsertion)
  )
```

Figure 7. Index undefNode reacts to `allNodes` changes.

one-to-one and many-to-one indices above, where `index` and `indexInverted` may yield empty results. Thus, no additional handling is needed to support optional data.

To encode lists, we use #*first* and #*next* relations. They also exhibit strong invariants: Each list has at most one first element and each node can only be the first element of a single list only. Hence, we can use our one-to-one index to implement the #*first* relation. Moreover, each node can only be the successor of a single list predecessor, so that we can use the one-to-one index for the #*next* relation, too.

4.4 Virtual Indices

The goal of our indices is to enable efficient querying of structured data encoded in the EDB. However, sometimes it is not necessary to materialize the structured data explicitly. We found that we can answer queries for the relations undefNode and undefLink by relying on other EDB relations without storing extra data. In databases, this would be called a *view*. We implement views with what we call *virtual indices*.

A virtual index supports the same interactions as other indices: enumeration, containment, and incremental updates. However, it does so by delegation to other EDB indices. For example, consider the undefNode relation. For each type T, undefNode_T contains all nodes not of type T. Rather than storing all these nodes explicitly, we can answer queries for this relation indirectly as shown in Figure 7.

The virtual index `UndefNode` takes two other indices as input: one containing all nodes of type T and one containing all nodes. The nodes *not* of type T can then be computed as the difference of all nodes and the nodes of type T. For containment, we only need to test `tyNodes`, because the membership in `allNodes` is implicit. However, the most interesting and complicated part of virtual indices concerns the handling of incremental updates. To this end, we install two change listeners in other relations. When `allNodes` changes, we propagate the change if the changed node is not of type T. A new node will become part of the `UndefNode` relation if and only if it is not of type T. Note that we do not observe changes in `tyNodes`, because the type of a node can not change.

```
class UndefLink(tyNodes: UnarySetIndex,
                links: OneToOneIndex):
  def entries: Iterable[UID] =
    tyNodes.entries.filter(n => links.index(n).isEmpty)
  def contains(k: UID): Boolean =
    tyNodes.contains(k) && links.index(k).isEmpty

  tyNodes.addListener((v: URI, isInsertion: Boolean) =>
    // fields are undefined when a node is (un)loaded
    notify(v, isInsertion)
  )
  links.addListener((v: URI, t: URI, isIns: Boolean) =>
    notify(v, !isIns)
  )
```

Figure 8. undefLink reacts to `tyNodes` and `links` changes.

```
type Field = (Type,Link)
class EDB:
  val nodes: Map[Type, UnarySetIndex[UID]]
  val nodeLinks: Map[Field, OneToOneIndex[UID, UID]]
  val valLinks: Map[Field, ManyToOneIndex[UID,Value]]
  val first: OneToOneIndex[UID, UID]
  val next: OneToOneIndex[UID, UID]

  // UndefNode and UndefLink indices
  val virtualIndices: Map[VKey, VirtualIndex]
```

Figure 9. Relational encoding, backed by efficient indices.

Similarly, we can define a virtual index `UndefLink` that contains all nodes of type T with an undefined link *fld*. This virtual index listens to two other indices as shown in Figure 8. When a node is inserted into `tyNodes`, its links are undefined by default. Therefore, we notify the listeners of `UndefLink` whenever `tyNodes` changes. If later the value of $T.fld$ is changed by the user, we must update `UndefLink` accordingly. When a new tuple (v, t) is inserted into $\text{Link}_{T.fld}$ (the field is set to a new value), then we must remove v from $\text{undefLink}_{T.fld}$. Conversely, when a tuple (v, t) is deleted from $\text{Link}_{T.fld}$ (the field is undefined), then we must insert v into $\text{undefLink}_{T.fld}$. The `links` listener captures this behavior.

Virtual indices allows us to implement additional EDB relations without memory overhead. Note also that there are no insertion and deletion functions in virtual indices, because they do not store data themselves.

4.5 Summary

We have now presented all indices required to implement our encoding of structural data as an EDB. We summarize the EDB definition in Figure 9. Note how node relations are indexed by the node type and link relations are indexed by the node type and link. In our implementation, indices are instantiated on-demand when they are first needed. For node relations, we use the unary set index. For binary relations with a one-to-one invariant, we use the `OneToOneIndex`. For

```
case class EditScript(edits: Seq[Edit])

enum Edit:
  case Load(n: Node, ks: Kids, vs: Vals)
  case Attach(n: Node, link: Link, p: Node)
  case Unload(n: Node, ks: Kids, vs: Vals)
  case Detach(n: Node, link: Link, p: Node)
  case Update(n: Node, ovs: Vals, nvs: Vals)
type Node = (Type, UID)
type Kids = Seq[(Link, UID)]
type Vals = Seq[(Link, Value)]
```

Figure 10. The edit script language *truechange* [9].

binary relations with a many-to-one invariant, we use the `ManyToOneIndex`. This indices allow the computation network to process changes of structural data efficiently.

5 Processing Changes of Structured Data

The EDB implementation describes the structured data in a relational encoding. To react to input changes, we need to translate changes of structured data to insertions and deletions of EDB tuples. To determine changes in structured data, we use the structural diffing algorithm *truediff* [9]. *truediff* generates edit scripts that precisely describe tree-diff patches, but it also describes how to construct an initial tree using edits. The runtime of IncA will translate edit scripts to insertions and deletions of the EDB as we show below. But first, we introduce the edit script language used by *truediff*.

5.1 *truechange*: Structural Edit Scripts

Tree diffing algorithms like *truediff* generate edit scripts that describe how a tree was modified. For *truediff*, the corresponding edit script language is called *truechange*. *truechange* uses UIDs to uniquely identify nodes of abstract syntax trees. This way, *truediff* can directly reference changed nodes without requiring navigation through the tree. This design fits well with the extensional database encoding of structured data we introduced in Section 3, which also relies on UIDs.

We reiterate the edit script language *truechange* [9] in Figure 10. An edit script consists of a sequence of edits. Each edit is one of five atomic operations: load, attach, unload, detach or update. Edits reference nodes by the node's type together the node's UID, which we visualize as a subscript. For example, node `Add#1` is a node of type `Add` with UID `#1`.

A `Load(n,ks,vs)` constructs a new node with a new UID and connects that node with all its child nodes `ks` and contained values `vs`. Note how child nodes and contained values also provide a link, with which they connect to the newly constructed node. An `Unload(n,ks,vs)` is the dual edit of `Load`: it deconstructs a node and disconnects all child nodes, which remain loaded. Disconnected nodes can be reattached through another `Load` or through `Attach`. An `Attach(n,link,p)` connects node `n` with the parent node `p` via link `link` if that link is currently unoccupied. Conversely, `Detach(n,link,p)` disconnects `n` from `p`. At last, an `Update(n,ovs,nvs)` edit which updates the old values `ovs` of the node with new values `nvs`.

For example, consider the change we have already seen in Section 3:

$$\begin{aligned}&\mathtt{Add}_{\#1}(\mathtt{Num}_{\#2}(5), && \mathtt{Num}_{\#3}(3))\\ &\mathtt{Add}_{\#1}(\mathtt{Neg}_{\#4}(\mathtt{Num}_{\#2}(5)), && \mathtt{Num}_{\#3}(3))\end{aligned}$$

truediff computes the following *truechange* edit script:

$$\begin{aligned}&\mathtt{Detach}(\mathtt{Num}_{\#2}, \text{"left"}, \mathtt{Add}_{\#1})\\ &\mathtt{Load}(\mathtt{Neg}_{\#4}, \mathtt{Seq}(\text{"e"} \to \#2), \mathtt{Seq}())\\ &\mathtt{Attach}(\mathtt{Neg}_{\#4}, \text{"left"}, \mathtt{Add}_{\#1})\end{aligned}$$

First, we detach node `Num#2` from link `left` of node `Add#1`. Then we load a new node of type `Neg#4` with node `Num#2` attached at link `e`. Lastly, we attach the newly loaded node `Neg#4` at the previously freed link `left` of node `Add#1`. We process edit scripts like this in IncA to update the structured data encoded in the EDB.

5.2 Translating Edit Scripts

The runtime of IncA takes an edit script as input and translates it to changes of the extensional database. We assume that the type of the structured data is consistent with the node and link relations in the EDB: For each node type, there are corresponding node and link relations in the EDB. The translation then follows the code in Figure 11.

The EDB provides the function `processEditscript`, which processes each edit individually by calling `processEdit`. The EDB only notifies the computation network after the whole edit script has been processed and the EDB has been altered. The function `processEdit` handles each of the five edits differently. When processing a load edit, we insert the UID of the node into all supertype node indices. We also insert into the supertype indices to enable the computation network to query nodes of all types without requiring extra computation. Additionally, we need to insert all kids in the corresponding link index. The same holds for values of the node. Note that loads of list nodes do not have direct kids and values.

When attaching a node we distinguish between the provided links. When considering link `First`, we insert it into the `first` index. The same holds for the when encountering `Next`. We need to make this distinction because first and next links are not indexed by the nodes type. If the link is neither a first or next link, we insert the parent-child pair into the link relation indexed by the parent's type and the link itself.

Since `Load` and `Unload` are dual edits, we delete instead of insert into the same indices when encountering an unload edit. The same reasoning holds for `Attach` and `Detach`.

At last, we process the Update edit by deleting all old values `ovs` for the respective value link relations and inserting all new values `nvs` afterwards. We assume that `ovs` and `nvs` mention the same links.

```
def processEditScript(es: EditScript): Unit =
  es.edits.foreach(processEdit)

def processEdit(edit: Edit): Unit = edit match
  case Load((ty, uid), kids, vals) =>
    supertypes(ty).foreach { supty =>
      nodes(supty).insert(uid)
    }
    kids.foreach { case (link, kuid) =>
      links(ty -> link).insert(uid, kuid)
    }
    vals.foreach { case (link, v) =>
      valLinks(ty -> link).insert(uid, v)
    }
  case Attach((nty, nuid), First, (pty, puid)) =>
    first.insert(puid, nuid)
  case Attach((nty, nuid), Next, (pty, puid)) =>
    first.next(puid, nuid)
  case Attach((nty, nuid), link, (pty, puid)) =>
    links(pty -> link).insert(puid, nuid)
  case Unload((ty, uid), kids, lits) =>
    supertypes(ty).foreach { supty =>
      nodes(supty).delete(uid)
    }
    kids.foreach { case (link, kuid) =>
      links(ty -> link).delete(uid, kuid)
    }
    vals.foreach { case (link, v) =>
      valLinks(ty -> link).delete(uid, v)
    }
  case Detach((nty, nuid), First, (pty, puid)) =>
    first.delete(puid, nuid)
  case Detach((nty, nuid), Next, (pty, puid)) =>
    first.delete(puid, nuid)
  case Detach((nty, nuid), link, (pty, puid)) =>
    links(pty -> link).insert(puid, nuid)
  case Update((nty, nuid), ovs, nvs) =>
    ovs.foreach { case (link, v) =>
      valLinks(nty -> link).delete(v)
    }
    nvs.foreach { case (link, v) =>
      valLinks(nty -> link).insert(v)
    }
```

Figure 11. Translating edits to EDB insertions and deletions.

Consider the edit script we have seen in the previous subsection. This edit script will be processed into the following insertions and deletions:

$$-\text{Link}_{Add.left}(\#1, \#2)$$

$$+\text{Node}_{Neg}(\#4) \quad +\text{Node}_{Exp}(\#4) \quad +\text{Node}_{Any}(\#4)$$
$$+\text{Link}_{Neg.e}(\#4, \#2)$$

$$+\text{Link}_{Add.left}(\#1, \#4)$$

We separate the insertions and deletions into three packages where each package is emitted by one edit. The detach edit results into a single link relation deletion of *Add.left*. The load edit results in three node relation insertions: *Neg*, *Exp*, and *Any*, one for each supertype of *Neg*. Our implementation assumes that every node type has `Any` as its supertype. Additionally, we insert into the link relation of *Neg.e*. The attach edit results in another link relation insertion of *Add.left*.

6 Evaluation

We implemented our encoding for structural data as part of the incremental Datalog engine IncA. IncA is focused on static program analysis that processes the abstract syntax tree of programs. Our encoding was used to represent those trees and to make incremental changes to them since 2016. Our encoding has been extensively experimented with and we have refined it multiple times, as we report below.

6.1 Evolution of IncA

IncA was introduced in 2016 [30] as a constraint language for incremental static analysis based on Datalog. In 2018, IncA was extended with user-defined recursive aggregation by introducing a new incremental algorithm for Datalog [28]. In 2021, this algorithm was refined to improve the scalability of IncA for whole-program analyses [29]. During this time, IncA was implemented in the JetBrains Meta Programming System (MPS) [1], which uses projectional editing. Projectional editing means that the user edits a view of the program's abstract syntax. To change the tree, the user triggers change requests that are applied to the abstract syntax, which then is re-projected for the user. Within IncA, these change requests were translated to trigger corresponding changes in the EDB, where the encoded structured data resides.

IncA was reimplemented in Scala as an open-source project in 2020. Conceptually, the main change is that IncA does not rely on projectional editing anymore, but uses efficient tree diffing to trigger concise changes [9]. In Section 5, we showed how to process tree-diff patches to update the relational encoding of structured data. The reimplementation also realizes a number of performance improvements: Compiler optimizations like constant folding, constant propagation, and inlining, and virtual indices in the EDB.

6.2 Performance Evaluation

Our encoding has been used in various IncA performance experiments. We provide an overview of these experiments in Figure 12.

The initial IncA paper evaluated the incremental performance of IncA on three program analyses for Java and C [30]. These analyses process the structured source-level code directly, whereas later analyses processed some intermediate format such as Java bytecode in form of Jimple [17]. In all cases, the input language was modeled in MPS as abstract syntax that can be modified programmatically or through

[1] https://www.jetbrains.com/mps

Language	Analysis	Programs	Non-inc. time	Inc. time	Speedup
Java	FindBugs	mbeddr importer (10k LoC)	n/a	7 ms	n/a
C	flow-sensitive points-to	Toyota ITC code[2](15k LoC)	5800 ms	23.3 ms	249x
C	well-formedness checks	Smart Meter (44k LoC)	209 ms	12.8 ms	16.3x
Jimple	Strong-update points-to	GTruth (9k), Gson (14k), PGSQL	6500–64300 ms	1–10 ms	650–6430 x
Jimple	String analysis	JDBC (45k), BerkleyDB (70k)	13500–20400 ms	2 ms	6500–10000x
Jimple	Constant propagation	antlr (22k), emma (26k)	5000–23000 ms	1–3 ms	1600–7600x
Jimple	Interval analysis	pmd (61k), ant (105k)	3000–23000 ms	1–6 ms	500–3800x
Lambda	type checking	synthesized code	6/50 ms	44/2 ms	0.14/25x

Figure 12. IncA performance experiments based on our encoding of structured data.

projectional user interactions. The abstract syntax was then transformed into an EDB schema using the encoding presented in Section 3.

Except the last one, all analyses were written against the encoded structured data in Datalog. That is, the analyses query our Node and Link relations and make use of our option and list encoding. In contrast, the last analysis was written in a type-checking DSL that compiles to Datalog, hiding the details of our encoding from the developer [22]. In general, generating Datalog code seems to become increasingly popular and is possible for completely unrelated programming paradigms such as functional programming [21].

Unfortunately, there is no standard benchmark for incremental source-code changes. Therefore, the IncA performance experiments synthesized changes programmatically, trying to impact the analysis result in order to challenge the incremental engine. The evaluation results demonstrate that our encoding of structured data does not impede incremental performance. Indeed, small source-code changes were translated into small EDB changes, thus triggering a minimal incremental update.

7 Related Work

We propose to encode structured data following three principles: (i) use context-insensitive unique identifiers to reference structured data, (ii) only use unary and binary relations to describe relations between data, and (iii) use relative positioning to encode list data. In the remainder of this section we compare our design with related work. Except where we say so explicitly, the encodings used by related work were not designed with incremental updates in mind.

Doop is a highly configurable family of points-to analyses for Java bytecode written in Datalog [5]. We have already discussed some of Doop's design decisions for encoding bytecode in the introduction. Doop uses large compound relations and context-sensitive identifiers to describe the abstract syntax of Java bytecode. In contrast, we use only unary and binary relations to describe tree-shaped data and use atomic UIDs. Additionally, they use indices to mark the order of list elements such as instructions. We encode lists utilizing first and next links to express the order of list elements relatively.

QL is an object-oriented language based on Datalog primarily used for static analysis [2]. In the front end, QL supports several mainstream languages, including e.g. Java, C/C++, Ruby, and Python. For each one of these languages, a database schema is defined that determines how the EDB structure looks like. For some of the languages, the schema is defined by hand, while for others it is automatically generated from the corresponding TreeSitter grammar. However, it is true in general that the schemas are not compatible with incrementalization because many of the EDB relations use positional information about AST nodes (e.g. index of an argument in a function call) or identifiers that are generated in a context-sensitive manner. In contrast, we store contextual information (e.g., the parent node) in separate relations, so that a node remains stable when it's moved and less information has to be re-derived.

bddbddb [16, 33] is a Datalog engine that represents relations as binary decision diagrams (BDDs). BDDs allow the concise representation of relations with efficient storing and querying. Their work focuses on program analysis and they encode abstract syntax using flat relations. However, they do not follow our design principles as they not only use unary and binary relations to describe the abstract syntax, and they use absolute positioning to encode lists.

DIMPLE [4] and Eichberg et al. [7] use tabled Prolog to describe static analyses, where the latter also supports incrementality. They use nested constructors to describe the abstract syntax. While this is possible when using Prolog, we focus on using Datalog to incrementalize computations, which requires flat relations containing primitive data such as strings, integers, and booleans. Eichberg et al. [7] also uses absolute position indices to encode the instruction ordering, while our approach uses first and next links.

Soufflé is a Datalog framework consisting of a Datalog language and a solver [26]. Recently, Soufflé investigated how to add elastic incrementalization to their solver [34]. Elastic incrementalization reacts based on the impact of the change: Small impact changes trigger incremental updates, and high

[2] https://github.com/regehr/itc-benchmarks

impact changes trigger complete re-computation. In evaluating the incremental performance, Soufflé used Doop analyses as a benchmark including the Doop encoding of structured data. However, they did not map actual code changes to EDB updates, which would inhibit the incremental performance as we explained above. Instead, they modified individual EDB tuples directly, although this does not correspond to any actual code change. We expect that Soufflé could use our encoding of structured data to improve their incremental performance on real code changes.

The work on query shredding [6, 14, 27] tackles a problem of nested queries over nested data in the form of bags. These queries return nested data itself. In our work we only consider nested data in the form of tree-shaped data that is the input of Datalog programs. Our work does not focus on generating tree-shaped data during the evaluation of Datalog programs. The derived tuples can reference tree-shaped data in the form of unique identifiers, but these unique identifiers only ever reference tree-shaped data that is part of the extensional database. Hence, we do not encounter the problem of deep updates that query shredding tries to solve.

TreeToaster [3] proposes to implement compiler optimizations by applying incremental view maintenance on abstract syntax trees (ASTs) for pattern rewritings. They encode ASTs using relations as well. They also identify AST nodes with unique identifiers. However, they use n-ary tuples to describe ASTs nodes. We propose to only use at most binary relations to encode tree-shaped data to enable small insertions and deletions when editing a single child node. TreeToaster does not allow recursive pattern matches while we integrated our work with state-of-the-art Datalog solvers which allow processing highly recursive Datalog programs and terminate even in the presence of cyclic data.

Adapton [12] is a general-purpose programming language to describe incremental computations. The language is ML-like and supports algebraic data types to encode structured data. While it is natural for Adapton to describe structured data, the challenge they encounter is to incrementalize computations. Our work uses Datalog which has an incremental semantics since the 1990's [11]. In contrast, describing structured data in Datalog is not natural and, in particular, it is not obvious how to describe structured data such that it enables efficient incremental performance. Hence, our work tackles incremental computing from a different angle.

There are other incrementalization techniques such as memoization [1, 18, 23]. Memoization reuses a result when the input of a function does not change, even if the changed part of the input does not contribute to the result. Hence, if one part of the structured data changes, memoization requires to redo the computation. In contrast, Datalog is more fine-grained in the incremental update propagation and our encoding of structured data retains this granularity. That is, we only update a computation's result if relevant parts of the input data changes.

8 Conclusion

In this paper, we showed how to process structured data in Datalog incrementally. Specifically, we presented an encoding of structured data in flat relations that is amenable to incremental updates such that small changes to the structured data translate to small changes of the relations. In doing so, we discovered and followed three design principles: (i) use context-insensitive unique identifiers, (ii) use unary and binary relations only, and (iii) use relative positioning for list elements. We also showed how to implement the relations required by our encoding efficiently, exploiting invariants about the structural data to optimize time and memory. Finally, we explained how users can provide and update the encoded structural data using the tree-diffing algorithm *truediff*, which finds structural differences between trees efficiently and describes them concisely. Our encoding has been part of the IncA framework since 2016, but this paper presents its first systematic description.

References

[1] Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. 1996. Analysis and Caching of Dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 83–91. https://doi.org/10.1145/232627.232638

[2] 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, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[3] Darshana Balakrishnan, Carl Nuessle, Oliver Kennedy, and Lukasz Ziarek. 2021. TreeToaster: Towards an IVM-Optimized Compiler. 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, 155–167. https://doi.org/10.1145/3448016.3459244

[4] William C. Benton and Charles N. Fischer. 2007. Interactive, scalable, declarative program analysis: from prototype to implementation. In *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 14-16, 2007, Wroclaw, Poland*, Michael Leuschel and Andreas Podelski (Eds.). ACM, 13–24. https://doi.org/10.1145/1273920.1273923

[5] 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.

[6] James Cheney, Sam Lindley, and Philip Wadler. 2014. Query shredding: efficient relational evaluation of queries over nested multisets. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 1027–1038. https://doi.org/10.1145/2588555.2612186

[7] Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. 2007. Automatic Incrementalization of Prolog Based Static Analyses. In *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007 (Lecture Notes in Computer Science, Vol. 4354)*, Michael Hanus (Ed.).

Springer, 109–123. https://doi.org/10.1007/978-3-540-69611-7_7

[8] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A Sound and Optimal Incremental Build System with Dynamic Dependencies. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 89–106.

[9] Sebastian Erdweg, Tamás Szabó, and André Pacak. 2021. Concise, type-safe, and efficient structural diffing. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 406–419. https://doi.org/10.1145/3453483.3454052

[10] Ashish Gupta and Inderpal Singh. Mumick. 1999. *Materialized views : techniques, implementations, and applications*. MIT Press. 589 pages. https://mitpress.mit.edu/books/materialized-views

[11] Ashish Gupta, Inderpal Singh Mumick, and V S Subrahmanian. 1993. Maintaining views incrementally. In *Proceedings of Conference on Management of Data (SIGMOD)*. ACM Press, New York, New York, USA, 157–166. https://doi.org/10.1145/170035.170066

[12] Matthew A. Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 748–766. https://doi.org/10.1145/2814270.2814305

[13] 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.

[14] Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 75–90. https://doi.org/10.1145/2902251.2902286

[15] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable incremental building with dynamic task dependencies. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. ACM Press, New York, New York, USA, 76–86. https://doi.org/10.1145/3238147.3238196

[16] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, Chen Li (Ed.). ACM, 1–12. https://doi.org/10.1145/1065167.1065169

[17] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15. 35.

[18] Yanhong A. Liu and Tim Teitelbaum. 1995. Systematic Derivation of Incremental Programs. *Sci. Comput. Program.* 24, 1 (1995), 1–39. https://doi.org/10.1016/0167-6423(94)00031-9

[19] David Maier, K Tuncay Tekle, Michael Kifer, and David Scott Warren. 2018. Datalog: concepts, history, and outlook. In *Declarative Logic Programming: Theory, Systems, and Applications*, Michael Kifer and Yanhong Annie Liu (Eds.). ACM / Morgan & Claypool, 3–100. https://doi.org/10.1145/3191315.3191317

[20] Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. 2014. i3QL: Language-Integrated Live Data Views. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 417–432.

[21] André Pacak and Sebastian Erdweg. 2022. Functional Programming with Datalog. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.7

[22] André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A systematic approach to deriving incremental type checkers. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 127:1–127:28. https://doi.org/10.1145/3428195

[23] William W. Pugh and Tim Teitelbaum. 1989. Incremental Computation via Function Caching. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 315–328. https://doi.org/10.1145/75277.75305

[24] G Ramalingam and Thomas Reps. 1993. A categorized bibliography on incremental computation. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, New York, USA, 502–510. https://doi.org/10.1145/158511.158710

[25] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019 (CEUR Workshop Proceedings, Vol. 2368)*, Mario Alviano and Andreas Pieris (Eds.). CEUR-WS.org, 56–67.

[26] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 196–206. https://doi.org/10.1145/2892208.2892226

[27] Jaclyn Smith, Michael Benedikt, Milos Nikolic, and Amir Shaikhha. 2020. Scalable Querying of Nested Data. *Proc. VLDB Endow.* 14, 3 (2020), 445–457. https://doi.org/10.5555/3430915.3442441

[28] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 139:1–139:29. https://doi.org/10.1145/3276509

[29] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1–15. https://doi.org/10.1145/3453483.3454026

[30] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. https://doi.org/10.1145/2970276.2970298

[31] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. 2016. Road to a Reactive and Incremental Model Transformation Platform: Three Generations of the VIATRA Framework. *Softw. Syst. Model.* 15, 3 (jul 2016), 609–629. https://doi.org/10.1007/s10270-016-0530-4

[32] Tim A. Wagner and Susan L. Graham. 1997. Incremental Analysis of real Programming Languages. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, Nevada, USA, June 15-18, 1997*, Marina C. Chen, Ron K. Cytron, and A. Michael Berman (Eds.). ACM, 31–43. https://doi.org/10.1145/258915.258920

[33] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June*

9-11, 2004, William W. Pugh and Craig Chambers (Eds.). ACM, 131–144. https://doi.org/10.1145/996841.996859

[34] David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. 2021. Towards Elastic Incrementalization for Datalog. In *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, Niccolò Veltri, Nick Benton, and Silvia Ghilezan (Eds.). ACM, 20:1–20:16. https://doi.org/10.1145/3479394.3479415

Received 2022-08-12; accepted 2022-10-10