Automatic Index Selection for Large-Scale Datalog Computation

Pavle Subotić[†], Herbert Jordan[‡], Lijun Chang[§], Alan Fekete[§], Bernhard Scholz[§]
[†] University College London, [‡] University of Innsbruck, [§] The University of Sydney
[†] pavle.subotic.15@ucl.ac.uk, [‡] herbert.jordan@uibk.ac.at
[§] {lijun.chang, alan.fekete, bernhard.scholz}@sydney.edu.au

ABSTRACT

Datalog has been applied to several use cases that require very high performance on large rulesets and factsets. It is common to create indexes for relations to improve search performance. However, the existing indexing schemes either require manual index selection or result in insufficient performance on very large tasks. In this paper, we propose an automatic scheme to select indexes. We automatically create the minimum number of indexes to speed up all the searches in a given Datalog program. We have integrated our indexing scheme into an open-source Datalog engine SOUFFLÉ. We obtain performance on a par with what users have accepted from hand-optimized Datalog programs running on state-of-the-art Datalog engines, while we do not require the effort of manual index selection. Extensive experiments on large real Datalog programs demonstrate that our indexing scheme results in considerable speedups (up to 2x) and significantly less memory usage (up to 6x) compared with other automated index selections.

PVLDB Reference Format:
Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, Bernhard Scholz. Automatic Index Selection for Large-Scale Datalog Computation. *PVLDB*, 12(2): 141-153, 2018.
DOI: https://doi.org/10.14778/3282495.3282500

1. INTRODUCTION

There has been a resurgence in the use of Datalog in several computer science communities [18], including program analysis where it is used as a domain specific language for succinctly specifying various classes of static analyses. In this setup, an input program to be analyzed is converted into an extensional database (EDB), while the analysis specification is encoded as a set of Datalog *rules* that compute the analysis result as an intensional database (IDB).

Example 1. *Figures 1a and 1b depict a simplified taint analysis encoded as a* Datalog *program, used for detecting the vulnerabilities of a web-based hospital management system. The source code of the management system is converted into EDB relations, e.g.,* `Src`*,* `Sink`*,* `Role`*,* `Access`*,* `Zone`*, and* `Priv`*, where relations* `Role` *and* `Access` *for access policy are shown in Figure 1a.*

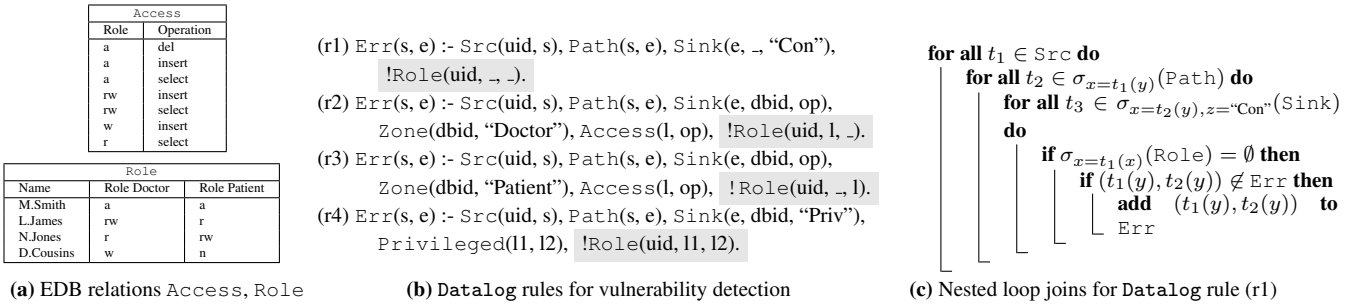
Proceedings of the VLDB Endowment, Vol. 12, No. 2
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3282495.3282500

Datalog *rules are constructed for the security analysis, enumerating all possible vulnerability cases. This part of the ruleset is shown in Figure 1b; we omit the* Datalog *rules for computing the IDB relation* `Path` *that defines the control flow of the source code. For example, the first rule adds an error involving source code locations s and e to the appropriate IDB relation, whenever s is a user (uid) input (Src) for which there exists a program path to e, a database connection (Sink) location that has no Role for uid. That is, the user has connected to the database without a role. Note that, as conventional, underscore is used for anonymous variables whose values are not important. The results of the analysis, determining the set of error paths, are stored in the IDB relation* `Err`*.*

Such use cases of Datalog in program analysis, typically consist of hundreds of rules and result in giga-tuple sized IDB relations, as shown in [21, 37]. Several specialized high-performance Datalog engines [19, 21, 24] have been employed for performing such computations. These engines use Datalog as a computational notation, and exploit bottom-up evaluation techniques that usually involve some degree of compilation. To reduce lookup times, relations are stored as *in-memory, index-organized tables* [24, 36]. Selecting the appropriate indexes in this setting requires novel techniques, compared to those standard for physical design in relational platforms.

The theory of the index selection problem (ISP) for relational database management systems [12, 20, 22, 31] uses variants of the 0-1 knapsack problem, which has been shown to be NP-hard [26]. Deployed approaches such as [11] use heuristics and integrate with what-if query optimization calculations. These techniques are surveyed in Bruno [8], but they are too computationally expensive for large Datalog analyses. Essential differences include (i) indexes are needed for both EDB and IDB relations, (ii) the Datalog relations are often wide (not normalized), and thus they offer a very large number of possible indexes, and (iii) the Datalog programs typically consist of *hundreds of relations and hundreds of deeply nested rules* (see Table 2 in Section 7). As a result, the specialized Datalog engines often require users to provide annotations to guide the choice of indexes; for example, the DOOP framework [37] uses a code-rewriting technique that *manually* chooses an index for each relation and introduces "Opt" relations for building multiple indexes on a relation. To allow widespread use of program analysis, we must move beyond approaches that put the optimization burden on the user, who requires painstaking trial and error over hundreds of rules and annotations.

Auto-Indexing. In this paper, we formulate an *automatic* indexing scheme for Datalog computations, aiming to achieve the best performance/memory usage while not requiring the intervention of end users. Our approach was motivated by experiences with industry use cases involving large-scale program analysis performed with the state-of-the-art compilation-based Datalog en-



Access

Role	Operation
a	del
a	insert
a	select
rw	insert
rw	select
w	insert
r	select

Role

Name	Role Doctor	Role Patient
M.Smith	a	a
L.James	rw	r
N.Jones	r	rw
D.Cousins	w	n

(r1) Err(s, e) :- Src(uid, s), Path(s, e), Sink(e, _, "Con"), !Role(uid, _, _).
(r2) Err(s, e) :- Src(uid, s), Path(s, e), Sink(e, dbid, op), Zone(dbid, "Doctor"), Access(l, op), !Role(uid, l, _).
(r3) Err(s, e) :- Src(uid, s), Path(s, e), Sink(e, dbid, op), Zone(dbid, "Patient"), Access(l, op), !Role(uid, _, l).
(r4) Err(s, e) :- Src(uid, s), Path(s, e), Sink(e, dbid, "Priv"), Privileged(l1, l2), !Role(uid, l1, l2).

for all $t_1 \in$ Src **do**
  for all $t_2 \in \sigma_{x=t_1(y)}$(Path) **do**
    for all $t_3 \in \sigma_{x=t_2(y),z=\text{"Con"}}$(Sink) **do**
      if $\sigma_{x=t_1(x)}$(Role) $= \emptyset$ **then**
        if $(t_1(y), t_2(y)) \notin$ Err **then**
          add $(t_1(y), t_2(y))$ **to** Err

(a) EDB relations Access, Role

(b) Datalog rules for vulnerability detection

(c) Nested loop joins for Datalog rule (r1)

Figure 1: Example Datalog analysis for vulnerability detection

gine SOUFFLÉ [21]. We found inadequate performance until we introduced our new technique into SOUFFLÉ, however the ideas should apply more broadly to any engine that computes a Datalog program in successive phases. That is, initially there are analysis phases that consider only the rules and produce code to perform a query evaluation plan resembling a nested loop join, and these are followed by an evaluation phase that executes the compiled query on the facts (i.e., IDB), producing a materialized IDB. Our auto-indexing is conducted at one of the analysis phases, and it chooses indexes that improve the performance of the compiled code.

The key insights of our work are as follows. We identify that the compiled evaluation is built from frequently repeated calls to simple selections, each on a single relation (which might be in EDB or in IDB). We call these *primitive searches*, and a primitive search returns the tuples in a relation which satisfy a predicate that involves testing some of the attributes for equality to a given value. For example, Figure 1c depicts the evaluation logic that is compiled for the Datalog rule (r1) in Figure 1b where the first, second, and third attributes of a relation are assumed to be accessed by x, y, and z, respectively. There are three primitive searches $\sigma_{x=t_1(y)}(\texttt{Path})$, $\sigma_{x=t_2(y),z=\text{"Con"}}(\texttt{Sink})$, and $\sigma_{x=t_1(x)}(\texttt{Role})$, where the first one looks up all tuples in relation Path whose first attribute value is equal to $t_1(y)$ — the second attribute value of a tuple t_1 from relation Src. Note that each primitive search is a very restricted kind of range query: for each attribute, we are either checking equality to a value, or else we accept any value in that attribute.

Our next insight is that the evaluation of a primitive search can be greatly sped up if the relation has a clustered B-tree index that *covers* the search predicate. This means that the set of attributes where equality is checked, forms a prefix of the sequence of attributes used to lexicographically define the index. For example, the primitive search $\sigma_{x=v_1,z=v_3}$ is covered by the index $\ell = x \prec z$ (that means, an index using x followed by z as its key) but not by $\ell' = x \prec y \prec z$. When a search is covered by an index, the tuples that match the search are a contiguous part of the scan of the index leaves. Accessing these can be much faster than a full table scan, which is what an engine would use in the absence of an index. Because the relations are so large, we find that queries are typically infeasible in practice unless there is some index to cover every primitive search among the rules. On the other hand, each index uses considerable space, and so we are driven to minimize the number of indexes constructed. Thus we define an abstract task, the *Minimum Index Selection Problem* (MISP), aiming to select the *minimum* number of indexes to cover all primitive searches used in the ruleset. We notice that this can be significantly fewer than one index for each primitive search on the relation. For example, the index $\ell = x \prec y \prec z$ covers three primitive searches: $S_1 = \sigma_{x=v_1}$, $S_2 = \sigma_{x=v'_1,y=v'_2}$, and $S_3 = \sigma_{x=v''_1,y=v''_2,z=v''_3}$.

Finally, we are able to solve the MISP efficiently, using a relationship between the search space of indexes and the search space of *search chains* among lexicographic orders. To do so, we abstract each primitive search as its set of search attributes, referred to as a *search*; for example, $S_1 = \{x\}, S_2 = \{x, y\}$, and $S_3 = \{x, y, z\}$ are the searches corresponding to the above three primitive searches. A sequence of k searches $S_1, \ldots, S_k$ form a search chain if each search S_i is a proper subset of its immediate successor search S_{i+1}. As a result, all searches in the same search chain can be covered by a single index. We prove that the optimal MISP solution can be constructed from the optimal set (i.e., with the minimum cardinality) of search chains that cover all primitive searches. Then we apply the combinatorial result of Dilworth's theorem [15] to compute the minimum number of search chains, and thus the minimum number of indexes, in $\mathcal{O}(|\mathcal{S}|^{2.5} + |\mathcal{S}|^2 \cdot m)$ time, for a set $\mathcal{S}$ of primitive searches on a relation with m attributes. This is much faster than a brute force examination of all possible sets of indexes on this relation, which would have a time complexity of $\mathcal{O}(2^{m^m})$.

We have implemented our index selection approach as the default indexing technique of the SOUFFLÉ Datalog engine. We found that the computation overhead for our index selection is negligible, i.e., no slowdowns were observed during compilation. Using our technique, SOUFFLÉ has managed to efficiently compute program analyses typically deemed too large for Datalog engines, and moreover, the performance exhibited by SOUFFLÉ has been on a par with recent state-of-the-art hand-crafted analyzers [14].

Contributions. Our contributions are summarized as follows.

- We formally define the minimum index selection problem (MISP) to find the minimum number of indexes to cover all primitive searches.
- We present a polynomial-time algorithm to solve MISP optimally via computing search chains.
- We formulate an automatic indexing scheme for large-scale Datalog computation based on this theory.
- We demonstrate the effectiveness of our indexing scheme in an open-source Datalog engine, SOUFFLÉ, with large, real world rulesets and factsets.

Note that this paper builds on prior work [33, 21] by some of the same authors. In [33] we introduce an overview of SOUFFLÉ as a compilation framework for Datalog, and the paper [21] is a tool paper focusing on the synthesis of C++ via Futamura projections. Both papers mention index selection, however, neither the theory nor the implementation details of index selection is filled in by prior papers. This work introduces the formal problem definition of MISP, precise algorithms for solving MISP, brute-force estimates, proof sketches, and evaluation of MISP in comparison to other index selection techniques.

We remark that our scheme is based on clustered B-tree index structures kept in-memory. If multiple indexes are needed, we materialize replicas of the relation so that each index can be clustered.

Organization. The paper is organized as follows. We highlight related works in Section 2, and present preliminary definitions in Section 3. In Section 4, we introduce an automatic indexing scheme and formally define the minimum index selection problem (MISP). In Section 5, we present a polynomial-time algorithm to solve MISP optimally. We evaluate our automatic indexing scheme in an open-source `Datalog` engine in Section 7. We discuss other extensions of our techniques in Section 8 and draw relevant conclusions in Section 9.

2. RELATED WORK

`Datalog` **Engines.** `Datalog` has been pro-actively researched in several computer science communities [9, 28, 29, 30], where a comprehensive introduction to `Datalog` can be found in [1]. Driven by applications in data integration, networking, and program analysis, `Datalog` has recently regained considerable interests, e.g., see [18] for a survey of these developments. Logicblox [3] is a commercial propitiatory system which focuses on encoding business logic. The latest version 4 of LogicBlox is single-threaded execution and less amenable for recursive queries. Hence, Logicblox cannot be directly employed for highly-recursive workloads occurring in static program analysis. BigDatalog [36] is a Datalog system that executes queries on the unified analytics engine Apache Spark. The system is designed for recursive aggregate queries and applications typically found in social network and other data-analytics application with large-data. The aim of BigDatalog is to exploit coarse-grain parallelism in Datalog programs. Static program analysis and security analysis have different workload characteristics requiring a fine-grain parallelism caused by a large number of mutual recursive relations with several hundred rules. Datalog-MC [39] uses an in-memory parallel evaluation of Datalog programs on shared-memory multi-core machines. Datalog-MC hash-partitions tables and executes the partitions on cores of a shared-memory multi-core system using a variant of hash-join. To parallel evaluate Datalog, Datalog rules are represented as and-or trees that are compiled to Java. Flix [25] is a new Datalog-inspired domain specific language for static program analysis extending the expressiveness of Datalog with arbitrary lattice structures. Flex does not have as a research objective performance rather expressiveness.

Other Datalog platforms. Note that the use-cases of static program analysis requires particular capabilities in `Datalog` engines including fast fixed-point calculations for highly mutual recursive relations with very deep joins, domain specific extensions of Datalog including complex element types, components, and widening-techniques. Various engines have been used for static program analysis including Logicblox version 3 [23], μZ [19], bddbddb [38], and SOUFFLÉ [21], which is currently the state-of-the-art `Datalog` engine used in Java points-to [6], Amazon's AWS cloud, and Smart-Contract analysis [17].

Index Selection in `Datalog` Engines. Consider `PA-Datalog`, which is a variant of Logicblox version 3, and has been used in DOOP for program analysis. This engine stores each relation (whether EDB or IDB) in an index, where the structure is based on the order of attributes as listed in the relation. As shown in [7], execution efficiency of DOOP can be greatly improved by a manual code-rewriting technique [2], which replicates a relation multiple times (corresponding to attributes listed in different orders) and thus it creates a distinct index for each replica. This manual index creation, although resulting in an enormous speedup [7], requires end-users to be familiar with the underlying indexing mechanism of a `Datalog` engine. The manual code-rewriting technique is error-prone and consumes much human time and effort, on programs with hundreds of rules. Also, the hand-optimized `Datalog` rule-sets become obfuscated, and maintainability and readability are hampered. In contrast, we seek an automated approach to identifying appropriate index structures.

In bddbddb the system chooses a global variable order, and indexes each relation once, according to the restriction of the global order to the attributes of the relation. This means that many searches are not able to use the index with an increasing number of rules and relations occurring in standard static program analysis workloads.

Index Selection in Relational Databases. A recent monograph on optimizing performance of SQL queries is given by Bruno [8]. One aspect is physical design, including the choice of index structures. In the context of relational databases, the problem of automatically selecting indexes for a set of database queries, referred to in the literature as the index selection problem (ISP) [12, 20, 22, 31], is well studied and has been shown to be NP-hard [26]. It is typically formulated as a variant of the 0-1 knapsack problem, which balances the overall execution time of queries for an index configuration (i.e., a subset of indexes that influence the performance of a query) and the cost of index maintenance. Our index selection problem differs from the classic ISP literate and to the best of our knowledge is the first formulation for `Datalog`. Firstly, in our case, we only need to support primitive searches, which occur in equi-joins and simple value queries. Secondly, the nature of `Datalog` restricts the search predicate of each primitive search to be an equality predicate over the attributes of the relation. We further assume that each primitive search benefits from being indexed. Thus, we formulate our problem as automatically selecting the minimum number of indexes to cover all searches, and we show that unlike the relational problem, we can solve it optimally in polynomial time. In contrast to semi-automatic techniques such as WFIT index tuning algorithm [32] that are aimed at relational databases, our approach is designed to be fully automatic and computed on the fly at compilation time. Offline index selection approaches aimed such as AutoAdmin [10] relay recommendations to a DBA by performing a cost based analyses of a workload. The DBA then makes the final selection based on the feedback. Our approach is designed to automatically select the optimal index set on very large rulesets, and, hence, is designed to scale to large Datalog programs by having minimal overheads. Our algorithm however, can be used in conjunction with a manual or automatic join selection algorithm to provide an additional optimal index set cost metric to aid in general query optimization.

3. PRELIMINARIES

Like database queries, `Datalog` programs also work on relations. A *relation* R is a subset of an m-ary Cartesian product $\mathcal{D} = \mathbb{D}_1 \times \cdots \times \mathbb{D}_m$ (i.e., $R \subseteq \mathcal{D}$), where $\mathbb{D}_i$ $(1 \leq i \leq m)$ are the *domains* of the relation. Elements of a relation R are referred to as *tuples*. Each tuple $t = \langle e_1, e_2, \ldots, e_m \rangle \in R$ has a fixed length m, and e_i is an element of the domain $\mathbb{D}_i$ for $1 \leq i \leq m$.

Given a relation R, *attributes* are used to refer to specific element positions of tuples of R. The set of attributes of R, denoted by $A_R = \{x_1, \ldots, x_m\}$, are m distinct symbols, and we write $R(x_1, \ldots, x_m)$ to associate symbol x_i to the i-th position in the tuples. The elements of a tuple $t = \langle e_1, \ldots, e_m \rangle$ can be accessed

by *access function* $t(x_i)$ that maps tuple t to element e_i. For example, given a relation $R(x, y, z)$ and a tuple $t = \langle e_1, e_2, e_3 \rangle \in R$, the access function is $\{t(x) \mapsto e_1, t(y) \mapsto e_2, t(z) \mapsto e_3\}$.

3.1 Datalog Program Computation

A `Datalog` program P consists of a finite set of `Datalog` *rules* $\{r_1, r_2, \ldots\}$, each of the form:

$$r : R_0(X_0) \leftarrow R_1(X_1), R_2(X_2), \ldots, R_d(X_d).$$

Each $R_j(X_j)$ is called an *atom*, where R_j is a relation name and X_j is a sequence of constants, variables, and symbol "_" indicating irrelevance; for example, $R(u, \_, 1)$ where u is a variable. $R_0(X_0)$ is called the *head* of the rule, and other atoms form the *body* of the rule. The semantic meaning of a `Datalog` rule is that given a binding of all variables to constants, the head of the rule holds if each atom in the body of the rule holds. In this paper, we allow negated predicates in the body, but we limit its usage by the semantics of *stratified* `Datalog` (see [1] for the details of stratified `Datalog`).

The set of relations that appear in the heads of P's rules are referred to as the intensional database (IDB), while the set of other relations are referred to as the extensional database (EDB). In a `Datalog` program, tuples of the EDB are given, and the system computes tuples of the IDB. This is typically achieved by a bottom-up evaluation of the set of rules [1]. In brief, the process starts from an instance I of P that consists only of EDB tuples (also called facts). Then, an *immediate consequence operator* Γ_P is repeatedly applied to I to generate new IDB tuples to be included into I. The process completes when a fixed-point is reached, i.e. no more IDB tuples can be generated.

Primitive Search for `Datalog` Rule Evaluation. In the bottom-up evaluation process, a `Datalog` rule is typically evaluated via nested loop joins. For presentation simplicity, we partition the sequence of body atoms of a `Datalog` rule into positive (referred to as R_i^+) and negative (referred to as R_j^-) occurrences (i.e., negative if it is negated in the body), and restate the above `Datalog` rule as

$$R_0(X_0) \leftarrow R_1^+(X_1), \ldots, R_h^+(X_h), R_{h+1}^-(X_{h+1}), \ldots, R_d^-(X_d),$$

where h is the number of positive atoms. Then, this `Datalog` rule is evaluated via *nested loop joins*, as shown in Figure 2. Note, the ordering may change due to leveling, i.e, negative predicates hoisted to outer loops for performance reasons.

```
loop_1: for all t_1 ∈ σ_{φ_1(X_1)}(R_1^+) do
loop_2:   for all t_2 ∈ σ_{φ_2(t_1,X_2)}(R_2^+) do
  ...       ...
loop_h:     for all t_h ∈ σ_{φ_h(t_1,...,t_{h-1},X_h)}(R_h^+) do
              if σ_{φ_{h+1}(t_1,t_2,...,t_h)}(R_{h+1}^-) = ∅ then
                ...
                if σ_{φ_d(t_1,t_2,...,t_h)}(R_d^-) = ∅ then
                  if π(t_1, ..., t_h) ∉ R_0 then
                    add π(t_1, ..., t_h) to R_0
```

Figure 2: Nested loop joins for evaluating a `Datalog` rule

In the nested loop joins, we iterate over tuples that are obtained from a *primitive search*, which will be defined shortly, on a positive relation. Then, negative occurring atoms are tested for emptiness with respect to primitive searches. Finally, the appropriate attributes of the tuples involved in the current iteration are projected, and a new tuple is inserted into the IDB relation for the head atom of the rule, if that tuple is not already in the relation.

A vital benefit of the nested loop implementation is its memory efficiency. At any time, the system stores the current tuples of the primitive searches on h relations only; there is no need to fully materialize the intermediate results of joining a prefix of the set of relations. The size of intermediate results could easily exceed the sizes of the eventual IDB tables.

Definition 1 (PRIMITIVE SEARCH)**.** *A primitive search has the following form:*

$$\sigma_{x_1=v_1,\ldots,x_k=v_k}(R_i) = \{t \in R_i \mid t(x_1) = v_1, \ldots, t(x_k) = v_k\}.$$

Here, R_i is a relation and $x_1 = v_1, \ldots, x_k = v_k$ is a search predicate*, where $x_1, \ldots, x_k$ are attributes and $v_1, \ldots, v_k$ are constants.*

A primitive search extracts all tuples from a relation that adhere to the *search predicate*. In this paper, we limit the search predicate to be equalities of left-hand-side attributes and right-hand-side constants as it holds for all the real `Datalog` programs we tested in Section 7. Note that in our notation $\{x_1, \ldots, x_k\}$ does not necessarily have to consist of the first k attributes of the relation R_i, and the constants $v_1, \ldots, v_k$ are obtained either from X_i or from other tuples in relations further up the nested loop joins (i.e., $t_1, \ldots, t_{i-1}$ in Figure 2)

Speeding Up Primitive Searches via Indexing Relations. After constructing the nested loop joins for all rules in a `Datalog` program, the most critical factor to the performance of evaluating the `Datalog` program is how the primitive searches are conducted. Obviously, a primitive search can be achieved by conducting a *linear scan* of all tuples of the relation and checking the search predicate against each tuple. However, the time complexity of linear scan over a relation with n tuples is $\mathcal{O}(n)$, which is too costly for large relations considering that each primitive search is invoked repeatedly many times. In this paper, we aim at creating indexes for relations to speed up the primitive searches, and we study the following problem whose formal definition will be given in Section 4.

Problem 1. Given the primitive searches in the nested loop joins of all rules in a `Datalog` program, we study the problem of creating indexes for relations to speed up all the primitive searches.

4. INDEXING RELATIONS

In this section, we first introduce indexes to speed up primitive searches, and then formally define our problem of minimum index selection.

4.1 From Primitive Search to Lex Search

To enable indexes on a relation, we introduce an order among tuples in a relation to make them comparable. Since a tuple may have several elements, an order of tuples is imposed by element-wise comparison using a sequence over all attributes of the relation; this comparison is known as a *lexicographical order*. We denote an attribute sequence by $\ell = x_1 \prec x_2 \prec \cdots \prec x_m$ where $\prec$ denotes a chaining of elements to form a sequence. Then, given ℓ that is formed by all attributes of a relation, a lexicographical order $\sqsubseteq_\ell \mathcal{D} \times \mathcal{D}$ is a total order (i.e., *reflexive, asymmetric, transitive*) defined over the domain $\mathcal{D}$ of the relation with respect to ℓ. For two tuples $a, b \in \mathcal{D}$, when $(a, b) \in \sqsubseteq_\ell \mathcal{D} \times \mathcal{D}$, we write $a \sqsubseteq_\ell b$ and we say that a is smaller than b with respect to ℓ. Note that $a \sqsubseteq_\ell a$, and for any two different tuples $a, b \in \mathcal{D}$, we either have $a \sqsubseteq_\ell b$ or $b \sqsubseteq_\ell a$ but not both.

Given an ordered set of tuples, tuple lookups can be performed efficiently using some notion of a balanced search tree, called an *index*, in which tuples can be found in logarithmic time rather than

Table 1: Primitive and Lex searches for relation `Role` in the nested loop joins for rules (r1)–(r4) in Figure 1b

Literal	Primitive Search	Lex Search Predicate $\rho(\ell, a, b)$			
		Lower Bound a	Upper Bound b	Naïve ℓs	Minimum ℓs
$\texttt{Role}(v_1, \_, \_)$	$\sigma_{x=v_1}$	$\langle v_1, \bot, \bot\rangle$	$\langle v_1, \top, \top\rangle$	x	$x \prec y \prec z$
$\texttt{Role}(v_1, v_2, \_)$	$\sigma_{x=v_1, y=v_2}$	$\langle v_1, v_2, \bot\rangle$	$\langle v_1, v_2, \top\rangle$	$x \prec y$	$x \prec y \prec z$
$\texttt{Role}(v_1, \_, v_3)$	$\sigma_{x=v_1, z=v_3}$	$\langle v_1, \bot, v_3\rangle$	$\langle v_1, \top, v_3\rangle$	$x \prec z$	$x \prec z$
$\texttt{Role}(v_1, v_2, v_3)$	$\sigma_{x=v_1, y=v_2, z=v_3}$	$\langle v_1, v_2, v_3\rangle$	$\langle v_1, v_2, v_3\rangle$	$x \prec y \prec z$	$x \prec y \prec z$

in linear time. In this paper, *we abstract away the underlying implementation details of an index with an attribute sequence*, and we use ℓ to denote both an index and the attribute sequence based on which the index is constructed. It is worth mentioning that different attribute sequences usually result in different lexicographical orders, and thus different indexes. That is, for tuples $a, b \in \mathcal{D}$ and attribute sequences ℓ and ℓ', it is possible that $a \sqsubseteq_\ell b$ and $b \sqsubseteq_{\ell'} a$.

Given an index ℓ, we define a lex search as follows.

Definition 2 (LEX SEARCH)**.** *A lex search $\sigma_{\rho(\ell,a,b)}$ is defined for a relation $R \subseteq \mathcal{D}$ and its semantics is given by,*

$$\sigma_{\rho(\ell,a,b)}(R) = \{t \in R \mid a \sqsubseteq_\ell t \sqsubseteq_\ell b\}.$$

$\rho(\ell, a, b)$ is a lex search predicate*, where ℓ is an index on R, and the lower bound a and the upper bound b are tuples in $\mathcal{D}$.*

Constructing Lex Searches from Primitive Searches. As lex searches can be efficiently conducted based on an index, we would like to transform each primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$ into an equivalent lex search $\sigma_{\rho(\ell,a,b)}(R)$. A lex search contains two symbolic bounds a and b, as well as an index ℓ, in the lex search predicate. Thus, we need to construct a, b, and ℓ, which will be discussed in the following. We assume that the relation R has m attributes in total.

Firstly, we describe how to construct the lower bound a and the upper bound b. If $k = m$, then all attributes of R are in the search predicate, and $a = b$ and they are trivially defined by the search predicate. Otherwise, the primitive search does not specify all attributes of R in its search predicate, and unspecified values need to be padded with infima and suprema values for lower and upper bounds, respectively. We define an unspecified element for the lower/upper bound construction by an artificial constant $\triangle$, and let $v_{k+1} = \triangle$. We assume that $\triangle$ is not element of any of the domains $\mathbb{D}_i$. We define a surjective index mapping function $i : \{1, \ldots, m\} \to \{1, \ldots, k+1\}$ that maps the specified elements to their corresponding constant values, and maps the unspecified elements to $\triangle$ (i.e., v_{k+1}). The construction of the lower and upper bound is performed by the functions *lb* and *ub*, respectively,

$$a = \mathit{lb}(v_1, \ldots, v_k)$$
$$b = \mathit{ub}(v_1, \ldots, v_k)$$

that replace the unspecified $\triangle$ value with the infimum $\bot_j$ and the supremum $\top_j$ of the domain $\mathbb{D}_j$, respectively. Formally, the functions are defined as $\mathit{lb}(v_1, \ldots, v_k) = \langle v'_1, \ldots, v'_m\rangle$ where

$$v'_j = \begin{cases} v_{i_j} & \text{if } v_{i_j} \neq \triangle \\ \bot_j & \text{otherwise} \end{cases}$$

and $\mathit{ub}(v_1, \ldots, v_k) = \langle v''_1, \ldots, v''_m\rangle$ where

$$v''_j = \begin{cases} v_{i_j} & \text{if } v_{i_j} \neq \triangle \\ \top_j & \text{otherwise} \end{cases}$$

Secondly, we show in Lemma 1 that given $a = \mathit{lb}(v_1, \ldots, v_k)$ and $b = \mathit{ub}(v_1, \ldots, v_k)$, we have $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R) = \sigma_{\rho(\ell,a,b)}(R)$ if the k-th prefix of l is $\{x_1, \ldots, x_k\}$. Before that, we first define prefix set.

Definition 3 (PREFIX SET)**.** *Given an attribute sequence (i.e., an index) $\ell = x_1 \prec x_2 \prec \cdots \prec x_m$, the k-th prefix of ℓ is $\{x_1, \ldots, x_k\}$ if $k \leq m$, and it is $\{x_1, \ldots, x_m\}$ otherwise.*

Lemma 1. *Given $a = \mathit{lb}(v_1, \ldots, v_k)$, $b = \mathit{ub}(v_1, \ldots, v_k)$, and an index ℓ whose k-th prefix is $\{x_1, \ldots, x_k\}$, then*

$$\sigma_{x_1=v_1,\ldots,x_k=v_k}(R) = \sigma_{\rho(\ell,a,b)}(R),$$

holds for any $R \subseteq \mathcal{D}$.

From Lemma 1, to transform a primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$ into an equivalent lex search, the index for the lex search can be any sequence of all attributes of R such that the first k attributes are $x_1, \ldots, x_k$ in an arbitrary order. Thus, *we also use $\ell = x_1 \prec \cdots \prec x_k$ which is only a subsequence of the attributes of R to denote an index*, since the chaining order of the remaining attributes is irrelevant for the lex search.

Example 2. *Consider the primitive searches in the second column of Table 1, their corresponding lex searches are illustrated in the third to fifth columns, where the third column shows the lower bound a, the forth column shows the upper bound b, and the fifth column shows the index ℓ. Here, given a primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$, the index is selected as $\ell = x_1 \prec \cdots \prec x_k$. Thus, each lex search uses a distinct index.*

Remarks. The lex searches $\sigma_{\rho(\ell,a,b)}(R)$ constructed from primitive searches $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$, as discussed in above, are in a special form. That is, for any attribute $x_i \in \{x_1, \ldots, x_k\}$ we have $a(x_i) = b(x_i) = v_i$, and for any attribute $x_i \in A_R \backslash \{x_1, \ldots, x_k\}$ we have $a(x_i) = \bot_i$ and $b(x_i) = \top_i$. Thus, the results of a lex search form a consecutive interval in the lexicographical order of all tuples of R with respect to ℓ. As a result, any one-dimensional order-based index (e.g., B-tree) can be used to implement ℓ, and a lex search can be executed in linear-log time in the size of the output in the worst case, i.e., $\mathcal{O}(|\sigma_{\rho(\ell,a,b)}(R)| \log n)$ where n is the number of tuples in the relation R. It is worth mentioning that for general range searches, one would need a multi-dimensional index (e.g., R-tree) to implement ℓ, which has a higher time complexity and runs slower than one-dimensional index such as B-tree. Thus, in this paper we only consider the special range searches, which we refer to as lex searches. Lex searches can be supported by one-dimensional indexes.

It is easy to construct an example where a particular primitive search cannot be transformed into a lex search using a particular index ℓ, and thus this search cannot be sped up by ℓ. For example, for $R(x, y) = \{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle\}$ and $\ell = x \prec y$, we have $\sigma_{y=1}(R) = \{\langle 1, 1\rangle, \langle 2, 1\rangle\}$ which is the first and third tuple in the lexicographical order ℓ. In view of this, we say that an index *covers* a primitive search if it can be used to speed up the primitive search by a lex search. We have the following corollary.

Corollary 1 (INDEX COVER)**.** *An index ℓ covers a primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$ for all $R \subseteq \mathcal{D}$* ***if and only if*** *the k-th prefix of ℓ is $\{x_1, \ldots, x_k\}$.*

As the lex search that is transformed from a primitive search is uniquely determined by the index and the primitive search, we focus our discussions on indexes rather than lex searches in the remainder of the paper.

4.2 Minimum Index Selection

Due to the lower look-up time complexity of lex searches compared with that of linear scan, indexes are essential for efficient `Datalog` program computations. However, when constructing indexes, the question remains: *what is the best set of indexes needed to cover all primitive searches* for a given relation. In this section we define the minimum index selection problem.

Before formally defining our problem, we first establish some additional notations. Firstly, we abstract a primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}$ as its set of search attributes, which we refer to as a *search* and is denoted by $S = \{x_1, \ldots, x_k\}$. This is because the constants $v_1, \ldots, v_k$ are irrelevant to index creation. Secondly, given a set $\mathcal{S}$ of searches and a set $\mathcal{L}$ of indexes on a relation R, we would like to know whether $\mathcal{L}$ can cover $\mathcal{S}$. Note that, since all primitive searches with the same set of attributes (i.e., the same search) can be covered by the same index, in the following when referring search set we use the set-based semantics. We formalize this via the **l-cover** predicate.

Definition 4 (L-COVER)**.** *Given a set $\mathcal{S}$ of searches and a set $\mathcal{L}$ of indexes on a relation R, we define a predicate $\textbf{l-cover}_{\mathcal{S}}(\mathcal{L})$ which is true if for every search $S \in \mathcal{S}$, there exists an index $\ell \in \mathcal{L}$ that covers S.*

Then, based on the definition of **l-cover**, we would like to find the smallest set of indexes that cover a search set $\mathcal{S}$. The rationalities of minimizing the number of indexes are as follows. Firstly, following Corollary 1, an index represented by an attribute sequence ℓ may cover a multitude of searches assuming the elements of its prefixes coincide with the attributes of the searches. For example, two searches $S_1 = \{x\}$ and $S_2 = \{x, y\}$ on a relation can be covered by the same index $\ell = x \prec y$. Secondly, for a search that can be covered by multiple indexes, the benefits of the different indexes are the same, i.e., they will result in the same running time. Thirdly, the fewer the indexes, the lower the creation and maintenance costs of these indexes.

As indexes and searches on different relations are independent, we consider each relation separately. We formulate our problem as follows.

Problem 2 (Minimum Index Selection Problem (MISP)). Given a set $\mathcal{S}$ of searches on a relation R, the minimum index selection problem is to find a set of indexes with the minimum cardinality such that all searches of $\mathcal{S}$ are covered by the index set, i.e.,

$$f_{\mathcal{S}} = \arg \min_{\mathcal{L}:\textbf{l-cover}_{\mathcal{S}}(\mathcal{L})} |\mathcal{L}|.$$

Example 3. *Continuing Example 2, the set of searches in Table 1 is $\mathcal{S} = \{\{x\}, \{x, y\}, \{x, z\}, \{x, y, z\}\}$. It can be covered by two indexes $\ell_1 = x \prec y \prec z$ and $\ell_2 = x \prec z$, which is shown in the sixth column of Table 1; this is smaller than the four indexes used in Example 2. Indeed, two is the smallest number of indexes to cover $\mathcal{S}$, since it is easy to see that $\{x, y\}$ and $\{x, z\}$ cannot be covered by the same index.*

5. COMPUTING THE OPTIMAL MISP

In this section, we propose an algorithm to solve MISP optimally in polynomial time. We begin with discussing the inviability of a brute-force approach.

5.1 Inviability of a Brute-force Approach

Before presenting our algorithm, we discuss the size of the search space of MISP. If it is very large, then a brute-force algorithm is not viable, especially for high performance engines.

Given a set $\mathcal{S}$ of searches on a relation R, let A be the set of attributes of R that are relevant for the searches, i.e., $A = \bigcup_{S \in \mathcal{S}} S$. We use $\mathcal{L}_A$ to represent the set of all possible permutation/sequences that may be formed by the elements of A, i.e., $\mathcal{L}_A = \bigcup_{X \subseteq A, X \neq \emptyset} \mathsf{Pm}(X)$. Here, $\mathsf{Pm}(X)$ denotes the set of *permutations* of a set X which is the set of all possible sequences formed by all elements of X such that each element occurs exactly once. Now, we bound $|\mathcal{L}_A|$. Although constructing a closed form is hard, it can be bounded by the following lemma.

Lemma 2. *Given a set A of m attributes (i.e., $A = \{x_1, \ldots, x_m\}$), the cardinality of the set $\mathcal{L}_A$ of all sequences of A is bounded by $m! \leq |\mathcal{L}_A| \leq \mathsf{e} \cdot m!$.*

Proof. The lower bound is given by $|\mathsf{Pm}(A)| = m!$, since $\mathsf{Pm}(A) \subseteq \mathcal{L}_A$. The upper bound is computed as follows, $|\mathcal{L}_A| = \left|\bigcup_{X \subseteq A, X \neq \emptyset} \mathsf{Pm}(X)\right| = \sum_{X \subseteq A, X \neq \emptyset} |X|! = \sum_{1 \leq i \leq m} \binom{m}{i} i! = m! \sum_{1 \leq i \leq m} \frac{1}{(m-i)!} = m! \sum_{0 \leq i \leq m-1} \frac{1}{i!} \leq m! \sum_{i \geq 0} \frac{1}{i!} = \mathsf{e} \cdot m!$ where the second equality follows from the fact that, for any $X \subseteq A$ and $Y \subseteq A$ with $X \neq Y$, we have $\mathsf{Pm}(X) \cap \mathsf{Pm}(Y) = \emptyset$. □

Note that, the absolute error of the over-approximation of $|\mathcal{L}_A|$ is small, i.e., $\mathsf{e} \cdot m! - |\mathcal{L}_A| = m! \sum_{i \geq m} \frac{1}{i!} = \sum_{i \geq 0} \frac{m!}{(i+m)!} \leq \sum_{i \geq 0} \frac{1}{i!} = \mathsf{e}$. The values of $|\mathcal{L}_A|$ and the relative error $\varepsilon = \frac{\mathsf{e} \cdot m! - |\mathcal{L}_A|}{|\mathcal{L}_A|}$ of its over-approximation, for m varying between 1 and 9, is given in the table below:

m	$\lvert\mathcal{L}_A\rvert$	$\varepsilon \cdot 100$
1	1	171.828
2	4	35.914
3	15	8.731
4	64	1.936
5	325	0.367
6	1956	0.059
7	13699	0.008
8	109600	0.001
9	986409	≈ 0.000

Recall that, MISP searches for the smallest subset of $\mathcal{L}_A$ that covers all primitive searches on a relation. Thus, a brute-force approach would require to iterate through all subsets of $\mathcal{L}_A$. Then, the search space of a brute-force approach is $2^{\mathcal{L}_A} = \{\mathcal{L} \mid \mathcal{L} \subseteq \mathcal{L}_A\}$, and its size is $|2^{\mathcal{L}_A}| = 2^{|\mathcal{L}_A|}$. Using the approximation of $|\mathcal{L}_A|$ in Lemma 2, we obtain a complexity of $\mathcal{O}(2^{\mathsf{e} \cdot m!})$.

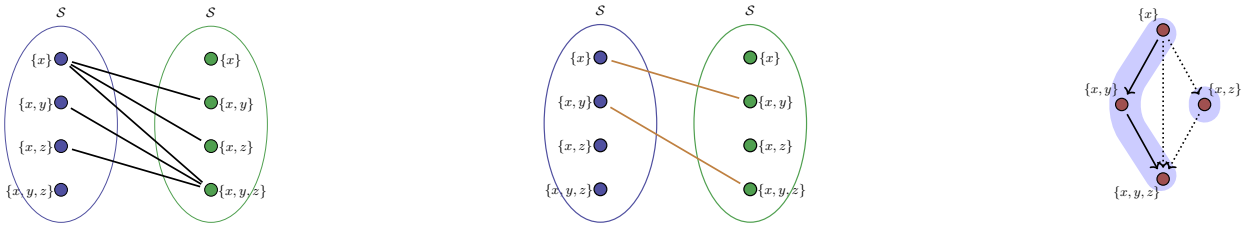
Theorem 1. *A brute-force approach for MISP exhibits a worst-case time complexity of $\mathcal{O}(2^{m^m})$.*

Proof. As discussed above, the time complexity of a brute-force approach for MISP is $\mathcal{O}(2^{\mathsf{e} \cdot m!})$. Then, this theorem follows from Sterling's approximation of $m!$. Note that, the approximation becomes more precise for a large m. □

As a result, a brute-force approach becomes intractable very quickly. For example, for a relation with 4 attributes, a brute-force MISP algorithm has to test $2^{64} \approx 1.8 \times 10^{19}$ different subsets of $\mathcal{L}_A$ for coverage and minimality.

5.2 Computing MISP via Chain Cover

In view of the inviability of a brute-force approach, we propose to solve MISP via computing a chain cover of the searches. In the following, we first formulate the minimum chain cover problem (MCCP) and prove that an optimal MISP solution can be obtained from an optimal MCCP solution. Then, we propose a polynomial-time algorithm `MinIndex` that solves MISP optimally.



(a) Bipartite graph **(b)** Maximum matching **(c)** Chain cover

Figure 3: Running example of computing MCCP for searches $\{x\}$, $\{x, y\}$, $\{x, z\}$, and $\{x, y, z\}$.

5.2.1 Minimum Chain Cover Problem

We define a search chain C as a set of searches $\{S_1, \ldots, S_k\}$ that subsume each other and form a total order, i.e., $C \equiv S_1 \subset S_2 \subset \cdots \subset S_k$. A search chain is related to an index as follows.

Lemma 3. *Given a search chain $C = S_1 \subset S_2 \subset \cdots \subset S_k$, we can construct an index to cover all searches of C.*

Proof. We prove this lemma by constructing such an index that covers all searches of C. Let $S_i - S_{i-1}$ denote the set of attributes of S_i that are not in S_{i-1}. Then, it is easy to see that any index conforming with $S_1 \prec (S_2 - S_1) \prec \cdots \prec (S_k - S_{k-1})$ is such an index, i.e., attributes of $S_{i+1} - S_i$ appear later than attributes of $S_i - S_{i-1}$. Note that, the attributes of S_1 and the attributes of $S_i - S_{i-1}$ can be ordered arbitrarily, respectively, within their sets of attributes. □

Following Lemma 3, we say that a search chain C covers all its searches, i.e., C covers S for every search $S \in C$. Then, we would like to know whether a set $\mathcal{C}$ of search chains can cover all searches in a search set $\mathcal{S}$. We formalize this via the **c-cover** predicate.

Definition 5 (C-COVER)**.** *Given a set $\mathcal{S}$ of searches and a set $\mathcal{C}$ of search chains on a relation R, we define a predicate **c-cover**$_{\mathcal{S}}(\mathcal{C})$ which is true if for every search $S \in \mathcal{S}$, there is a search chain $C \in \mathcal{C}$ that covers S, i.e.,*

$$\textbf{\textit{c-cover}}_{\mathcal{S}}(\mathcal{C}) = \forall S \in \mathcal{S} : \exists C \in \mathcal{C} : S \in C.$$

Now, we are ready to define our minimum chain cover problem, which aims to find the smallest set of search chains to cover all searches in a given set of searches.

Problem 3 (Minimum Chain Cover Problem (MCCP)). Given a set $\mathcal{S}$ of searches on a relation R, the minimum chain cover problem is to find the minimum set $g_{\mathcal{S}}$ of search chains to cover $\mathcal{S}$, i.e.,

$$g_{\mathcal{S}} = \arg \min_{\mathcal{C}:\textbf{c-cover}_{\mathcal{S}}(\mathcal{C})} |\mathcal{C}|$$

The rational of defining MCCP is that given a set $\mathcal{C}$ of search chains covering all searches in a search set $\mathcal{S}$, we can construct a set of indexes of cardinality $|\mathcal{C}|$ to cover $\mathcal{S}$ by following Lemma 3. Thus, the smaller the cardinality of $\mathcal{C}$, the better.

Moreover, there is a one-to-one correspondence between solutions of MISP and solutions of MCCP, as proved by the following lemma.

Lemma 4. *Given any search set $\mathcal{S}$ on a relation R, there is a one-to-one correspondence between search chains $\mathcal{C}$ that cover $\mathcal{S}$ and indexes $\mathcal{L}$ that cover $\mathcal{S}$, such that $|\mathcal{C}| = |\mathcal{L}|$.*

Proof. Following from Lemma 3, we know that given any set $\mathcal{C}$ of search chains that cover $\mathcal{S}$, we can construct an index set of cardinality $|\mathcal{C}|$ to cover $\mathcal{S}$. Thus, what remains to be proved in this lemma is that given any index ℓ, we can construct a search chain C to cover all searches that are covered by ℓ.

Given an index ℓ and a set $\mathcal{S}$ of searches, we let $\mathcal{S}_\ell$ denote the subset of $\mathcal{S}$ that are covered by ℓ. We will show that $\mathcal{S}_\ell$ is a search chain. Firstly, it is easy to see that for any $S, S' \in \mathcal{S}_\ell$, we have $|S| \neq |S'|$. Secondly, following Corollary 1, we know that for any $S, S' \in \mathcal{S}_\ell$, we have either $S \subset S'$ or $S' \subset S$, since the k-th prefix of ℓ is a subset of a $(k+1)$-th prefix of ℓ. Thus, the lemma holds. □

Following from Lemma 4, we have the following corollary, which states that we can obtain an optimal MISP solution from an optimal MCCP solution.

Corollary 2. *Given any search set $\mathcal{S}$ on a relation R, an optimal MISP solution can be obtained from an optimal MCCP solution.*

5.2.2 A Polynomial-time MISP Algorithm

We have shown in Corollary 2 that we can obtain an optimal MISP solution from an optimal MCCP solution. The good news is that MCCP can be solved optimally in polynomial time by the Dilworth's Theorem [15], which states that in a finite partial order, the size of a maximum anti-chain is equal to the minimum number of chains needed to cover its elements. An anti-chain is a subset of a partially ordered set such that any two elements in the subset are unrelated, and a chain is a totally ordered subset of a partial ordered set. Although Dilworth's Theorem is non-constructive, there exists constructive versions that solve the minimum chain cover problem either via the maximum matching problem in a bipartite graph [16] or via a max-flow problem [27]. Both problems are optimally solvable in polynomial time.

The general idea of computing a minimum chain cover for a search set $\mathcal{S}$ is as follows. Firstly, a bipartite graph $G_{\mathcal{S}} = (U, V, E)$ is constructed such that there is a vertex in both U and V for each search $S \in \mathcal{S}$, and there is an edge between $S \in U$ and $S' \in V$ if S is a proper subset of S' (i.e., $S \subset S'$). For example, Figure 3a illustrates the bipartite graph constructed for the search set $\mathcal{S} = \{\{x\}, \{x, y\}, \{x, z\}, \{x, y, z\}\}$ in Table 1, where the edge set is given by the strict subset relationship between a search pair, i.e., $(\{x\}, \{x, y\})$, $(\{x\}, \{x, z\})$, $(\{x\}, \{x, y, z\})$, $(\{x, y\}, \{x, y, z\})$ and $(\{x, z\}, \{x, y, z\})$.

Secondly, a minimum chain cover is obtained from a maximum matching $\mathcal{M}$ of $G_{\mathcal{S}}$. A subset $\mathcal{M}$ of $G_{\mathcal{S}}$'s edges forms a *matching* if each vertex of U and V appears at most once in $\mathcal{M}$, and it is a *maximum matching* if it has the largest cardinality among all matchings of $G_{\mathcal{S}}$. Note that, a vertex u of U is considered to be different from a vertex v of V, even when u and v refer to the same search. For example, for the bipartite graph $G_{\mathcal{S}}$ in Figure 3a, $\{(\{x\}, \{x, y\}), (\{x, y\}, \{x, y, z\})\}$ is a maximum matching as shown in Figure 3b, while $\{(\{x\}, \{x, y\}), (\{x\}, \{x, z\})\}$ is not a matching since $\{x\}$ of U appears twice. Given a matching $\mathcal{M}$ of $G_{\mathcal{S}}$, a set of $|\mathcal{S}| - |\mathcal{M}|$ search chains that cover all searches

of $\mathcal{S}$ is constructed as follows: initially each search of $\mathcal{S}$ forms a singleton search chain of its own, then each edge of $\mathcal{M}$ joins two search chains and thus reduces the total number of search chains by one. For example, for the maximum matching in Figure 3b, the two search chains are obtained as $\{x\} \subset \{x, y\} \subset \{x, y, z\}$ and $\{x, z\}$ which are pictorially shown in Figure 3c.

Alternatively, we can view the edges of the bipartite graph $G_{\mathcal{S}}$ as directed edges in a unipartite graph with $\mathcal{S}$ as the set of vertices, e.g., see Figure 3c with both solid and dotted lines. Then, the edges of a matching $\mathcal{M}$ of $G_{\mathcal{S}}$ form $|\mathcal{S}| - |\mathcal{M}|$ directed paths in the unipartitie graph (each corresponding to one search chain), since each vertex has at most one in-coming edge and at most one out-going edge due to the definition of matching. Specifically, each chain starts from a search that do not have any predecessors (i.e., in-coming edges) in the matching $\mathcal{M}$. Moreover, the set of search chains constructed from a maximum matching of $G_{\mathcal{S}}$ has the smallest cardinality. This is because, given any set $\mathcal{C}$ of non-overlapping search chains of $\mathcal{S}$, a matching of $|\mathcal{S}| - |\mathcal{C}|$ edges can be constructed for $G_{\mathcal{S}}$, since each search chain $C \in \mathcal{C}$ adds $|C| - 1$ edges to the matching; two search chains are *non-overlap* if their sets of searches are non-overlap. Note that, for any search set $\mathcal{S}$, there is a minimum chain cover whose search chains are non-overlap, since a search chain remains valid after removing any search from it. As a result, a minimum set of search chains is constructed from a maximum matching of $G_{\mathcal{S}}$, where the pseudocode of the computation is shown in Algorithm 1. Finally, given the

Algorithm 1: MinChainCover($\mathcal{S}$)

Input: A set $\mathcal{S}$ of searches
Output: A minimum chain cover $\mathcal{C}$ of $\mathcal{S}$

```
1 M ← MaximumMatching(S, S, {(S, S') ∈ S × S | S ⊂ S'});
2 Initialize C to be the empty set;
3 for all u_1 ∈ S s.t. ∄(u_0, u_1) ∈ M do
4     Find maximal path (u_1, u_2), (u_2, u_3), ..., (u_{k-1}, u_k) ⊆ M ;
5     Add u_1 ⊂ u_2 ⊂ u_3 ⊂ ··· ⊂ u_{k-1} ⊂ u_k to C ;
6 return C
```

set $\mathcal{C}$ of search chains that is computed by Algorithm 1 for covering the search set $\mathcal{S}$, a set $\mathcal{L}$ of $|\mathcal{C}|$ indexes can be constructed to cover $\mathcal{S}$ by following the proof of Lemma 3. For example, the search chain $\{x\} \subset \{x, y\} \subset \{x, y, z\}$ is converted to the index $\{x\} \prec \{x, y\} - \{x\} \prec \{x, y, z\} - \{x, y\}$ which is $x \prec y \prec z$, and the search chain $\{x, z\}$ can be converted to either index $x \prec z$ or index $z \prec x$. The pseudocode for such a conversion is shown in Algorithm 2, and denoted by MinIndex.

Algorithm 2: MinIndex($\mathcal{S}$)

Input: A set $\mathcal{S}$ of searches
Output: A minimum set $\mathcal{L}$ of indexes to cover $\mathcal{S}$

```
1 C ← MinChainCover(S);
2 Initialize L to be the empty set;
3 for all S_1 ⊂ S_2 ⊂ ··· ⊂ S_{k-1} ⊂ S_k ∈ C do
4     Add to L an arbitrary index conforming with
        S_1 ≺ S_2 − S_1 ≺ ··· ≺ S_k − S_{k-1};
5 return L
```

The correctness of MinIndex (Algorithm 1 and Algorithm 2) follows from the above discussions. Let m be the number of distinct attributes in $\mathcal{S}$; note that, m is at most the number of attributes in a relation. Then, the time complexity of MinIndex is bounded by the following theorem.

Theorem 2. *The time complexity of* MinIndex *(Algorithm 1 and Algorithm 2) is* $\mathcal{O}(|\mathcal{S}|^{2.5} + |\mathcal{S}|^2 \cdot m)$.

Proof. The time complexity follows from the facts that, constructing the bipartite graph G takes $\mathcal{O}(|\mathcal{S}|^2 \cdot m)$ time, computing the maximum matching in G takes $\mathcal{O}(|\mathcal{S}|^{2.5})$ time, and both constructing chain cover from matching $\mathcal{M}$ and constructing indexes from chain cover take $\mathcal{O}(|\mathcal{S}| \cdot m)$ time. □

Note that, as both $|\mathcal{S}|$ and m are not large in practice (e.g., they are at most hundreds), the running time of MinIndex usually is negligible compared with the total running time of a `Datalog` program.

6. INTEGRATING INTO SOUFFLÉ

We have implemented our index selection approach as the default indexing technique of the open-source `Datalog` engine SOUFFLÉ, which works as follows. It first translates a given `Datalog` ruleset (also called a program) into C++ code during the *code generation phase*, then it compiles the C++ code into binary executable code at the *code compilation phase*, and finally the *code execution phase* executes the binary code on the EDB (i.e., input facts) to compute the IDBs. For more details of SOUFFLÉ, please refer to [21, 33].

Index selection occurs in the code generation phase, which also performs several rewrite transformations. In the first step, a *query translator* converts each rule of an input `Datalog` program to a nested loop join. It selects the best loop order, minimizing the iteration space of the nested loop join with the aid of a query planner [1] or user hints. In the second step, primitive searches (see Definition 1) are identified from the nested loop joins. In the last step, indexes are selected by our algorithm MinIndex to cover the primitive searches, and the primitive searches are replaced by index operations on relations based on the selected indexes.

The code execution phase of SOUFFLÉ is also divided into several steps. It ingests the whole factset (i.e., EDB) into main memory and stores them in EBD index structures. The binary code runs on in-memory structures, and repeatedly adds rows to the various IDB index structures which are initially empty. Finally, the computed IDB relations are output to disk. Note that, the first two steps are interleaved.

7. EXPERIMENTS

In this section, we evaluate our auto-indexing scheme by measuring an implementation of it, and also some alternative schemes, in a production-strength `Datalog` engine SOUFFLÉ [21]. The outcome of our evaluations is to validate the following claims.

Claim-I: Negligible Index Selection Overhead During Analysis. The time taken for selecting the indexes using our auto-indexing scheme does not substantially slow down the code generation and compilation phases compared to alternative indexing schemes.

Claim-II: Significant Performance Benefit During Execution. Our auto-indexing scheme provides a good combination of fast runtime evaluation and low memory footprint.

Claim-III: Good Enough, Without Hand Optimizations. Our auto-indexing scheme delivers runtime evaluation speed and memory usage that compare well with what users have accepted as worth the effort of hand optimizations.

7.1 Experimental Setup

Our experiments were performed on an Intel(R) Core(TM) i7-7700K CPU at 4.20GHz with 64GB of physical RAM running Ubuntu 16.04.3 LTS on the bare-metal. The experiments were conducted in isolation without virtualization so that runtime results are

Table 2: Cloud security ruleset sizes

Program	#Rules	#Relations
sec1	250	325
sec2	254	329
sec3	245	320

Table 3: Network factset sizes

Dataset	#Facts	Dataset	#Facts
N1075	3,515	N3511	4,290
N2340	3,503	N9087	4,343
N3500	4,340		

Table 4: DaCapo factset sizes

Dataset	# Facts	Dataset	#Facts
lu-index	4,396,394	antlr	8,319,095
lu-search	4,396,394	jython	5,203,400
bloat	4,468,277	pmd	8,388,217
eclipse	4,389,763	fop	8,769,560

	Auto			None		Single		Maximal	
	MinIndex	gen	compile	gen	compile	gen	compile	gen	compile
1o1h	0.0015	0.99	86.86	1	79.64	0.99	76.94	0.99	89.11
2o2h	0.0015	1.02	87.02	1	79.3	1	76.78	0.99	89.3
3o3h	0.0015	1.01	87.13	0.99	79.88	0.99	77.59	1.01	89.4
sec1	0.0008	0.09	106.9	0.1	94.84	0.1	93.11	0.1	114.23
sec2	0.0008	0.11	113.11	0.11	95.56	0.1	92.31	0.11	120.01
sec3	0.0008	0.09	107	0.1	93.41	0.09	91.71	0.1	109

Figure 4: Code generation time (gen) and compilation time (compile) in seconds

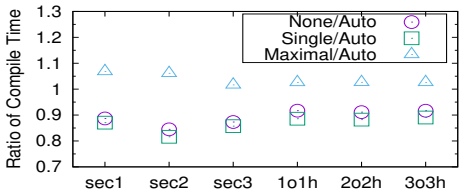


Figure 5: Ratio of compilation time w.r.t. Auto

robust. All experiments shown here are run in single-thread mode for SOUFFLÉ V1.3.1-175-g2977f469 and G++ V7.3.0.

7.1.1 Compared Indexing Schemes

We compare the following three indexing schemes, all implemented by us in SOUFFLÉ.

- Auto: our auto-indexing scheme presented in Algorithm 2.
- Maximal: one index for each distinct search on a relation.
- Single: only one index for each relation. To choose the best index for a relation R for a given workload, we first count the frequency of each individual search S on R which is obtained by instrumenting the search pattern while executing the Datalog program once. Then, the best single index is selected as the one whose set of covered searches has the maximum total frequency. This can be computed in quadratic time to the number of searches by dynamic programming (cf. Chapter 12, [13]); we omit the details.

Intuitively, these two alternative indexing schemes, Maximal and Single, should be especially good for the execution speed and the memory efficiency, respectively. However Maximal uses much more memory for the numerous indexes, and Single doesn't cover every search and thus could be very slow in evaluating the program. Our experiments in Section 7.2.2 validate these expectations, and show that Auto offers an excellent compromise, with runtime similar to Maximal and much less than Single, and using memory similar to Single and substantially less than Maximal.

To aid in understanding the implications and overheads of the indexing in SOUFFLÉ, we also include some measurements for two radically different approaches. In the code analysis steps, we consider an scheme we call None, in which there is no work done to choose indices based on the searches to be performed; instead the system stores each relation with the single index that is determined by the lexicographic order of the attributes in the relation. This establishes a baseline for seeing the overhead of the work done in any approach that examines the set of searches in order to create suitable indices. For the execution phase, we have implemented what we call Hash, where the relations (both EDB and IDB) are stored using the STL hash_map. As a hash map cannot be shared between two different searches, Hash builds one hash map for each distinct search in the same way as Maximal. We discuss the implications of this below.

In addition, for the workloads of one use case from program analysis, we also compare our auto-indexing scheme in SOUFFLÉ to another Datalog system PA-Datalog, an optimized Logicblox Ver.3 for program analysis. The ruleset used with PA-Datalog has been heavily hand-optimized through months of work by experts, specially for the use case. Because these are different engines, the comparison of speed and memory is not truly apples-to-apples. Nevertheless, we will illustrate in Section 7.2.3 that our auto-indexing scheme in SOUFFLÉ results in better performance without human optimizing effort, compared to what users have accepted as sufficiently good to justify the effort of hand-optimization.

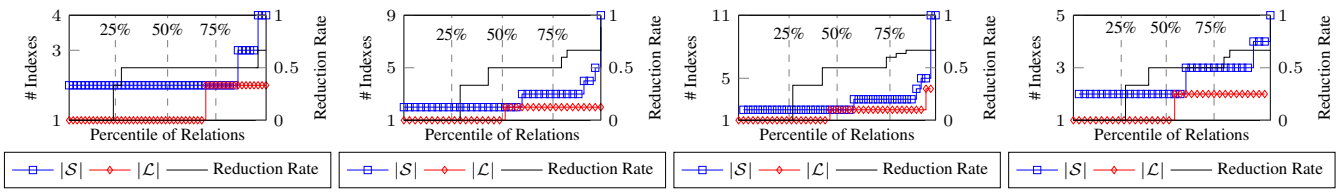
7.1.2 Case Studies

We perform our evaluations using two real-world case studies: namely, a cloud security use case and a program analysis use case. These use cases are of very large scale, where the Datalog programs contain hundreds of rules and relations and produce gigatuple output relations.

Use Case-I: Cloud Security Analysis. The first use case is to analyze the security of Amazon networks. In this industrial use case, a Domain Specific Language (DSL) is used to describe security properties of networks and to query about the networks. A translator automatically converts the security specifications and queries in a DSL to a Datalog program where the properties of the given networks are encoded as EDBs (i.e., input relations). The generated Datalog programs are unoptimized, since the DSL doesn't offer annotations for hand-crafted optimizations such as enforcing good indexing schemes. It is worth pointing out that *this use case has resource constraints including a low memory footprint and runtime limitations, as imposed by running the security analysis as a service in Amazon Lambda* [35].

For this use case, we consider three security analysis workloads (i.e., three Datalog programs), each encoding specific security properties and security queries. We name these three programs as sec1, sec2, and sec3, where the numbers of rules and relations of these programs are shown in Table 2. At execution time, the programs run on five network factsets that vary in complexity: networks N1075 and N2340 have less complexity whereas networks N3500, N3511, and N9087 are more complex in terms of their network connectivity. The EDB sizes (i.e., total number of tuples in all input relations) of the five network datasets are summarized in Table 3.

Use Case-II: Program Analysis. The second use case is DOOP program analysis that performs points-to analyses for Java programs; DOOP is publicly available and open source [37]. Specifically, a Java program is encoded as an EDB (i.e. input relations) and the points-to analysis is expressed as a Datalog program. DOOP's points-to analysis has been used to analyze very large libraries such as the Oracle JDK [21]; as a result, *it requires very fast execution and low memory footprints* in order to be solved in a feasible time and with feasible resources. The DOOP analysis workloads have different parameterizable precisions, which



(a) Index distribution of DOOP **(b)** Index distribution of `sec1` **(c)** Index distribution of `sec2` **(d)** Index distribution of `sec3`

Figure 6: Index distribution for all rulesets

depend on (1) how concrete Java objects are abstracted to a finite set of objects in a sound fashion and (2) how much context is stored for each variable. For example, a context could be a trace over last few call-sites or receiver object of a method call. In our testing, we use three representative precision settings, 1-object-sensitive+1-heap (`1o1h`), 2-object-sensitive+2-heap (`2o2h`), and 3-object-sensitive+3-heap (`3o3h`). Each of these precision settings corresponds to a `Datalog` program containing 496 relations and 469 rules. However, increased precision leads to larger relations due to added rule complexity. Each analysis program is applied at execution time to 8 factsets from the DaCapo06 benchmark suite [4], where the sizes of these factsets are summarized in Table 4.

7.2 Experimental Results

We present experimental results to validate our claims in the following three subsubsections.

7.2.1 *Code Analysis Performance*

Index Selection Overhead. In order to quantify the overhead of index selection in our Auto indexing scheme, we also implemented an indexing scheme (None) which trivially builds the index on a relation's attributes in order as they appear. The code generation time (gen) and the code compilation time (compile) for all the four indexing schemes, Auto, None, Single, and Maximal, for both use cases are shown in Figure 4. We observe that the code generation time for the four indexing schemes are almost the same. Recall that index selection occurs during code generation. Thus, the different index selection methods have little impact on the code generation time. This is because index selection takes a negligible portion of the code generation time, e.g., less than 1% for Auto; the time of index selection by MinIndex is shown in the second column of the table in Figure 4. Note that, here we did not include in the measurement for Single the extra preliminary activities that collect statistics information such as frequencies of searches.

On the other hand, different index choices may lead to different work in the code compilation phase too, and the more indexes whose construction needs to be compiled, the longer the compilation time. The main reason is that each index requires additional templatized comparator functions that the C++ compiler needs to unroll at template instantiation time. Thus, in this phase, None and Single are slightly faster than Auto and Maximal, as shown in Figure 4 and Figure 5. Nevertheless, the differences are not significant.

Overall, the time for code generation which also conducts index selection is negligible compared with the code compilation time, and **our** Auto **indexing scheme does not substantially slow down the code generation and compilation time**. It is also worth mentioning that the binary code is independent of the dataset and, once generated, it can be run on any input dataset (i.e., factset).

Distribution of Index Reduction. We analyze the number of indexes constructed for the various `Datalog` programs. Recall that, given a set of searches we compute the smallest set of indexes $\mathcal{L}$ to cover/speed up all searches of $\mathcal{S}$, while the Maximal indexing scheme constructs one index for each search in $\mathcal{S}$, and Single constructs one index for each relation. Thus, the reduction ratio for the number of indexes of Auto over Maximal will be upper bounded by $|\mathcal{S}|$ for a relation (which is the reduction ratio for Single over Maximal). The distributions of $|\mathcal{S}|$ among all relations that have at least two searches for the three cloud security analyses, `sec1`, `sec2`, and `sec3`, are shown as blue squares measured against the left-hand scale, in Figures 6b, 6c, and 6d, respectively. We can see that more than 50 percent of the relations have only two searches, and more than 80 percent of the relations have at most three searches; this means that for 80 of the relations, $|\mathcal{L}|/|\mathcal{S}|$ is at least $1/3$. In order to quantify the reduction ratio of Auto over Maximal, we define it as $1 - |\mathcal{L}|/|\mathcal{S}|$. The distributions of the reduction ratio are shown as black line in Figures 6b, 6c, and 6d, measured against the right-hand scale. We can see that, for 25 percent of the relations, there is no reduction (i.e., $|\mathcal{S}| = |\mathcal{L}|$), for another 25 percent of the relations, the reduction is around 50%, and for the remaining 50 percent of relations, the reduction is between 50% and 70%. Finally, the distributions of the actual number of indexes $|\mathcal{L}|$ constructed for the relations are shown as red diamond in Figures 6b, 6c, and 6d measured against the left-hand scale. For cloud security analyses `sec1` and `sec3`, the largest number of indexes constructed for a relation is only two, while the largest number searches on a relation is 9. For cloud security analysis `sec2`, the largest number of indexes constructed for a relation is three, while the largest number searches on a relation is 11. As shown in Figure 6a, similar results are also observed for the DOOP program analysis.

7.2.2 *Evaluation-time Performance*

To justify our choice of adopting a linear order-based index (i.e., B-tree index), we also implemented an indexing scheme that uses the hash technique, denoted by Hash. Specifically, Hash uses C++'s hash_map to index relations. As a hash map cannot be shared between two different searches, Hash builds one hash map for each distinct search in the same way as Maximal.

Running Time. The time of the code execution phase for the four indexing schemes, Auto, Maximal, Hash and Single, running on the three cloud security analyses on networks is shown in Figure 7. The time is divided into loading time that loads factset/EDB from disk to main memory and builds indexes for the EDB, and executing time that repeatedly computes and adds rows to the IDB index structures. As there are some searches that can't exploit an index in Single, this takes an excessively long time (i.e., more than 24 hours, denoted by TLE) when given one of the three large networks: N3500, N3511, and N9087. Thus, Single is not suitable to process large-scale `Datalog` programs. The running time of Auto is almost the same as that of Maximal, and in fact Auto is often slightly faster. This is because execution involves both constructing the indexes for IDB as the facts are computed on the fly, as well

		N1075		N2340		N3500		N3511		N9087	
		load	exec	load	exec	load	exec	load	exec	load	exec
sec1	Auto	0.001	123	0.005	155	0.003	4788	0.003	4643	0.003	4676
	Maximal	0.003	194	0.005	156	0.003	5000	0.003	4940	0.003	4832
	Hash	0.1	194	0.35	126	1.12	11962	1.64	11096	2	9455
	Single	0.002	357	0.004	181	-	TLE	-	TLE	-	TLE
sec2	Auto	0.003	147	0.005	156	0.003	4416	0.003	4476	0.003	4494
	Maximal	0.003	232	0.005	158	0.003	4662	0.003	4660	0.003	4449
	Hash	0.1	442	0.34	122	1.57	59230	-	OOM	1.56	10495
	Single	0.002	367	0.005	177	-	TLE	-	TLE	-	TLE
sec3	Auto	0.003	132	0.005	157	0.004	4439	0.003	4602	0.003	4610
	Maximal	0.003	184	0.005	156	0.003	5076	0.003	5106	0.003	5043
	Hash	0.1	4819	0.28	121	1.49	86410	-	TLE	-	TLE
	Single	0.002	381	0.005	188	-	TLE	-	TLE	-	TLE

Figure 7: Running time in seconds for cloud security analysis (TLE: time limit exceeded)

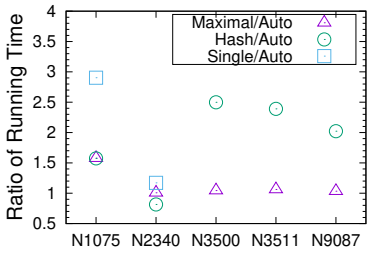


Figure 8: Ratio of running time w.r.t. Auto for `sec1`

		N1075		N2340		N3500		N3511		N9087	
		load	exec	load	exec	load	exec	load	exec	load	exec
sec1	Auto	0.002	0.35	0.002	0.35	0.002	4.598	0.002	4.598	0.002	3.742
	Maximal	0.002	0.43	0.002	0.44	0.002	28.527	0.002	28.527	0.002	22.515
	Hash	0.002	1.1	0.003	1.187	0.002	44.333	0.002	44.337	0.002	36.381
	Single	0.002	0.35	0.002	0.35	-	TLE	-	TLE	-	TLE
sec2	Auto	0.003	1.4	0.003	0.35	0.002	14.969	0.002	14.970	0.002	11.910
	Maximal	0.002	3.7	0.002	0.44	0.002	37.443	0.002	37.443	0.002	29.778
	Hash	0.002	6.9	0.003	1.199	0.002	58.370	-	OOM	0.002	48.228
	Single	0.002	0.7	0.002	0.35	-	TLE	-	TLE	-	TLE
sec3	Auto	0.002	0.9	0.003	0.35	0.002	1.710	0.002	1.710	0.002	1.693
	Maximal	0.002	1.6	0.002	0.44	0.002	3.054	0.002	3.054	0.002	3.031
	Hash	0.002	3.3	0.004	1.199	0.002	6.792	-	TLE	-	TLE
	Single	0.002	0.4	0.002	0.35	-	TLE	-	TLE	-	TLE

Figure 9: Memory usage for cloud security analysis (GB)

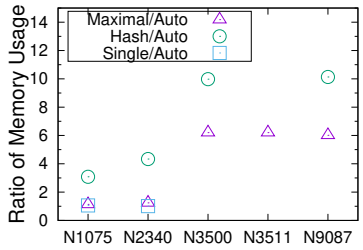


Figure 10: Ratio of memory usage w.r.t. Auto for `sec1`

as doing the primitive searches. The latter aspect should in principle be the same for Auto and Maximal, but Maximal constructs more indexes than Auto. Hash is often much slower than Auto and Maximal, due to the inefficient data structure of hash map in practice. The ratio of the running time of Maximal, Hash, and Single with respect to Auto for `sec1` is shown in Figure 8.

The running time of the execution phase for Auto, Maximal, Hash and Single on the three DOOP program analyses are illustrated in Figure 11. The general trend is similar to that for cloud security analysis. Although Single can complete all the DOOP program analysis, it takes significantly more time than Auto and Maximal.

Overall, we find that Auto **runs even a bit faster than** Maximal**, and it is significantly faster than** Single **and** Hash**.** This validates our motivation to construct enough indexes so that every search is sped up, and to use linear order-based index structures.

Memory Usage. We evaluate the memory usage of Auto compared to Single, Maximal, and Hash. We define the memory usage improvement of an indexing scheme A over another scheme B as the ratio of memory usage or B compared to that of A.

The memory usages of Auto, Single, Maximal, and Hash for cloud security analyses, `sec1`, `sec2`, and `sec3`, are shown in Figure 9. We see that Single always consumes the smallest amount of memory, and Hash always consumes the largest amount of memory. The memory usage improvement of Auto over Maximal can be up-to 6, e.g., see Figure 10. The memory usage penalty for Auto compared to Single is at most two times. Figure 12 shows the memory usage of Auto, Single, Maximal and Hash for DOOP program analysis, where the memory usage improvement of Auto over Maximal is around 2, and Auto consumes only around 20% more memory than Single.

Overall, the memory usage of Auto **is not far from that of** Single**, and better than** Maximal **and** Hash**.**

7.2.3 *Against* `PA-Datalog`

We also measured the heavily hand-optimized `PA-Datalog` system for the DOOP program analysis, on the same hardware. The results are shown in Figure 11 and Figure 12. We see that Auto is faster and consumes less memory than `PA-Datalog`. The running time improvement ranges from 3–5x, and the memory usage improvement ranges from 2–5x. This demonstrates that **our** Auto **indexing scheme works well enough**, automatically getting performance that previously required extensive hand optimization.

7.2.4 *Hashing*

As mentioned, the SOUFFLÉ engine uses a linear B-tree index structure for each relation. In light of the value of hash techniques in SQL database engines [5], we explain why those ideas are not used in SOUFFLÉ. Note first that the traditional hash-join, where each input is partitioned by a hash on the join attribute, and then the output is produced in hash buckets, is not suitable for the multi-way joins involved in many `Datalog` rules, as we can't afford the space for materializing the whole intermediate relation (which will be the input to the next stage of the join, and thus needs to be hashed itself). So we are left with doing a nested loop join, where each relation is stored as a hash-index. This is what we measured above, as Hash, and found it was not competitive. We reported experiments with the STL hash_map, but we also explored other hash implementations such as Google sparse hashmap implementation, but none were notably better in memory consumption and runtime at the same time. For example, Google's sparse hash-set implementation has the issue that it would not permit multi-sets which are essential for storing different tuples with the same key in the index. Independent of this issue, micro-benchmarks for Google's sparse hashmap indicate that the memory consumption reduces at most by a factor of two. However, the runtime of the query execution would increase substantially. Their dense version of the set would have the reverse effect. Hence, the STL's unordered_multiset is a good compromise in terms of runtime and memory consumption. It is also the case that hash implementations typically don't paral-

		lu-index		lu-search		bloat		eclipse		antlr		jython		pmd		fop	
		load	exec	load	exec	load	exec	load	exec	load	exec	load	exec	load	exec	load	exec
1o1h	Auto	2.3	17.3	2.4	21.1	2.4	14.8	2.3	19.6	4.4	35.1	2.8	23.9	4.7	36.2	4.7	36.6
	Maximal	2.4	20.4	2.4	22.6	2.4	20.7	2.3	20.7	4.5	35	2.9	25.2	4.7	37.6	4.5	36.6
	Hash	2.8	38.4	2.8	27.2	2.9	35	2.7	36	5.3	63.3	3.4	51.5	5.5	78.3	5.8	82.7
	Single	2.3	823	2.2	811	2.1	834	2.3	810	4.4	1352	2.8	926	4.4	1386	4.5	1399
	PA-Datalog	-	46	-	46	-	45	-	48	-	91	-	115	-	112	-	112
2o2h	Auto	2.3	119	2.3	119	2.2	117	2.4	119	4.3	132	3	122	4.4	133	4.9	135
	Maximal	2.3	138	2.3	138	2.3	136	2.3	140	4.4	150	2.8	141	4.4	153	4.6	154
	Hash	2.6	183	2.6	181	2.6	182	3	217	5.1	201	3.3	188	5.1	203	5.2	210
	Single	2.3	3370	2	3376	2.0	3288	2.3	3326	4.4	3508	2.8	4869	4.4	5026	4.3	5112
	PA-Datalog	-	245	-	255	-	256	-	454	-	441	-	388	-	291	-	461
3o3h	Auto	2.3	445	2.2	446	2.3	443	2.3	442	4.2	454	2.8	448	4.3	463	4.5	458
	Maximal	2.3	508	2.2	506	2.2	505	2.3	506	4.4	519	2.7	509	4.4	522	4.5	518
	Hash	2.4	1826	2.6	1657	2.6	1798	2.6	1903	5.1	1757	3.2	1675	5.1	1691	5.3	1736
	Single	2.3	24441	2	25238	2	26734	2.3	23211	4.3	28030	2.7	37889	4.4	41476	4.5	44074
	PA-Datalog	-	1259	-	1301	-	1285	-	1477	-	1273	-	1464	-	1379	-	1415

Figure 11: Running time for DOOP program analysis (seconds)

		lu-index		lu-search		bloat		eclipse		antlr		jython		pmd		fop	
		load	exec	load	exec	load	exec	load	exec	load	exec	load	exec	load	exec	load	exec
1o1h	Auto	0.37	0.29	0.37	0.28	0.38	0.27	0.38	0.28	0.71	0.36	0.45	0.34	0.72	0.37	0.75	0.38
	Maximal	0.37	0.44	0.37	0.44	0.38	0.43	0.38	0.43	0.71	0.52	0.45	0.5	0.72	0.54	0.75	0.54
	Hash	0.44	1.35	0.44	1.34	0.45	1.3	0.44	1.35	8.46	2.02	0.55	1.67	0.86	2.1	0.89	2.09
	Single	0.37	0.20	0.37	0.19	0.38	0.18	0.38	0.19	0.71	0.25	0.45	0.23	0.71	0.27	0.75	0.25
	PA-Datalog	-	2.7	-	2.7	-	2.8	-	2.7	-	3.1	-	4.7	-	4.7	-	5
2o2h	Auto	0.37	0.8	0.37	0.8	0.38	0.79	0.38	0.79	0.71	0.87	0.45	0.84	0.72	0.89	0.75	0.89
	Maximal	0.37	1.68	0.37	1.68	0.38	1.67	0.38	2.05	0.71	1.77	0.45	2.18	0.72	1.78	0.75	1.79
	Hash	0.44	3.9	0.44	3.92	0.46	3.89	0.45	3.89	8.46	4.58	0.56	4.23	0.86	4.65	0.89	4.66
	Single	0.37	0.6	0.37	0.61	0.38	0.59	0.38	0.6	0.71	0.67	0.45	0.45	0.71	0.61	0.75	0.6
	PA-Datalog	-	3.3	-	3.7	-	3.7	-	3.3	-	3.7	-	4.9	-	5	-	5
3o3h	Auto	0.37	3.88	0.37	3.88	0.38	3.87	0.38	3.8	0.71	3.9	0.45	3.92	0.73	4.05	0.75	3.85
	Maximal	0.37	7.99	0.37	8	0.38	7.99	0.38	8	0.71	8.09	0.45	8.5	0.72	8.09	0.75	8.12
	Hash	0.44	17.3	0.45	19.43	0.46	19.41	0.46	19.41	8.46	15.9	0.56	19,76	0.86	20.17	0.89	20.2
	Single	0.37	2.99	0.37	2.99	0.38	2.98	0.38	2.92	0.71	2.98	0.45	3.05	0.71	3.11	0.75	2.95
	PA-Datalog	-	9.4	-	9.4	-	9.5	-	9.5	-	9.7	-	10.7	-	10.7	-	10.8

Figure 12: Memory usage for DOOP program analysis (GB)

lelize as well as B-trees. In some experiments with multiple cores, hash implementations showed limited scaling, unlike B-Trees.

7.2.5 Summary.

Overall, the experimental results demonstrate the value of our Auto indexing scheme for large-scale Datalog computation. During analysis, there is little extra work; and in executing, Auto runs with speed similar to (even faster than) Maximal, and using slightly more memory than Single. As Single is often too slow, and Maximal uses much memory, our Auto gives a good approach for processing large Datalog program such as those from program analysis, without needing the effort of hand optimization.

8. EXTENSIONS

Single Inequality. Although we limited the search predicate in our primitive search to be equalities of left-hand-side attributes and right-hand-side constants, our techniques can be extended for inequality constraints on one attribute: First, the bounds of the lex search predicate are to be adapted for the attribute of the inequality. Second, the attribute has to be the last one among the attributes in the search with respect to the lexicographical order. The ordering restriction is encoded in the bipartite graph $G = (U, V, E)$ by omitting edges in the standard construction. Specifically, there is an edge between $S \in U$ and $S' \in V$ if (1) S is a proper subset of S', (2) S has no inequality, and (3) if S' has an inequality on attribute x, then S does not have x. However, if there are multiple inequalities in a search, other techniques will be needed.

Loop Scheduling. Some Datalog engines such as Logicblox version 4 [24] use a leapfrog join that, while requiring users to specify indexes manually, alleviates users from specifying join order. Integrating our technique into such an engine is not obvious as we assume a fixed literal order before our technique is applied. Typically, this can be manually identified using a profiler, or automatically using heuristic techniques [34]. During performance tuning of large Datalog programs, we have observed that only a few rules require manual loop scheduling. Therefore, our preference is to fix loop orders rather than indexes for a better user experience.

Nevertheless, it will be an interesting future work to integrate automatic loop scheduling and automatic indexing selection.

9. CONCLUSION

We presented an automatic indexing scheme for large-scale Datalog applications that typically consist of hundreds of Datalog rules and millions of relation tuples. Such use cases could not previously be computed by state-of-the-art Datalog engines without considerable hand-crafted optimizations. We have formally defined the minimum index selection problem, aiming for a low memory footprint while still allowing every search to be sped up, and we proposed an algorithm to optimally solve this problem in polynomial time. Our technique has been implemented in the SOUFFLÉ Datalog engine, and measured to give fast speed and low memory. Our automatic indexing scheme releases end-users from a daunting obligation to carefully annotate Datalog programs, and it delivers comparable, even improved, performance to what they have accepted from making such efforts.

Acknowledgments. This research was supported partially by the Australian Government through the ARC Discovery Project funding scheme (DP180104030) and by the European Union's Horizon 2020 research and innovation program as part of the FETHPC AllScale project (No 671603).

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] T. Antoniadis, K. Triantafyllou, and Y. Smaragdakis. Porting doop to souffle;: A tale of inter-engine portability for datalog-based analyses. In *Proc. SOAP Workshop*, pages 25–30, 2017.

[3] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In *Proc. SIGMOD*, pages 1371–1382, 2015.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA*, pages 169–190, Oct. 2006.

[5] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proc. SIGMOD*, pages 37–48, 2011.

[6] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. OOPSLA*, pages 243–262, 2009.

[7] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, Oct. 2009.

[8] N. Bruno. *Automated Physical Database Design and Tuning*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2011.

[9] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, 1989.

[10] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. Association for Computing Machinery, Inc., June 1998.

[11] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proc. VLDB*, pages 146–155, 1997.

[12] D. Comer. The difficulty of optimum index selection. *ACM Trans. Database Syst.*, 3(4):440–445, Dec. 1978.

[13] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[14] J. Dietrich, N. Hollingum, and B. Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proc. OOPSLA*, pages 535–551, 2015.

[15] R. Dilworth. A decomposition theorem for partially ordered sets. *Ann. Math. (2)*, 51:161–166, 1950.

[16] D. R. Fulkerson. Note on dilworth's decomposition theorem for partially ordered sets. *Proc. Amer. Math. Soc.*, 7(4):pp. 701–702, 1956.

[17] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. In *Proc. OOPSLA (to appear)*, 2018.

[18] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.

[19] K. Hoder, N. Bjørner, and L. M. de Moura. *Z*- an efficient engine for fixed points with constraints. In *Proc. CAV*, pages 457–462, 2011.

[20] M. Ip, L. Saxton, and V. Raghavan. On the selection of an optimal set of indexes. *IEEE Trans. on Software Engineering*, SE-9(2):135–143, March 1983.

[21] H. Jordan, B. Scholz, and P. Subotic. Soufflé: On synthesis of program analyzers. In *Proc. CAV*, pages 422–430, 2016.

[22] J. Kratica, I. Ljubic, and D. Tošic. A genetic algorithm for the index selection problem. In *Proc. of EvoWorkshops*, pages 280–290, Berlin, Heidelberg, 2003. Springer-Verlag.

[23] LogicBlox and P. (UoA). PA-Datalog. `http://snf-705535.vm.okeanos.grnet.gr/agreement.html`, 2018. [Online; accessed 30-Jan-2018].

[24] LogicBlox Inc. Declartive cloud platform for applications that combine transactions & analytics. `http://www.logicblox.com`.

[25] M. Madsen, M.-H. Yee, and O. Lhoták. From datalog to flix: A declarative language for fixed points on lattices. *SIGPLAN Not.*, 51(6):194–208, June 2016.

[26] G. Piatetsky-Shapiro. The Optimal Selection of Secondary Indices is NP-complete. *SIGMOD Rec.*, 13(2):72–75, Jan. 1983.

[27] W. Pijls and R. Potharst. Another note on dilworth's decomposition theorem. *Journal of Discrete Mathematics*, 2013:4, 2013.

[28] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. In P. Dewilde and J. Vandewalle, editors, *Computer Systems and Software Engineering*, pages 287–324. Springer US, 1992.

[29] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1995.

[30] K. Ramamohanarao and J. Harland. An introduction to deductive database languages and systems. *PVLDB*, 3(2):107–122, 1994.

[31] M. Schkolnick. The optimal selection of secondary indices for files. *Information Systems*, 1(4):141 – 146, 1975.

[32] K. Schnaitter and N. Polyzotis. Semi-automatic index tuning: Keeping dbas in the loop. *PVLDB*, 5(5):478–489, 2012.

[33] B. Scholz, H. Jordan, P. Subotic, and T. Westmann. On fast large-scale program analysis in datalog. In *Proc. CC*, pages 196–206, 2016.

[34] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. SIGMOD*, pages 23–34, 1979.

[35] A. W. Services. Serverless Architectures with AWS Lambda. Technical report, Amazon Web Services, 11 2017.

[36] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on spark. In *Proc. SIGMOD*, pages 1135–1149, 2016.

[37] Y. Smaragdaiks, M. Bravenboer, and G. Kastrinis. Doop: A framework for java pointer analysis. `http://doop.program-analysis.org/`.

[38] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proc. APLAS*, pages 97–118, 2005.

[39] M. Yang, A. Shkapsky, and C. Zaniolo. Scaling up the performance of more powerful datalog systems on multicore machines. *The VLDB Journal*, 26(2):229–248, Apr. 2017.