Automatic Index Selection for Inequalities

SAMUEL ISAAC ARCH

SID: 470403402

Supervisor: Associate Professor Bernhard Scholz

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Engineering Honours (Software Engineering)

School of Computer Science
The University of Sydney
Australia

22 November 2020

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Samuel Isaac Arch

Signature: Samuel Isaac Arch **Date**: November 22, 2020

Abstract

In recent years, Datalog has surged in popularity as a domain-specific programming language for a variety of application domains such as static program analysis, network analysis and binary disassembly. Datalog allows the programmer to express the intended result of a computation in a declarative manner, resulting in concise logical specifications of programs. The resurgence of Datalog in modern computer science is primarily due to the development of high-performance Datalog engines such as SOUFFLÉ, with the capability to deliver performance competitive with imperative programs even when billions of tuples are involved. Crucial to the scalability of Datalog engines is their ability to accelerate the evaluation of logical rules by storing relations in data structures called *indexes*. Until very recently, users had to provide performance hints to Datalog engines to guide them to select certain indexes. Discovering the best performance hints to achieve satisfactory performance was not only painstaking but required a deep understanding of the underlying evaluation engine. To alleviate users from this burden of experimenting with performance hints, the state-of-the-art index selection technique deployed in SOUFFLÉ automatically selects the best set of indexes to accelerate Datalog rules, achieving performance on par with hand-crafted tools, without the need for user intervention. However, there is a crucial weakness to the state-of-the-art technique, as indexes only cover searches with *equality* constraints (known as equality primitive searches). As a consequence, key Datalog applications that require fast evaluation of searches with *inequality* constraints fail to meet real-world demands.

In this thesis, we introduce two new automatic index selection strategies that extend the state-of-the-art technology to accelerate rules with inequality constraints. We propose a gadget called a *spatial primitive search*, which abstracts indexable Datalog operations from the underlying choice of the index used to evaluate them. Our first strategy evaluates spatial primitive searches with a single R-Tree index to cover each relation. An alternate technique extends the state-of-the-art auto-index selection to cover spatial primitive searches having inequality constraints (denoted as inequality primitive searches) with a cluster of B-Tree indexes. We conduct experiments with industrial-strength real-world benchmarks, including DOOP, VPC and DDISASM. For these real-world benchmarks, we demonstrate that our novel approach incurs at most a 6% increase in compilation time and less than a 1% memory overhead while speeding up evaluation time to be up to $2.32\times$ faster.

Acknowledgements

First of all, an enormous thank you to my wonderful supervisor, Bernhard. Thank you for your incredible guidance, helpfulness and patience throughout the year. It was a pleasure working with you on such an exciting topic and learning about the world of computer science research. I've learnt so much from you, and I'm thrilled to see what we can achieve together in the future.

To my parents, Leonie and Michael, my girlfriend Jazlyn, and my friend Ive, thank you for your sincere support throughout the year.

A special thanks to Abdul, for mentoring me throughout this year and convincing me to do my honours thesis with Bernhard. Your help has been invaluable, and I couldn't have done it without you.

I want to thank Martin McGrane for lecturing me about C++ and software construction as well as reviewing and editing my code, helping me to become a better software developer. I also want to thank David for his words of wisdom throughout the year and Xiaowen for his help in integrating my changes into the SOUFFLÉ interpreter. I'd also like to thank David, Abdul, Xiaowen and Ive for helping me proofread this thesis.

Finally, I want to thank all of the remaining members of the programming languages research group, Rachel, Josh, Tytus, Xi, Sasha and Kamil for listening to my Friday presentations and for their fantastic feedback. It's been a great experience being part of this group and learning more deeply about programming languages research.

CONTENTS

Student Plagiarism: Compliance Statement **ii**

Abstract **iii**

Acknowledgements **iv**

List of Figures **viii**

List of Tables **x**

Chapter 1 Introduction **1**

1.1 Contributions 2

1.2 Organisation 3

Chapter 2 Background **4**

2.1 Datalog 4

2.1.1 Syntax and Semantics 4

2.1.2 Rule Evaluation 6

2.2 Indexing Relations in Datalog Engines 9

2.2.1 BDDBDDB 10

2.2.2 LogicBlox 11

2.2.3 SOUFFLÉ 12

2.3 Indexing Equality Primitive Searches in Datalog 14

2.3.1 Equality Primitive Searches 14

2.3.2 Hash Indexes 15

2.3.3 R-Tree Indexes 15

2.3.4 B-Tree Indexes 15

2.3.5 Equality Primitive Search to Lex Search 17

2.3.6 Covering Primitive Searches with Lex Searches 18

2.3.7 Minimum Index Selection Problem (MISP) 20

2.3.8 Implementation in SOUFFLÉ 22
2.4 Inequality Primitive Searches in Datalog Engines 23

Chapter 3 Spatial Primitive Searches **26**
3.1 Leveraging Inequality Constraints in Loop Nests 26
3.1.1 Datalog to Nested Loop Searches 26
3.1.2 Indexing Nested Loop Searches 28
3.2 Indexing Inequality Searches 29
3.2.1 Normalising Constraints to Weak Inequalities 29
3.2.2 Strengthening Filter Constraints 30
3.3 Abstracting Search Operations as Spatial Primitive Searches 33
3.4 Orthogonal Range Querying 35
3.4.1 Evaluating Spatial Primitive Searches as Orthogonal Range Queries 36
3.4.2 Choice of Index for Orthogonal Range Querying in Datalog Engines 37
3.4.3 k-d-trees 37
3.4.4 Range Trees 38
3.4.5 R-Trees 39
3.5 A Cluster of B-Trees for Spatial Primitive Searches 43
3.5.1 Lex Searches 44
3.5.2 Expressiveness of Lex Searches on Spatial Primitive Searches 45
3.5.3 Evaluating Spatial Primitive Searches with Multiple Range Constraints 46
3.5.4 The Minimum Index Selection Problem for Simple Spatial Primitive Searches 48
3.5.5 Characterising Indexes which Cover Simple Spatial Primitive Searches 50
3.5.6 Covering Simple Spatial Primitive Searches with a Common Index 50
3.5.7 Defining the Minimum Index Selection Problem (MISP) 50
3.5.8 Reduction to The Minimum Chain Covering Problem (MCCP) 51
3.5.9 Algorithms to Solve the MISP in Polynomial Time 54
3.5.10 Robustness and Efficiency of B-Tree SPS 55

Chapter 4 Experiments and Results **57**
4.1 (Q1) Frequency and Distribution of Spatial Primitive Searches 61
4.2 (Q2) R-Tree SPS vs Base 69
4.3 (Q3) B-Tree SPS vs Base 86
4.4 (Q4) R-Tree SPS vs B-Tree SPS 98

Chapter 5 Conclusion and Future Work 100
5.1 Future Work 101
5.1.1 Efficient Evaluation of Min/Max Aggregates 101
5.1.2 Improving Performance of Datalog Provenance 103
5.1.3 Index Selection with Partial Indexes 104
5.1.4 Dynamic Query Scheduling 106

Bibliography 108

Appendix A Spatial Primitive Searches and Orthogonal Range Queries 112

Appendix B Attributes with Inequalities Must be at the End of the k^{th} Prefix of ℓ 114

Appendix C Covering Search Set Pairs with B-Trees 116

List of Figures

2.1 Datalog Rules 6

2.2 Precedence Graph 6

2.3 Datalog Program 8

2.4 Tuples Produced in each Iteration 8

2.5 Indexed Loop-nest from Datalog rule. 10

2.6 Trie Representation of a Ternary Predicate (Aref et al., 2015) 11

2.7 SOUFFLÉ's Compilation Pipeline (Scholz et al., 2016) 13

2.8 Lexicographical Ordering of Two-dimensional Tuples with $\ell = x_1 \prec x_2$ 16

2.9 B-Tree Index with Lexicographical Ordering $\ell = x_1 \prec x_2$ 17

2.10 Transformation Pipeline of SOUFFLÉ (Scholz et al., 2016) 22

2.11 Naive Loop Nest 24

2.12 Filter Relation on c to Prune Search Space 24

2.13 Transformed Loop Nest with Filter Relation 24

2.14 Most Time Consuming Datalog Rule in DDISASM 25

3.1 Datalog Rule 27

3.2 Transformed Rule 27

3.3 Direct Translation of Datalog Rule to Loop Nest 27

3.4 Strengthened Loop Nest using Indexed Searches 28

3.5 Datalog Rule with Two Lower Bounds 30

3.6 Loop Nest with Indexed Searches 31

3.7 Datalog Rule with Two Equality Constraints 31

3.8 Indexed Searches 32

3.9 Filters Lifted Above Indexed Searches 32

3.10 Datalog Rule with Equality and Inequality Constraints 33

3.11 Filter Predicates Hoisted to Indexed Searches 33

3.12 Classes of Spatial Primitive Searches 34

3.13 A Spatial Primitive Search Interpreted as an Orthogonal Range Query 35

3.14 A k-d-tree cutting the plane into half-spaces with each non-leaf node (De Berg et al., 1997) 37

3.15 2-Dimensional Range Tree (De Berg et al., 1997) 38

3.16 Tree Structure of the R-Tree and Minimum Bounding Rectangles (MBRs) of Indexed Spatial Data 39

3.17 A Red MBR Must Expand to Accommodate the Blue MBR 40

3.18 Potential R-Tree Node Splitting Algorithms: (a) Naive (b) Linear (c) Quadratic 41

3.19 Loop-Nest Evaluation of a Datalog Rule 42

3.20 Strategy 1 47

3.21 Strategy 2 47

3.22 Venn Diagram for Sets of Tuples Satisfying Spatial Primitive Searches 47

4.1 Static Distribution of Spatial Primitive Searches (DDISASM) 62

4.2 Distribution of Attribute Constraints in Spatial Primitive Searches (DDISASM) 63

4.3 Nearby Naturals Program 69

4.4 Nearby Points Program 71

4.5 Tax Program 74

4.6 Insert Program with Arity $n = 3$ 76

List of Tables

4.1 Static Distribution of Spatial Primitive Searches 62

4.2 Attribute Constraint Distribution in Spatial Primitive Searches 63

4.3 Proportion of Rules with Inequality Primitive Searches 64

4.4 Dynamic Atom Frequency Distribution of Spatial Primitive Searches (DOOP) 65

4.5 Dynamic Atom Frequency Distribution of Spatial Primitive Searches (VPC) 65

4.6 Dynamic Run-Time Distribution of Rules (DOOP) 67

4.7 Dynamic Run-Time Distribution of Rules (VPC) 67

4.8 Number of Indexes (Nearby Naturals) 69

4.9 Maximum Resident Set Size (KB) and Evaluation Time (s) (Nearby Naturals) 70

4.10 Number of Indexes (Nearby Points) 72

4.11 Maximum Resident Set Size (KB) and Evaluation Time (s) (Nearby Points) 72

4.12 Number of Indexes (Tax) 74

4.13 Maximum Resident Set Size (KB) and Evaluation Time (s) (Tax) 74

4.14 Number of Indexes (Insert) 76

4.15 Maximum Resident Set Size (KB) and Evaluation Time (s) (Insert) 77

4.16 Number of Indexes and Compilation Time (s) (DOOP) 78

4.17 Maximum Resident Set Size (KB) and Evaluation Time (s) (DOOP) 79

4.18 Number of Indexes (VPC N-1075 sec1) 81

4.19 Number of Indexes (VPC N-1075 sec2) 81

4.20 Number of Indexes (VPC N-1075 sec3) 81

4.21 Compilation Time (s) of Index Selection Schemes (VPC) 82

4.22 Maximum Resident Set Size (KB) and Evaluation Time (s) (VPC) 82

4.23 Number of Indexes and Compilation Time (s) (DDISASM) 83

4.24 Maximum Resident Set Size (KB) and Evaluation Time (s) (DDISASM) 84

4.25 Number of B-Tree Indexes (Nearby Naturals) 86

4.26 Maximum Resident Set Size (KB) and Evaluation Time (s) (Nearby Naturals) 86

4.27 Number of Indexes (Nearby Points) 87

4.28 Maximum Resident Set Size (KB) and Evaluation Time (s) (Nearby Points) 88

4.29 Number of Indexes (Tax) 88

4.30 Maximum Resident Set Size (KB) and Evaluation Time (s) (Tax) 89

4.31 Number of Indexes (Insert) 90

4.32 Maximum Resident Set Size (KB) and Evaluation Time (s) (Insert) 90

4.33 Number of Indexes and Compilation Time (s) (DOOP) 92

4.34 Maximum Resident Set Size (KB) and Evaluation Time (s) (DOOP) 93

4.35 Number of Indexes (VPC N-1075 sec1) 93

4.36 Number of Indexes (VPC N-1075 sec2) 93

4.37 Number of Indexes (VPC N-1075 sec3) 94

4.38 Compilation Time (s) of Index Selection Schemes (VPC) 94

4.39 Maximum Resident Set Size (KB) and Evaluation Time (s) (VPC) 95

4.40 Number of Indexes and Compilation Time (s) (DDISASM) 95

4.41 Maximum Resident Set Size (KB) and Evaluation Time (s) (DDISASM) 96

CHAPTER 1

Introduction

Logic programming languages such as Datalog have gained increasing popularity in recent years in various application domains including program analysis (Bravenboer and Smaragdakis, 2009), data integration (Lenzerini, 2002), network analysis (Loo et al., 2006), smart contract analysis (Brent et al., 2020) and binary disassembly (Flores-Montoya and Schulte, 2020). Logic programming provides declarative semantics allowing the programmer to express the logic of a computation without specifying its control flow. As a consequence, programs can be represented succinctly, without the programmer concerning themselves with the implementation details of how the underlying program is evaluated. However, typical application domains of Datalog such as static program analysis operate at giga-scale with potentially billions of tuples (Antoniadis et al., 2017). Thus, the ability of logic programs to perform competitively with manually developed tools when faced with large-scale data sets is crucial.

Datalog programs are evaluated by repeated application of logical rules, deriving further knowledge until reaching a fix-point. Logical rules are computed as a series of searches over the involved logical relations. Searches can be accelerated by deploying high-performance data structures, called *indexes*, to store the underlying logical relations. However, when searches cannot use an index, they are performed naïvely as a scan over the entire relation and then are filtered to retrieve only tuples that satisfy the search. Evaluating these searches without an index results in dramatic slowdowns during evaluation time, with logic programs becoming utterly intractable to compute. Consequently, the selection of indexes is one of the most critical factors affecting the evaluation time of logic programs. Index selection is to logic programming as register allocation is to imperative programming. Each code generation optimisation is crucial to ensure that data involved in computations can be retrieved swiftly.

The state-of-the-art automatic index selection algorithm (Subotić et al., 2018) deployed in the SOUFFLÉ Datalog engine, selects the smallest set of B-Tree indexes to cover all searches over a given Datalog relation. Minimising the number of indexes is crucial, as replicating data across multiple indexes incurs a memory overhead and maintenance cost in the form of run-time. The technique proved to be exceptionally useful, enabling evaluation of Datalog programs with speeds competitive with hand-crafted tools.

However, only a particular class of searches are accelerated by indexes using the state-of-the-art technique. Specifically, indexes only cover search operations with *equality* constraints on the relation's attributes, denoted as equality primitive searches. In recent years there have been vital applications of Datalog, such as DDISASM (Flores-Montoya and Schulte, 2020), which express their semantics using searches with *inequality* constraints, called inequality primitive searches. Although the presence of inequality primitive searches in Datalog programs is sparse, they are evaluated naïvely without an index causing their execution to dominate the program evaluation time.

In this thesis, we introduce the notion of a *spatial primitive search*, to represent equality or inequality primitive searches on a Datalog relation. The concept of a spatial primitive search serves to distil searches on Datalog relations to their semantics as filter operations without concern for the underlying choice of the index used to evaluate them. We then propose two new automatic index selection schemes to accelerate all spatial primitive searches in a Datalog program. The first technique, R-Tree SPS (Spatial Primitive Search), uses a single R-Tree index for each relation to speed up inequality primitive searches and defaults to the original index selection scheme to cover relations with exclusively equality primitive searches. The other technique, B-Tree SPS (Spatial Primitive Search), extends the original scheme, covering all simple spatial primitive searches (i.e. spatial primitive searches with at most one attribute with an inequality constraint) with a cluster of B-Trees.

1.1 Contributions

We present the following contributions in this thesis:

- We have formally defined the notion of a spatial primitive search, to encapsulate search operations on Datalog relations. We use spatial primitive searches as an abstract device to focus on the automatic index selection framework without concern for the underlying choice of index.
- We have established a formal equivalence between the semantics of spatial primitive searches and orthogonal range queries. We then implemented an R-Tree indexing strategy, R-Tree SPS, to cover all spatial primitive searches on a relation with a single index.

- We extended the current state-of-the-art automatic index selection scheme to speed up search operations using inequality constraints, which we denote as B-Tree SPS. The key idea is to define a partial ordering over simple spatial primitive searches (search operations with at most one attribute constrained by an inequality predicate) and reuse the existing algorithmic framework to find the best index selection.
- We have implemented both index selection schemes in SOUFFLÉ. The implementation required the creation of new relational algebra transformers, implementing lower and upper bounds for search operations on relations, modifying the index analysis scheme in the relational algebra intermediate representation, integrating the C++ Boost R-Tree data structure into SOUFFLÉ, and implementing attribute type comparators for synthesised data structures. The B-Tree SPS implementation was merged across nine separate pull requests consisting of over 4000 lines of code as an open-source contribution to SOUFFLÉ, publicly accessible here: (Samuel, 2020b). The experimental evaluation was performed on a fork of SOUFFLÉ containing both index selection implementations, which can be found here: (Samuel, 2020a).
- We have conducted extensive experiments using large scale benchmarks such as DOOP, VPC and DDISASM. We measure the overhead concerning compilation time, maximum memory usage and evaluation time speed-up relative to the state-of-the-art auto-index scheme. The experimental analysis showcases that B-Tree SPS increases compilation time by at most 6% and increases peak memory usage by less than 1% while demonstrating a speed-up in evaluation time of up to $2.32\times$. At the same time, B-Tree SPS is robust, showcasing no performance degradation within the margins of experimental error for real-world Datalog applications when compared to the state-of-the-art technique. We also compare our index selection strategy with R-Tree SPS.

1.2 Organisation

The thesis is structured as follows: In Chapter 2, we cover related work and the state-of-the-art automatic index selection scheme used in SOUFFLÉ. In Chapter 3, we present the theory of spatial primitive searches, detail how to evaluate them and illustrate how to select indexes to cover them with both the R-Tree SPS and B-Tree SPS indexing schemes. In Chapter 4, we present our experiments to determine the feasibility of our proposed index selection techniques and discuss their results. Finally, in Chapter 5, we form our conclusions and discuss future work.

CHAPTER 2

Background

In this chapter, we give a summary of the Datalog language, existing work in the area of index selection in Datalog engines and the current techniques in the literature for evaluating searches with inequality constraints in Datalog. We begin by introducing the syntax and semantics of Datalog and the evaluation strategies used by modern Datalog engines. We then review the indexing techniques deployed in modern Datalog engines and the state-of-the-art automatic index selection strategy used in SOUFFLÉ. Finally, we conclude by exploring the existing work to evaluate searches with inequality constraints in Datalog.

2.1 Datalog

Datalog is a declarative programming language, and a fragment of first-order logic (Abiteboul et al., 1995) used initially as a relational database query language. Declarative languages such as Datalog allow the programmer to express *what* to compute rather than *how* to compute it. Datalog distinguishes itself from other database query languages due to its expressiveness and ease of use for solving problems in complex application domains.

2.1.1 Syntax and Semantics

A Datalog *rule* is a Horn clause of the form:

$$R_0(X_0) :- R_1(X_1), R_2(X_2), ..., R_n(X_n).$$

Each R_i is a *relation* and each X_i is a sequence of variables and constants matching the *arity* of the relation. Each $R_i(X_i)$ is an *atom* $R(x_1, x_2, ..., x_m)$ where each symbol x_i is either a variable or a constant.

The left-hand side of the rule (R_0) denotes the *head*, whereas the right-hand side of the rule $(R_1, R_2, ..., R_n)$ is known as the *body* of the rule. A concrete instance of a relation is a set of *tuples* where variables are substituted with appropriate constants.

A Datalog rule has the following interpretation: If every atom in the body of a rule is true, then the head of the rule is deduced to be true. In mathematical logic we would write:

$$R_0 \leftarrow R_1 \land R_2 \land ... \land R_n$$

A Datalog program also consists of *facts*. A fact is a rule with an empty body, i.e. $(R_0 :- .)$ or $(R_0.)$ and therefore, the head of the clause is unconditionally true. Rules and facts are both different types of knowledge. The rules deduce further knowledge from this initial set of facts, repeatedly executing until no new knowledge can be found.

Consider as an example the following Datalog rule:

$$grandparent(x, z) :- parent(x, y), parent(y, z).$$

We say that if $parent(x, y) \land parent(y, z)$ holds then $grandparent(x, z)$ consequently holds. Concretely, if we have $parent(\text{"}alice\text{"}, \text{"}bob\text{"})$ and $parent(\text{"}bob\text{"}, \text{"}carol\text{"})$ in our set of facts, then a deduction is made from the application of the above rule to deduce a new tuple $grandparent(\text{"}alice\text{"}, \text{"}carol\text{"})$.

Atoms in Datalog rules can also appear negated (written as $\neg A(X)$). A negated atom holds when the negated predicate, i.e. $A(X)$ does not hold. The introduction of negation to Datalog elevates its expressiveness but creates further complications. For example, the rule: $R(X) :- \neg R(X).$ is not logically sound yet is a valid rule if recursive negation is permitted. Naturally, the structure of rules containing negation requires further restrictions in order to ensure meaningful program semantics. To resolve this, *Stratified Datalog* semantics are usually adopted (Abiteboul et al., 1995).

Stratified Datalog orders relations into fixed strata and evaluates the relations from the bottom strata upward. The relevant rules for the relations in each stratum are applied repeatedly until no new tuples can be derived. These tuples are then passed upward into the next stratum as facts. An atom can only appear negated in a rule if the corresponding relation appears in a lower stratum. To evaluate the truth of a negated atom, one can perform a simple check for whether any satisfying tuples exist as facts from lower strata.

A precedence graph is built from the relations in a program to determine each stratum. For each rule, we construct directed edges from relations appearing in the rule body to the relation at the head of the rule. A topological ordering over the strongly connected components of the precedence graph then determines the placement of relations into fixed strata.

$$A(x) :\!-\ B(x).$$

$$B(x) :\!-\ C(x), \neg D(x).$$

$$C(x) :\!-\ A(x).$$

FIGURE 2.1. Datalog Rules

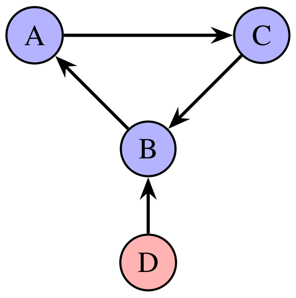


FIGURE 2.2. Precedence Graph

In the above example, A, B and C form a strongly connected component and D forms its own strongly connected component. Therefore, since D precedes the other strongly connected component in the topological ordering we place D into Stratum 0 and A, B and C into Stratum 1. Although multiple valid stratifications can exist for a single Datalog program, the final result of the computation is always the same (Abiteboul et al., 1995; Greco and Molinaro, 2015). If a topological ordering does not exist, then the program is unstratified and invalid.

In Datalog, a relation (and its corresponding tuples) are either *extensional* or *intensional*. A relation is extensional or in the extensional database (EDB) if the relation consists of only facts and no rules. All other relations are called intensional or in the intensional database (IDB). Intuitively, tuples from extensional relations are the "input" to the program. The Datalog program then uses the tuples in the EDB (the facts) to produce the IDB which is analogous to the "output" of a procedural program.

2.1.2 Rule Evaluation

Datalog as a language specification does not require that any particular evaluation strategy is adopted. As a result, different Datalog engines have adopted different evaluation strategies which are broadly classified as either *top-down* or *bottom-up*. Bottom-up evaluation is utilised almost exclusively in modern Datalog engines including SOUFFLÉ (Jordan et al., 2016), μZ (Hoder et al., 2011) and LogicBlox

(Aref et al., 2015) whereas top-down evaluation strategies were deployed in older engines such as XSB (Sagonas et al., 1994).

Top-down evaluation operates by working from a query (known as the *goal clause*) down to the facts in the EDB. If the goal clause resolves to facts that all exist in the EDB, the goal clause must hold.

$$:-\ A_1, A_2, ..., A_n$$

For each atom A_i, top down-evaluation tries to match the atom with the head of a rule. The atom in the goal clause gets replaced by the body of the matching rule. This process, known as *unification*, is applied repeatedly until facts in the EDB have entirely replaced the goal clause, or no satisfying resolution of the query can be found.

An example of a top-down evaluation strategy is SLD resolution. The algorithm mechanically attempts to replace each atom from left to right in the goal clause, substituting rules in the order that they appear in the program. The algorithm continues until either a satisfying resolution is produced or all substitution paths lead to failure.

Top-down evaluation strategies have the weakness that they tend to apply the same substitutions during the unification process repeatedly. Tamaki and Sato (1986) improve on SLD resolution with OLD resolution using a technique known as variant tabling, which stores previously encountered truth values of atoms in tables for future lookup. Tekle and Liu (2011) further improve this method with subsumptive tabling, demonstrating that it outperforms bottom-up evaluation with the magic set transformation (MST) applied (Ross, 1990).

Interestingly, top-down evaluation is not guaranteed to terminate as a rule such as $A(X)\ :-\ A(X)$ will never halt. In the process of unifying the atom $A(X)$, it matches with the head atom of the same rule (also $A(X)$) and is substituted with the same atom infinitely. Bottom-up evaluation differs in this regard as it is guaranteed to terminate (Greco and Molinaro, 2015; Abiteboul et al., 1995).

Bottom-up evaluation occurs on an instance I of a Datalog Program P beginning with a set of facts in the EDB and terminating with a completed IDB. It operates in successive iterations, with each iteration deriving tuples satisfying bodies in any of the rules (formally this is the application of the *immediate consequence operator* Γ_P to I) and adding them into the IDB. The evaluation terminates when a fixpoint occurs, and no more IDB tuples are derived. Due to the ease of implementation, Bancilhon and Ramakrishnan (1989) explain that this naïve evaluation is the most discussed in the literature. However, naïve bottom-up evaluation tends to result in the production of tuples that already exist within the IDB, resulting in large amounts of redundant computation. Consider the following naïve evaluation of a Datalog program:

$edge(a, b), edge(b, c).$
$edge(c, c), edge(c, d).$

$path(x, y) :\!- edge(x, y).$
$path(x, y) :\!- edge(x, z), path(z, y).$

FIGURE 2.3. Datalog Program

$(1) : \emptyset$
$(2) : \{(a, b), (b, c), (c, c), (c, d)\}$
$(3) : \{(a, b), (b, c), (c, c), (c, d), (a, c), (b, d)\}$
$(4) : \{(a, b), (b, c), (c, c), (c, d), (a, c), (b, d), (a, d)\}$

FIGURE 2.4. Tuples Produced in each Iteration

In the above evaluation, we consider the state of the $path$ relation in the IDB on each iteration of the naïve algorithm. Initially, there are no IDB tuples for $path$. After the first iteration, we have produced corresponding IDB tuples for each $edge$ fact. On subsequent iterations, since every rule is applied to the existing EDB and IDB, all of the tuples discovered in iteration (2) are rediscovered and then discarded. Deriving duplicate tuples is the primary weakness of the naïve evaluation strategy as the entire EDB and IDB gets passed as input for each subsequent iteration.

This weakness of the naïve algorithm was recognised by Bancilhon (1986) who substantially improved on it with the introduction of *semi-naïve* evaluation. The critical insight of semi-naïve evaluation is to use delta relations to store new knowledge in the IDB, passing only new knowledge forward in each iteration of the evaluation. Semi-naïve evaluation has much more robust performance guarantees than the naïve evaluation strategy, ensuring that for any particular tuple, it will only be derived in the same way exactly once.

Despite neither bottom-up nor top-down evaluation for a given program P being decidedly faster than another (Bancilhon and Ramakrishnan, 1989), Ullman (1989) proved a fundamental result about the semi-naïve evaluation strategy. Ullman demonstrated that for a given Datalog program P, it can always be transformed into a semantically equivalent program P' with a guarantee on the number of rule firings. In particular, when executing P' with a semi-naïve evaluation strategy, the number of rule firings is never more than that of evaluating the original program P using a top-down evaluation strategy. Effectively, as long as the original program P is optimised appropriately through program transformation techniques such as the Magic Set Transformation, semi-naïve evaluation is the optimal strategy. For these reasons, semi-naïve evaluation is the chosen strategy for nearly all modern Datalog engines.

2.2 Indexing Relations in Datalog Engines

In recent years, Datalog engines such as BDDBDDB , LogicBlox and SOUFFLÉ have been developed to execute Datalog programs efficiently using semi-naïve evaluation. These engines translate a given logic program into an equivalent imperative program, taking advantage of modern computer architectures to achieve high performance. Although, significant performance gains can be achieve with source-level program transformations such as the Magic Set Transformation (Ullman, 1989), the underlying evaluation of logical rules is most crucial to achieving performance on-par with hand-crafted imperative programs.

To translate a Datalog program into an equivalent imperative program, modern Datalog engines unroll each logical rule into a *loop nest* over the relations involved in the rule. It is worthwhile to note that although a Datalog rule can unroll into multiple valid loop-nests, the result of the computation will remain the same as the relational algebra of each order is equivalent (Chaudhuri, 1998).

$$R_{k+1}(x_{k+1}) :- R_1(x_1), ..., R_k(x_k).$$

Datalog Rule

```
for all $t_1 \in R_1$ do
  ...
    for all $t_k \in R_k$ do
      if $p_1(t_1)$ do
        ...
          if $p_k(t_k)$ do
            if $(...) \notin R_{k+1}$ do
              project $(...)$ into $R_{k+1}$
```

Corresponding Loop-nest Where $p_i(t_i)$ is an Equality Predicate on Tuple t_i.

The above figure showcases a naïve translation of a Datalog rule to a corresponding imperative loop nest. The advantage of the loop-nest is that no intermediate results are materialised and instead each element of the Cartesian product of the involved relations is iterated through as it is needed. However, the performance of the loop nest is its greatest weakness.

For each loop in the loop nest, a table scan is performed on the relation with time complexity of $\mathcal{O}(|R_i|)$. Therefore, the overall complexity of the loop-nest is $\mathcal{O}(\prod_i |R_i|)$, excluding the time to project any tuples into the relation appearing in the head of the rule. Since the sizes of these relations are typically very large in real-world Datalog applications, evaluating logic programs using this naïve approach becomes completely intractable.

However, instead of generating the Cartesian product over all relations and then filtering on the corresponding predicates lower in the loop nest, these predicates can be hoisted into *equality primitive searches* on the relations. Equality primitive searches are search operations on relations where some subset of the relation's attribute have equality predicates.

```
for all $t_1 \in R_1$ on index $p_1(t_1)$do
  ...
    for all $t_k \in R_k$ on index $p_k(t_k)$do
      if $(...) \notin R_{k+1}$ do
        project $(...)$ into $R_{k+1}$
```

FIGURE 2.5. Indexed Loop-nest from Datalog rule.

Figure 2.5 illustrates the transformed loop nest where table scan and filter operations are combined to form equality primitive searches. We note that predicates can only be hoisted up into table scans to form equality primitive searches when values in the predicate are constants or are determined by variables at higher levels of the loop nest. For each of the equality primitive searches in the loop nest, an index can accelerate the evaluation. In particular, if using an index such as a B-Tree, the data structure can be queried for the tuples satisfying the search predicate with a complexity of $\mathcal{O}(\log(n) + |Q|)$ where Q is the set of tuples satisfying the range query. By evaluating searches with an index, each loop reduces in complexity from being proportional to the size of the relation to being proportional to the size of the output. Since $|Q|$ is typically significantly smaller than n, evaluating equality primitive searches with indexes is crucial to the high-performance evaluation of logic programs. The choice of index to store logical relations is, therefore, the most crucial factor for modern Datalog engines to cope with giga-scale data common in many application domains (Jordan et al., 2016).

We now shift focus to the index selection strategies deployed in modern Datalog engines.

2.2.1 BDDBDDB

The first scalable, context-sensitive, inclusion-based, pointer alias analysis for Java programs in Datalog became feasible with the BDDBDDB Datalog engine (Whaley et al., 2005). Previously, these large scale context-sensitive analyses were completely intractable as the analysis operated on call graphs with as many as 10^{14} acyclic paths (Whaley and Lam, 2004). Storing these paths explicitly in logical relations would quickly exceed the available system memory of modern machines. The choice of binary decision diagrams (BDDs) (Akers, 1978) as an index for logical relations allows tuples to be stored implicitly as compressed truth tables making the analysis tractable. BDDs as an index for logical relations are not without fault, however. Bravenboer and Smaragdakis (2009) suggests that BDDs are not the most suitable representation for fast and precise context-sensitive analysis. The authors note that the overall performance of the analysis is highly sensitive to the variable ordering chosen for large BDDs, noting that the problem of finding an optimal variable ordering for a single BDD is an NP-Hard problem (Meinel and Theobald, 2012). Therefore, extensive experimentation with different variable orderings is required to achieve satisfactory performance (Berndl et al., 2003).

2.2.2 LogicBlox

LogicBlox (Aref et al., 2015) is a more recent Datalog engine, designed to reduce the inherent complexity in developing modern enterprise applications (Green et al., 2012). LogicBlox is not specialised for any specific application domain. Therefore the designers opt for a more general Trie (Fredkin, 1960) data structure with the intent to provide fast performance in a variety of application contexts. Despite, LogicBlox lacking specialisation for program analysis, it has outperformed BDDBDDB due to its high performance as an evaluation engine (Bravenboer and Smaragdakis, 2009). Tries, much like BDDs, need not store the data of logical relations explicitly, with shared prefixes of tuples stored only once. Figure 2.6 demonstrates the motivation for a Trie index, with a high level of information density achievable when many tuples have shared prefixes relative to the variable order of the relation. LogicBlox stores each relation in a single Trie index in the same variable order that the relation uses, which need not have many shared prefixes among tuples.

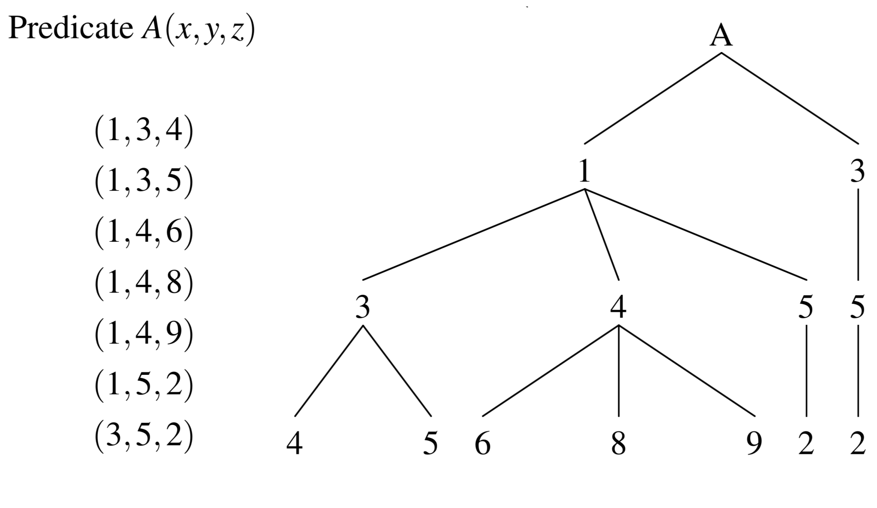


FIGURE 2.6. Trie Representation of a Ternary Predicate (Aref et al., 2015)

Furthermore, a novel join algorithm, the *Leapfrog Trie Join* (LFTJ) (Veldhuizen, 2012) is employed to provide efficient searches on indexes. When searching an index on a prefix of the variable order, only the direct children of the relevant sub-tree need to be visited. However, in other cases, the search performance can be abysmal. The worst performance occurs when searching on the last attribute in the relation's variable order, requiring iteration over the entire index to find all of the satisfying tuples. The performance of LogicBlox is highly sensitive to the variable order, not only with regards to the information density achievable within relations but also the time complexity of search operations across indexes. DOOP (Bravenboer and Smaragdakis, 2009), a popular framework for points-to analysis in Java, was first implemented in LogicBlox. The performance of DOOP crucially relied on a code-rewriting technique (Antoniadis et al., 2017), whereby relations were rewritten with different variable orderings, replicating a relation multiple times to optimise the performance of searches. The choice of variable orderings for each relation and which relations to duplicate required not only extensive manual human time and effort but deep familiarity with the underlying implementation of the Datalog engine.

2.2.3 SOUFFLÉ

SOUFFLÉ (Jordan et al., 2016) represents the state of the art in Datalog engine technology, initially developed in Oracle Labs Brisbane for program analysis of the OpenJDK. SOUFFLÉ was invented to fulfil the need to execute Datalog programs at speeds competitive with manually optimised imperative tools. Other Datalog engines lacked this ability to specialise for a specific program instance. SOUFFLÉ meets this demand by employing novel specialisation techniques through the use of Futumura projections (Futamura, 1999) to translate a given Datalog program into a highly efficient parallel C++ program. First, the Datalog program is parsed to an Abstract Syntax Tree (AST) where high level optimisations are performed. Next, the AST is converted to relational algebra operations to be performed on an abstract machine (known as the Relational Algebra Machine or RAM). In this format, mid-level optimisations are performed. After this, the RAM representation is converted to high performance concurrent C++ that executes relational algebra operations. Finally, this C++ code is compiled into a binary, using the powerful optimisation capabilities of modern C++ compilers such as GCC and Clang.

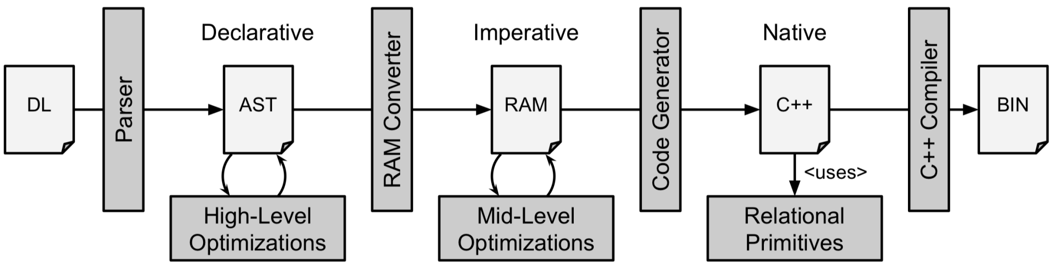


FIGURE 2.7. SOUFFLÉ's Compilation Pipeline (Scholz et al., 2016)

Unlike other Datalog engines which employ a single fixed data structure to index every relation, SOUFFLÉ supports a portfolio of different data structures depending on the workload (Jordan et al., 2020). The choice from different indexes is made available due to the fact the each of the highly specialised concurrent data structures deployed in SOUFFLÉ conform to a uniform interface. The default data structure within the engine is a highly optimised concurrent B-Tree (Jordan et al., 2019b) achieving up to $3\times$ faster Datalog evaluation performance than industry standard concurrent set implementations. For workloads with large volumes of data, a custom high-performance cache-friendly trie data structure specialised for concurrent Datalog is deployed (Jordan et al., 2019a). When used in a points-to analysis, it offers an impressive $4\times$ improvement over SOUFFLÉ's B-Tree, due to its ability to more densely represent a relation's tuples. Finally, a high-performance parallel union-find data structure known as EQREL (Nappa et al., 2019) is utilised for efficiently evaluating equivalence relations.

The most important distinction, however, between SOUFFLÉ and other Datalog engines is that SOUFFLÉ automatically selects a minimal set of indexes that cover every equality primitive search for relations in a given Datalog program. The technique frees the programmer from considerable human effort spent manually rewriting relations to achieve satisfactory performance. We note that the index selection technique is applied given a fixed loop schedule. Given that the scheduling of loops can influence performance dramatically, this is not ideal as users still need to manually find the best loop schedules to achieve satisfactory performance. However, this choice was made by Subotić et al. (2018) since manual tuning of loop schedules is significantly less time consuming than finding a satisfying index selection.

2.3 Indexing Equality Primitive Searches in Datalog

A variety of indexes can accelerate the evaluation of equality primitive searches such as hash indexes, multi-dimensional indexes of binary search tree indexes. However, the state-of-the-art auto-index selection strategy deployed in SOUFFLÉ has opted to use B-Tree indexes to speed up equality primitive searches. We review their formal definition of equality primitive searches and their rationale for selecting B-Tree indexes.

2.3.1 Equality Primitive Searches

Subotić et al. (2018) define the preliminaries. Firstly, a *relation* R is defined to be a subset of an m-ary Cartesian product $\mathcal{D} = \{\mathbb{D}_1 \times ...\mathbb{D}_m\}$ where $\mathbb{D}_i$ are the *domains* of each relation. Elements of relations are called *tuples*. A tuple $t_i = \langle e_1, e_2, ..., e_m\rangle \in R$ has a fixed arity m, where $e_i \in \mathbb{D}_i$ for $1 \leq i \leq m$. For a relation R it's attributes $A_R = \{x_1, x_2, ..., x_m\}$ refer to specific positions of elements within its tuples. $R(x_1, x_2, ..., x_m)$ is written to associate symbol x_i with position i in the tuple. Finally, $t(x_i)$ is an access function that maps an attribute x_i to the concrete value of a tuple t at position i.

For example, if we have a binary relation R where $\mathbb{D}_1 = \{x \in \mathbb{Z} \mid 0 \leq x \leq 2^{32} - 1\}$, $\mathbb{D}_2 = \{y \in \mathbb{Z} \mid -2^{31} + 1 \leq x \leq 2^{31}\}$ then $t = \langle 3, -4\rangle$ would be a valid tuple since $t(x_1) = 3 \in \mathbb{D}_1$ and $t(x_2) = -4 \in \mathbb{D}_2$. A domain $\mathbb{D}_i$ in Datalog is helpful because it allows us to define the range of values that are attainable by an element e_i given the type of the attribute.

An equality primitive search is of the following form:

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

R_i denotes a given relation, $x_1, x_2, ..., x_k$ are attributes of the relation R_i and $x_1 = v_1, x_2 = v_2, ..., x_k = v_k$ denote search predicates where $v_1, v_2, ..., v_k$ are constants.

It is worth noting that the attributes that appear in a search predicate do not have to be the first k in the relation. Instead, any of the attributes $x \in A_R$ may appear on the left hand side of search predicates. Additionally, the values $v_1, v_2, ..., v_k$ can be constants from t or may appear in tuples that appear above the current tuple in the loop nest.

2.3.2 Hash Indexes

When deciding on an index to speed up these equality primitive searches, there is a wealth of options. An obvious choice would be to use a hash index due to their ubiquity in SQL databases (Blanas et al., 2011). However, Subotić et al. (2018) found these hash indexes to not be very competitive for a few reasons. Hash-based data structures such as Google's sparse hash-map were found to not perform as well as B-Trees. The authors cite a poor trade-off of time to perform searches and memory usage as well as a failure to parallelise computation across multiple machine cores. A hash-index also has a significant weakness in that multiple hash indexes are needed to cover every search with an index. For example, if there are $|\mathcal{S}|$ distinct equality primitive searches where each search contains a unique set of attributes in the search predicate, $|\mathcal{S}|$ hash indexes must be constructed to cover them all. For each hash index built for the relation, tuples need to be inserted into all hash indexes for their states to be kept consistent. Therefore, not only are hash indexes too slow and hard to parallelise, but their memory usage becomes excessive for real-world Datalog programs.

2.3.3 R-Tree Indexes

Subotić et al. (2018) remark that a multi-dimensional index such as an R-Tree (Guttman, 1984) would cover all equality primitive searches in a Datalog program with a *single* index. In other words, a single R-Tree index for a given relation would cover all $|\mathcal{S}|$ equality primitive searches eliminating the need for replication of database tuples. Unlike hash-maps, R-Tree variants exist that effectively exploit multi-core architectures (Kamel and Faloutsos, 1992). R-trees also tend to perform quite well in practice (Hwang et al., 2003), however, performance is highly sensitive to the distribution of values and the order they are inserted (Beckmann et al., 1990). In the worst case, the query complexity is $\mathcal{O}(n)$ where n is the size of the relation i.e. every tuple is visited. Only when the data is well organised in the R-Tree does the query performance become competitive. The authors did not evaluate the performance of R-tree indexes, despite the ability of a single R-Tree index to cover all equality primitive searches of a relation.

2.3.4 B-Tree Indexes

Subotić et al. (2018) ultimately opted to use a B-Tree index to cover equality primitive searches. Although multiple B-Tree indexes may be required to speed up all equality primitive searches for a relation, they provide strong guarantees on the query complexity. In particular, B-Trees exhibit $\mathcal{O}(\log(n) + |Q|)$

worst case query performance where Q is the set of tuples satisfying the search. In other words, the query complexity is proportional to the size of the output. Therefore, when the size of the relation is large and the number of tuples satisfying the search is small, B-Tree indexes offer very strong performance.

Unlike a multidimensional index such as an R-Tree which stores each attribute of a multi-dimensional tuple as it's own coordinate in n-dimensional space, each tuple is compared by it's position in a total ordering. A comparison scheme is employed to compare each of these multi-dimensional tuples, enabling a traditionally uni-dimensional index such as a B-Tree to handle multi-dimensional data.

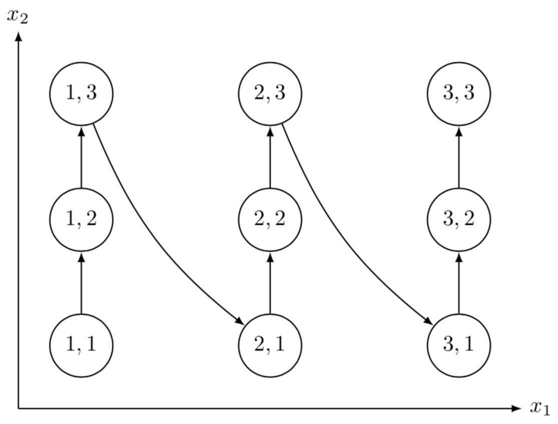


FIGURE 2.8. Lexicographical Ordering of Two-dimensional Tuples with $\ell = x_1 \prec x_2$

The authors employ a *lexicographical* ordering to compare tuples for a given index, by considering a total ordering over the attributes. $\ell = x_1 \prec x_2 \prec ... \prec x_m$ defines an attribute sequence which specifies the order that attributes are considered in when comparing between tuples. Two tuples are compared first by the value of the attribute x_1, terminating at this point if either value is larger. However, if their x_1 values are equal then their x_2 values are compared, continuing in this fashion until one attribute value is found to be larger than the other, or all attribute values coincide, and the tuples are considered to be equal. Figure 2.8 illustrates how a total ordering can be constructed over a set of two-dimensional tuples.

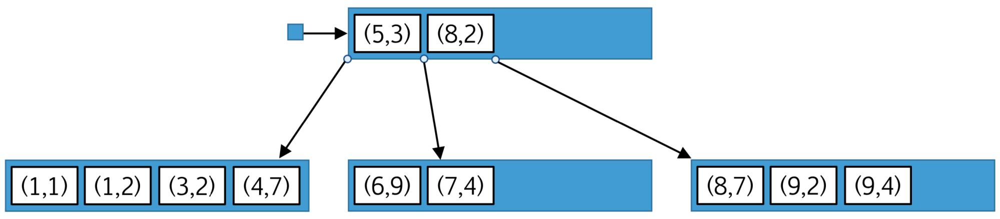


FIGURE 2.9. B-Tree Index with Lexicographical Ordering $\ell = x_1 \prec x_2$

Using ℓ to order all attributes in a relation, $\sqsubseteq_\ell \mathcal{D} \times \mathcal{D}$ denotes a total order over the domain $\mathcal{D}$ of a relation for ℓ. Given two tuples $a, b \in \mathcal{D}$, when $(a, b) \in \sqsubseteq_\ell \mathcal{D} \times \mathcal{D}$ we say that a is less than b or more concisely $a \sqsubseteq_\ell b$. Since the tuples form a total order for the lex-order chosen, it is possible for only one of $a \sqsubseteq_\ell b$ or $b \sqsubseteq_\ell a$ to be true. By building a B-Tree index for a given ℓ, the index can cover a subset of the equality primitive searches. Figure 2.9 illustrates how B-Tree indexes can be constructed given a lexicographical ordering ℓ.

When multiple indexes are needed to cover a collection of equality primitive searches, multiple lex-orders can be defined, with a B-Tree index built for each order. Subotić et al. (2018) use high-performance B-Tree indexes for their high parallelism and excellent trade-off of memory and speed. However, any uni-dimensional index with operational support for range queries would also be able to cover the same set of equality primitive searches.

2.3.5 Equality Primitive Search to Lex Search

To evaluate equality primitive searches with B-Tree indexes, Subotić et al. (2018) define the notion of a *lex search*. A lex-search is a range query on a uni-dimensional index. The authors detail how equality primitive searches can be translated to equivalent lex searches and which lex-orders can be used to cover certain equality primitive searches. A lex-search $\sigma_{\rho(\ell,a,b)}$ for a given relation $R \subseteq \mathcal{D}$ has semantics:

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

We denote $\rho(\ell, a, b)$ as a lex search predicate, ℓ as an index on R, a and b as lower and upper bounds to the search as tuples in $\mathcal{D}$. A range query can be evaluated by a lex-search by finding the lower and upper bounds a and b in the index and then iterating through all tuples between these bounds.

Given an equality primitive search $\sigma_{x_1=v_1,...,x_k=v_k}$, Subotić et al. (2018) translate it into an equivalent lex-search $\sigma_{\rho(\ell,a,b)}$ by finding a, b and ℓ for a relation R of m attributes. Trivially, if $k = m$ then all attributes are defined in the search predicate and we set $a = b = \langle v_1, ..., v_m \rangle$. For searches where some attributes are not specified in the search predicate then infima and suprema values are used to pad a and b respectively.

Specifically, Subotić et al. (2018) define an index mapping function from the index of an attribute of the relation to the index of the corresponding constrained attribute:

$$i : \{1, ..., m\} \to \{1, ..., k+1\}$$

If an attribute has a constraint in the search predicate, then it will map to itself. Otherwise, it will map to index $k + 1$. v_{k+1} is defined to be Δ, an artificial value used to represent an unspecified value where Δ does not belong to any given domain $\mathbb{D}_i$. The constant Δ signifies that the value requires padding. When padding Δ, the value will become an infimum in the lower bound or a supremum in the upper bound.

When constructing the bounds $a = lb(v_1, ..., v_k) = \langle v'_1, ..., v'_k \rangle$ where:

$$v'_j = \left\{ \begin{array}{ll} v_{i_j}, & \text{if } v_{i_j} \neq \Delta \\ \inf(\mathbb{D}_j), & \text{otherwise} \end{array} \right\}$$

and $b = ub(v_1, ..., v_k) = \langle v''_1, ..., v''_k \rangle$ where:

$$v''_j = \left\{ \begin{array}{ll} v_{i_j}, & \text{if } v_{i_j} \neq \Delta \\ \sup(\mathbb{D}_j), & \text{otherwise} \end{array} \right\}$$

2.3.6 Covering Primitive Searches with Lex Searches

After establishing how to construct a and b given an equality primitive search, Subotić et al. (2018) then classify which indexes ℓ can cover the equality primitive search. The k^{th}-prefix of an attribute sequence is formalised by the definition of a prefix set. For an attribute sequence (an index) $\ell = x_1 \prec x_2 \prec ... \prec x_m$, the k^{th}-prefix is $\{x_1, x_2, ..., x_k\}$ if $k \leq m$ and $\{x_1, x_2, ..., x_m\}$ otherwise. Subotić et al. (2018) prove that given $a = lb(v_1, ..., v_k), b = ub(v_1, ..., v_k)$ and ℓ has a k^{th}-prefix $\{x_1, ..., x_k\}$ then for all $R \subseteq \mathcal{D}$:

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

.

In effect, an equality primitive search can be covered by an index when its k^{th}-prefix coincides with the set of constrained attributes in the equality primitive search. Intuitively, we only need a k^{th}-prefix of the index to match the equality primitive search since the ordering of attributes that are not in the search are irrelevant.

For example, consider a relation $R(x, y) = \{\langle 2, 2\rangle, \langle 2, 3\rangle, \langle 3, 2\rangle\}$ and the equality primitive search $\sigma_{x=2}(R)$. We know from the above result that the equality primitive search can be converted to a lex-search if $\{x\}$ is a k^{th}-prefix of ℓ. This leaves us with only two choices for an index: $\ell = x$ or $\ell = x \prec y$ to speed up the equality primitive search. In fact, it is easy to show that the remaining indexes $y \prec x$ and y would not speed up this equality primitive search. Take for example $\ell = y \prec x$ where there is no k^{th}-prefix of ℓ that contains all the constrained attributes in the equality primitive search. $R(x, y) = \{\langle 2, 2\rangle, \langle 3, 2\rangle, \langle 2, 3\rangle\}$ shows the ordering of the tuples when using this lex-order. However, with this lex-order the first and third tuples satisfy the equality primitive search $\sigma_{x=2}(R) = \{\langle 2, 2\rangle, \langle 2, 3\rangle\}$. Thus no lex-search with bounds a and b can be constructed to capture the entire range of satisfying tuples. Given this, Subotić et al. (2018) strengthen the statement about which indexes can cover equality primitive searches: An index ℓ covers an equality primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$ for all $R \subseteq \mathcal{D}$ **if and only if** it is a k^{th}-prefix of ℓ.

Given these restrictions on which indexes may cover equality primitive searches, it can frequently be the case that multiple indexes are required to cover every search for a relation. The disadvantage is that maintaining extra indexes incurs a memory overhead and evaluation time overhead as for every insertion or deletion operation on the relation, every index must be updated to keep their states consistent. However, all equality primitive searches must be covered by an index or the evaluation performance degrades dramatically as searches are evaluated naïvely as table scan and filter operations. Therefore, a problem of great interest is to determine the minimum cardinality index selection which still covers every equality primitive search.

2.3.7 Minimum Index Selection Problem (MISP)

The Minimum Index Selection Problem (MISP). Given a set of searches $\mathcal{S}$ where $S \in \mathcal{S}$ is the set of attributes appearing in the search predicate of an equality primitive search on a relation R, the minimum index selection problem is to find the minimum cardinality set of uni-dimensional indexes $\mathcal{L}$ such that the index set covers all searches (Subotić et al., 2018).

We note that the problem formulation uses the notion of a search set rather than an equality primitive search itself. Each equality primitive search maps to the set of constrained attributes in the search since the constant values in the predicate are unimportant i.e. $\sigma_{x=2,y=3,z=5} \mapsto \{x, y, z\}$. Subotić et al. (2018) prove that a brute-force approach to solving the MISP would have a time complexity of $\mathcal{O}(2^{m^m})$ where m is the arity (the number of attributes) of the relation R. They propose a polynomial time algorithm to solve the MISP instead via a reduction to another problem, the Minimum Chain Covering Problem (MCCP).

The Minimum Chain Covering Problem (MCCP). Given a set of searches $\mathcal{S}$ where $S \in \mathcal{S}$ is the set of attributes appearing in the search predicate of an equality primitive search on a relation R, the minimum chain covering problem is to find the minimum cardinality set of search chains $\mathcal{C}$ where $C \in \mathcal{C}$ where $C = S_1 \subset S_2 \subset ...S_k$ such that the set of search chains covers all searches (Subotić et al., 2018).

The key idea in the reduction by Subotić et al. (2018) is establishing a one-to-one correspondence between indexes and search chains. At a high level, since a search chain is comprised of a set of searches that are all subsets of each other then by definition, the k^{th}-prefixes of $\ell = S_1 \prec (S_2 - S_1) \prec (S_3 - S_2) \prec ... \prec (S_k - S_{k-1})$ all coincide with the search sets to be covered. To prove the claim in the other direction, any index $\ell = S_1 \prec (S_2 - S1) \prec (S_3 - S_2) \prec ... \prec (S_k - S_{k-1})$ that covers the set of searches maps directly to the search chain $\mathcal{C} = S_1 \subset S_2 \subset ...S_k$, proving the equivalence.

Therefore, the only task left is solving the corresponding MCCP which (Subotić et al., 2018) solve using Dilworth's theorem. Dilworth's theorem (Dilworth, 2009) states that the number of chains required for a minimum chain covering over an associated lattice is equal to the cardinality of the largest anti-chain (a set of elements in the lattice such that no two elements in the set are comparable). Although, Dilworth's theorem does not provide an algorithm to solve for a minimum chain cover directly, others have since developed algorithms that solve for a minimum chain cover in polynomial time (Fulkerson, 1956). Specifically, by solving an associated maximum matching problem on a bipartite graph, the minimum chain cover can be computed. The algorithms used by Subotić et al. (2018) are below:

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

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

$\mathcal{M} \leftarrow$ MaximumMatching $(\mathcal{S}, \mathcal{S}, \{(S, S') \in \mathcal{S} \times \mathcal{S} \mid S \subset S'\})$
Initialize $\mathcal{C}$ to be the empty set
for all $u_1 \in \mathcal{S}$ s.t. $\nexists(u_0, u_1) \in \mathcal{M}$ do-
    Find maximal path $(u_1, u_2), (u_2, u_3), \ldots, (u_{k-1}, u_k) \subseteq \mathcal{M}$
      Add $u_1 \subset u_2 \subset u_3 \subset \cdots \subset u_{k-1} \subset u_k$ to $\mathcal{C}$
return $\mathcal{C}$
```

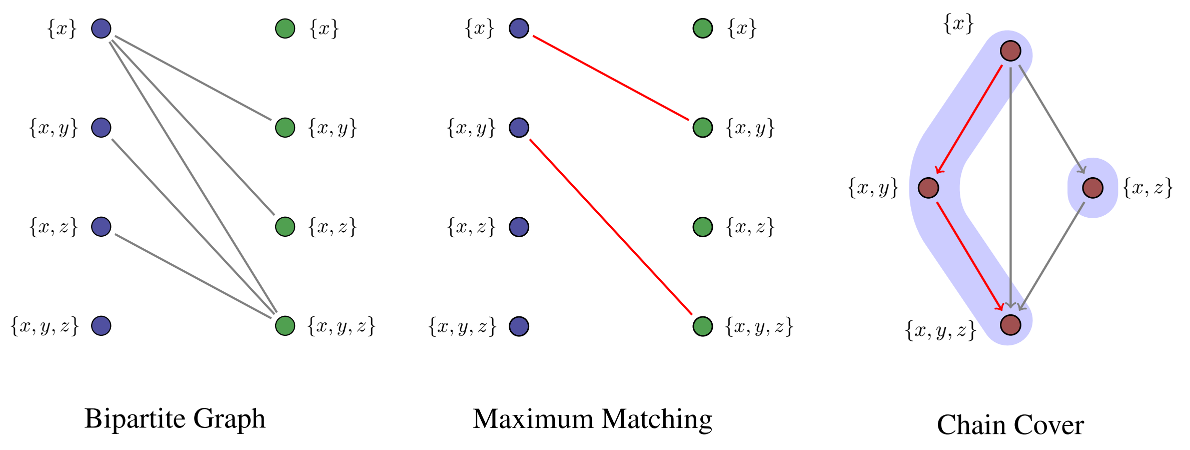
Firstly, all equality primitive searches are converted to search sets. Next, a minimum chain cover is computed over the search sets by computing the maximum cardinality matching over the bipartite graph. Each of the maximal paths in the maximum matching corresponds to a chain in the minimum chain covering on the lattice. The construction of the bipartite graph takes $\mathcal{O}(|\mathcal{S}|^2 \cdot m)$ since for each pair of searches, it needs to be computed whether one search is a subset of the other. The complexity of the bipartite matching is $\mathcal{O}(|\mathcal{S}|^{2.5})$.

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

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

$\mathcal{C} \leftarrow$ MinChainCover($\mathcal{S}$)
Initialize $\mathcal{L}$ to be the empty set
for all $S_1 \subset S_2 \subset \cdots \subset S_{k-1} \subset S_k \in \mathcal{C}$ do-
    Add to $\mathcal{L}$ an arbitrary index conforming with
      $S_1 \prec (S_2 - S_1) \prec \ldots \prec (S_k - S_{k-1})$
return $\mathcal{L}$
```

Next once the minimum chain cover is computed, each chain is transformed into a correspond index that covers the same set of searches. After transforming all chains to indexes, the index set is returned. Overall, transforming the chains to indexes is $\mathcal{O}(|\mathcal{S}| \cdot m)$. Therefore, the overall time complexity of the index selection technique is $\mathcal{O}(|\mathcal{S}|^2 \cdot m + |\mathcal{S}|^{2.5})$.



The above figure illustrates the process of solving the minimum chain covering problem:

(1) A bipartite graph is constructed from the Cartesian product of the search sets, drawing an edge from a node in the left bi-partition to a node in the right bi-partition if it is a proper subset.
(2) A maximum cardinality matching is computed over the associated bipartite graph.
(3) The paths in the maximum matching are mapped back to search chains in the lattice.

Although, it isn't shown in the diagram, this instance would results in the index set $\mathcal{L} = \{\ell_1, \ell_2\}$ where $\ell_1 = x \prec y \prec z$ and either $\ell_2 = x \prec z$ or $\ell_2 = z \prec x$. Therefore for this instance, 4 searches can be covered by only 2 B-Tree indexes using lex-orders ℓ_1 and ℓ_2.

2.3.8 Implementation in SOUFFLÉ

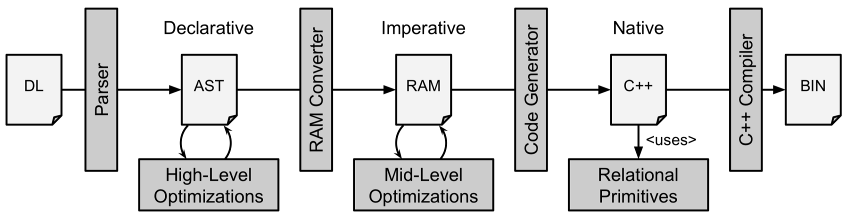


FIGURE 2.10. Transformation Pipeline of SOUFFLÉ (Scholz et al., 2016)

SOUFFLÉ has integrated this state-of-the-art index selection technique with great benefit. Figure 2.10 details the transformation process within the Datalog engine. After transforming the program from Datalog to AST to RAM, an analysis is performed on all loop nests, collecting equality primitive searches for each relation. Search sets are built from these searches and passed as input to the index selection algorithm. The code generation step from RAM to C++ builds the appropriate indexes according to the output of the index selection algorithm.

When evaluated using real-world applications, the technique proved to be very successful, evaluating programs up to $2\times$ faster and using up to $6\times$ less memory compared to existing index selection strategies. Furthermore, the increase in compilation time as a result of the index selection algorithm was negligible, contributing less than 1% to the overall compilation time (Subotić et al., 2018). Overall, for equality primitive searches the state-of-the-art technique elevates Datalog evaluation to a whole new level, achieving performance on-par and sometimes even surpassing the performance of hand-crafted programs (Subotić et al., 2018). However, the technique has a crucial limitation that indexes are constructed only to cover equality primitive searches. For searches with inequality constraints i.e. *inequality primitive searches*, they are evaluated without an index, heavily degrading the evaluation time of Datalog programs.

2.4 Inequality Primitive Searches in Datalog Engines

We now review the existing work to efficiently evaluate inequality primitive searches in Datalog. Campagna et al. (2012) outline a program rewriting transformation used to speed up inequality primitive searches, evaluating the merits of their transformation in the popular LogicBlox engine. Their technique is a source-to-source transformation that borrows ideas from constraint programming. They use a technique known as *constraint propagators*, to filter relations appearing in multi-way loop-nests.

Instead of evaluating the loop-nest by naïvely generating the Cartesian product of all involved relations and filtering on the inequality predicates, inequality predicates prune the relation before the product is enumerated. Extra relations denoted as *filter relations* are substituted in place of the original relations when using inequality constraints on the relation within the rule. Each filter relation is a duplicate of the original, applying the inequality constraints on the filter relation. In the ideal case, filter relations will be much smaller than the original relation, pruning the relation when inequality predicates are very selective. When the rules fire, these filter relations appear in the Cartesian product, which may significantly prune the search space.

When filtering over relations, inequality predicates may use values from attributes of other relations in the rule. In the filter relation, these values are not made available, and instead, the relation uses the extreme values of the abstract domain of the relevant attribute. An aggregate query for the minimum and maximum values of the relevant attributes are computed and used to replace the lower and upper bounds of an inequality constraint, respectively. In effect, by considering only the infima and suprema values in the abstract domain, the computation of each filter relation will never exceed $\mathcal{O}(n)$ time to compute where n is the size of the relation as the Cartesian product is computed in the original rule.

$$\begin{aligned} a(x, y) \coloneq\, & b(x), \\ & c(y), \\ & y < x. \end{aligned}$$

FIGURE 2.11. Naive Loop Nest

$$\begin{aligned} c\_filtered(y) \coloneq\, & highest\_b = max\ \ x:\ \ b(x)\ \ , \\ & c(y), \\ & y < highest\_b. \end{aligned}$$

FIGURE 2.12. Filter Relation on c to Prune Search Space

$$\begin{aligned} a(x, y) \coloneq\, & b(x), \\ & c\_filtered(y), \\ & y < x. \end{aligned}$$

FIGURE 2.13. Transformed Loop Nest with Filter Relation

The above figure illustrates the transformation technique. In the naïve inequality join, the Cartesian product of b and c is enumerated and then filtered for the values satisfying the inequality predicate. In the transformed program, a filter relation for c is materialised, which filters the relation c for values that are smaller than the maximum value of b. If c contains many values that are larger than b's maximum value, then this will filter the relation significantly. Then when evaluating the rule, the filtered relation $c\_filtered$ replaces c, reducing the size of the Cartesian product with b. Note that the inequality constraint is still present in the rule even after replacing relations with their filtered counterparts. The constraint is kept because filtering using extreme values of the abstract domain does not guarantee that it will satisfy the original inequality predicate. Overall, this technique only helps to prune the search space but cannot guarantee speedups in the general case as the filter relations may not reduce the size of the relation much or at all.

The transformation above only demonstrates the procedure for non-recursive rules. The authors employ an approximation technique when dealing with recursive rules in order to avoid unstratified aggregation. The lower and upper bounds are approximated, using the deriving rule bodies for the relation, avoiding aggregation on the relation directly.

The technique demonstrates substantial speedups of up to $8\times$ for synthesised benchmarks however it is not suited for workloads typical of modern Datalog engines. Modern application domains of Datalog such as static program analysis and security analysis rely on the fast evaluation of large numbers of mutually recursive rules (Subotić et al., 2018). Therefore rough approximations would have to be used that would fail to prune the search space of relations significantly.

$$
\begin{aligned}
&//\ Range\ Conflict\\
&data\_object\_conflict(EA_1, Size_1, Type_1, EA_2, Size_2, Type_2) :-\\
&\qquad\qquad data\_object\_candidate(EA_1, Size_1, Type_1),\\
&\qquad\qquad data\_object\_candidate(EA_2, Size_2, Type_2),\\
&\qquad\qquad EA_2 > EA_1,\\
&\qquad\qquad EA_2 < EA_1 + Size_1.
\end{aligned}
$$

FIGURE 2.14. Most Time Consuming Datalog Rule in DDISASM

Consider, for example, the above Datalog rule in DDISASM, consuming as much as 58% of the total evaluation time. DDISASM represents a typical application for modern Datalog engines which rely heavily on searches with inequality primitive searches. Since both atoms in the rule refer to the same relation, the abstract domain of each atom is the same. Since the technique relies on using the abstract domain of atoms earlier in the rule to prune the search space, no pruning will occur in the filter relation, and the technique will not speed up this rule at all. Furthermore, since the technique requires building filter relations, the performance will degrade as maintenance must be done for the extra indexes.

By contrast, if an index were to cover the relation, then the inequality constraints could be leveraged to speed up the evaluation of the search corresponding to the second atom. Therefore, to speed up rules with inequality primitive searches for modern Datalog applications, we cannot rely on program-level transformations and instead look to speeding up their evaluation with indexes.

CHAPTER 3

Spatial Primitive Searches

In this chapter, we present our two new index selection strategies for accelerating inequality primitive searches. We begin by detailing the transformation process taken by Datalog engines to translate rules in logic programs to loop nests. We detail how to translate and hoist inequality constraints that appear in rules into predicates that can be leveraged by indexed searches. Next, we define a gadget called a *spatial primitive search* which abstracts indexable search operations on Datalog relations from the underlying choice of the index used to evaluate them. We go on to prove an equivalence between the semantics of spatial primitive searches and orthogonal range queries. We explore multi-dimensional indexes which can evaluate orthogonal range queries efficiently in a Datalog setting and propose a new index selection strategy to accelerate spatial primitive searches with R-Trees. We also extend the state-of-the-art index selection strategy in a different direction, accelerating all spatial primitive searches for a given relation with a minimal cluster of B-Tree indexes. We present a constructive algorithm to compute this minimal index selection of B-Trees and prove its correctness.

3.1 Leveraging Inequality Constraints in Loop Nests

3.1.1 Datalog to Nested Loop Searches

First, we will outline the translation process from a Datalog program to a series of loop nests. For readability, I will use the notation of SOUFFLÉ's intermediate representation the Relational Algebra Machine (RAM) language.

$$\begin{aligned}&\mathit{direct\_jump}(EA, Dest) :- \\ &\quad \mathit{jump\_instruction}(EA), \\ &\quad \mathit{instruction\_get\_op}(EA, 1, Op_1), \\ &\quad \mathit{op\_immediate}(Op_1, Dest).\end{aligned}$$

FIGURE 3.1. Datalog Rule

$$\begin{aligned}&\mathit{direct\_jump}(x_0, x_5) :- \\ &\quad \mathit{jump\_instruction}(x_0), \\ &\quad \mathit{instruction\_get\_op}(x_1, x_2, x_3), \\ &\quad \mathit{op\_immediate}(x_4, x_5), \\ &\quad x_1 = x_0, x_2 = 1, x_4 = x_3.\end{aligned}$$

FIGURE 3.2. Transformed Rule

The first step in the translation process is shown in the above figures. Each named attribute in an atom is assigned a unique identifier $x_0, x_1, ..., x_k$. If the name of an attribute in one atom is referenced again in subsequent atoms in the rule then equality constraints are introduced for the subsequent atoms, enforcing that the value of their corresponding attributes must match the one in the first atom. Since there is no dependence between any of the referenced attributes in the atoms in the rule, it can now be safely unrolled into a loop nest with equality constraints used in filter operations.

```
for all $t_0 \in \mathit{jump\_instruction}$ do
  for all $t_1 \in \mathit{instruction\_get\_op}$ do
    for all $t_2 \in \mathit{op\_immediate}$ do
      if $t_1(x_1) = t_0(x_0)$ and $t_1(x_2) = 1$ and $t_2(x_4) = t_1(x_3)$ do
        if $(t_0(x_0), t_2(x_5)) \notin \mathit{direct\_jump}$ do
          project $(t_0(x_0), t_2(x_5))$ into $\mathit{direct\_jump}$
```

FIGURE 3.3. Direct Translation of Datalog Rule to Loop Nest

As shown in the above figure, the direct translation begins by generating the Cartesian product over the underlying relations of all atoms. The filter predicate is a conjunction of all equality constraints in the rule and is checked against all generated tuples in the Cartesian product. If the predicate is satisfied, an existence check is performed on the output tuple to ensure the tuples of a relation are unique and satisfy the set property. Finally, if the output tuple does not already exist in the relation it is inserted into the relation.

This naïve unrolling of the Datalog rule into the loop nest is inefficient. The time complexity of the loop nest will be of the order of the Cartesian product of each relation involved in the rule i.e. $\mathcal{O}(\prod_i |R_i|)$ where R_i is the relation used in atom A_i of the rule body. For this example, the complexity is $\mathcal{O}(|\mathit{jump\_instruction}| \cdot |\mathit{instruction\_get\_op}| \cdot |\mathit{op\_immediate}|)$. The source of this inefficiency is that the searches are evaluated in a generate and test approach rather than pruning the search space as early as possible. To achieve this, constraints in the filter predicate can be lifted and utilised as search criteria for indexed searches over the relations.

3.1.2 Indexing Nested Loop Searches

The next step in the translation process is to iteratively hoist the filter predicates into the search operations in the outer levels of the loop nest. A filter predicate can be hoisted into a search operation if the filter acts on an attribute of that relation. If a predicate is hoisted it transforms a regular table scan into an indexed search over the relation. An indexed search is strengthened further by lifting additional filter predicates into the search, restricting the range of satisfying tuples. The process repeats until reaching a fix-point when no more filters can be brought higher in the loop nest.

```
for all $t_0 \in \mathit{jump\_instruction}$ do
    for all $t_1 \in \mathit{instruction\_get\_op}$ on index $t_1(x_1) = t_0(x_0)$ and $t_1(x_2) = 1$ do
        for all $t_2 \in \mathit{op\_immediate}$ on index $t_2(x_4) = t_1(x_3)$ do
            if $(t_0(x_0), t_2(x_5)) \notin \mathit{direct\_jump}$ do
                project $(t_0(x_0), t_2(x_5))$ into $\mathit{direct\_jump}$
```

FIGURE 3.4. Strengthened Loop Nest using Indexed Searches

In the figure above, we can see predicates in the filter operation raised to higher levels of loop nest for indexed search operations. Note that a tuple can appear on either the left-hand side or the right side of a predicate. However, it can only be hoisted to a given level of the loop nest if the other side of the constraint is a constant. For instance, $t_1(x_1) = t_0(x_0)$ could not be hoisted to the outer-most layer because $t_1(x_1)$ would not be grounded.

For simplicity, the only predicates that can be lifted and employed in indexed operations are those of the form $t_i(x_k) = \langle expr \rangle$ or $\langle expr \rangle = t_i(x_k)$ where $\langle expr \rangle$ is a constant at the level of t_i in the loop-nest. Since predicates must be of this exact form, seemingly simple predicates such as $t_1(x_2) - 1 = 0$ may not be hoisted and leveraged by indexed searches. Therefore, the robustness of rewriting transformations in the Datalog engine determines the range of predicates that can prune the search space during rule evaluation.

This transformation technique proves to be incredibly powerful, eagerly pruning the search space of satisfying tuples as each relation is searched for a much smaller set of tuples required by the rule. Using a B-Tree to index each relation yields a complexity of a given indexed search as $\mathcal{O}(|\sigma_\rho(R_i)| + \log(|R_i|))$

where ρ is the filter predicate used by the indexed search. Therefore, the total complexity of evaluating the loop nest becomes $\mathcal{O}(\prod\limits_{i} |\sigma_\rho(R_i)| + \log(|R_i|))$. These search predicates dramatically reduce the search space of satisfying tuples since the resultant product is often orders of magnitude smaller than the Cartesian product across all involved relations, leading to significant speed ups in evaluation time.

3.2 Indexing Inequality Searches

3.2.1 Normalising Constraints to Weak Inequalities

(1) $t_i(x_k) < \langle expr \rangle$
(2) $t_i(x_k) > \langle expr \rangle$
(3) $t_i(x_k) \leq \langle expr \rangle$
(4) $t_i(x_k) \geq \langle expr \rangle$

Inequality constraints can appear in any of the four distinct forms shown above. Note that another form is possible where expressions appear on the left-hand side and constraints appear on the right-hand side of the predicate. However, by swapping the left and right-hand sides of the constraint and reversing the direction of the inequality, the expression can match the above form. Typically, data structures accept lower and upper bounds for range queries where satisfying values may be equal to either bound. The first two classes of constraints with strict inequalities, therefore, pose a problem and must be converted to weak inequalities. To use data structures for indexed searches on inequality constraints, we consider a rewrite transformation to translate any strict inequality into a weak inequality.

Consider signed integer types, the most commonly used type in Datalog engines. Typically a fixed size is given for the type, i.e. 32 bits, and the range is divided into the interval $[-2^{31}-1, 2^{31}]$. A naïve approach to transform a strict inequality constraint into a weak inequality would be to increment or decrement a value by 1 as required, converting a strict inequality to a weak inequality. However, this approach leads to complications. A constraint may attain the maximum or minimum value of a domain and adding or subtracting the value 1 may result in an integer overflow. Furthermore, for more sophisticated types such as floating-point numbers, attaining the next highest or lowest representable value introduces its own set of problems. However, we cannot merely leave strict inequalities as naïve filter operations as this would place the burden on the user to rewrite every strict inequality constraint into a weak inequality to use the predicate in an indexed search. Therefore, to translate strict inequalities to weak inequalities, we perform the following rewrite transformations:

$$t_i(x_k) < \langle expr \rangle \longrightarrow t_i(x_k) \leq \langle expr \rangle \wedge t_i(x_k) \neq \langle expr \rangle$$

$$t_i(x_k) > \langle expr \rangle \longrightarrow t_i(x_k) \geq \langle expr \rangle \wedge t_i(x_k) \neq \langle expr \rangle$$

This approach widens the range of the indexed search by one value and requires the application of an extra filter predicate on every iteration of the loop nest. However, this cost is negligible compared to the potential performance gains from leveraging the inequality constraint in an indexed search. Ultimately, we ensure that indexed operations maintain semantic correctness and are used automatically for a range of primitive types (float, unsigned and signed numbers) without burdening the user with extensive manual rewriting of strict inequalities.

3.2.2 Strengthening Filter Constraints

In the same manner, as with equality constraints, we hoist inequality constraints from filters deep in the loop nest to outer layers to be used for indexed searches. When lower and upper bounds constrain a given attribute, both bounds can be utilised by the indexed search, performing a two-sided range query on the relation. However, care is required when multiple lower or upper bounds are present for the same attribute. Consider the following simplified rule from DDISASM (Flores-Montoya and Schulte, 2020):

$$\begin{aligned}
&valid\_multiplier(Mult, Size) \text{ :- } \\
&\quad data(Size), \\
&\quad multiplier(Mult), \\
&\quad Mult \geq 1, Mult \geq Size.
\end{aligned}$$

FIGURE 3.5. Datalog Rule with Two Lower Bounds

In the above Datalog Rule, the multiplier atom has two provided lower bounds, 1 and $Size$. To accelerate this search, we wish to perform an indexed search by raising the lower bounds from the body of the loop nest upwards as we did before with equality constraints. However, when performing a range query on an index, only a single lower bound for an attribute can be provided. A naïve approach would be to arbitrarily raise one of the filter predicates and use it for the index search, leaving the remaining predicate intact in a filter operation. However, a larger lower bound will restrict the range of satisfying tuples further than a smaller one. The challenge lies in that before evaluating the rule, it is unknown which lower bound is greatest.

```
for all $t_0 \in data$ do
  for all $t_1 \in multiplier$ on index $t_1(x_0) \geq max(1, t_0(x_0))$ do
    if $(t_1(x_0), t_0(x_0)) \notin valid\_multiplier$ do
      project $(t_1(x_0), t_0(x_0))$ into $valid\_multiplier$
```

FIGURE 3.6. Loop Nest with Indexed Searches

Therefore, as the filter predicates are hoisted up and used to perform an indexed search on $multiplier$, the loop nest is rewritten so that the maximum of the two lower bounds comprises the lower bound of the search. An analogous technique is employed for when multiple upper bounds appear for the same attribute, considering the minimum of the two upper bounds when forming the upper bound of the indexed search. The transformation ensures that the tightest bound is used for the indexed search and prunes the search space as much as possible.

In practice, an attribute should only be searched once over a predicate specifying the range of satisfying values. When the range is tight, the predicate forms an equality constraint. Otherwise, inequalities bound the values that are attainable. However, when multiple equality constraints or both equality and inequality constraints are present for the same attribute, then it should be handled correctly. Consider the following program:

$$\begin{aligned} &valid\_multiplier(Mult, Size) :- \\ &\quad data(Size), \\ &\quad multiplier(Mult), \\ &\quad Mult = 1, Mult = Size. \end{aligned}$$

FIGURE 3.7. Datalog Rule with Two Equality Constraints

Both equality constraints must be satisfied for the body of the rule to hold. However, for an indexed search, only one predicate can be used. Therefore, the technique we propose is to lift the first filter predicate on the attribute into the indexed search. For all subsequent equality constraints on that attribute, we issue a new filter predicate in the loop-nest to check whether the new constraints match the original. This transformation is still able to use the filter predicates to perform an index search yet retains the semantics of the original Datalog program.

```
for all $t_0 \in data$ do
  for all $t_1 \in multiplier$ on index $t_1(x_0) = 1$ do
    if $(t_1(x_0), t_0(x_0)) \notin valid\_multiplier$ do
      project $(t_1(x_0), t_0(x_0))$ into $valid\_multiplier$
```

FIGURE 3.8. Indexed Searches

Note that further transformations take place that lift filter predicates to appear as early in the loop-nest as possible. By lifting filter predicates upwards to the outer layers, when the predicates are not satisfied, the loop-nest is exited earlier, eliminating redundant iterations of the Cartesian product. For instance, the above loop nest can transform to:

```
for all $t_0 \in data$ do
  if $t_0(x_0) \neq 1$ do
    for all $t_1 \in multiplier$ on index $t_1(x_0) = 1$ do
      if $(t_1(x_0), t_0(x_0)) \notin valid\_multiplier$ do
        project $(t_1(x_0), t_0(x_0))$ into $valid\_multiplier$
```

FIGURE 3.9. Filters Lifted Above Indexed Searches

Since $t_0(x_0) \neq 1$ relies only on the outermost loop to ground $t_0(x_0)$ it can be lifted to this position in the loop nest. After the above transformation, if multiple equality constraints on the same attribute have different values, then the indexed search is avoided entirely. This transformation can always be applied when any of the equality constraints are usable for an indexed search. The reason for this is that used equality constraints must appear grounded, either as constants or tuple values from above the current level in the loop nest. Therefore, the filter predicate that checks whether multiple equality constraints coincide is always possible, raising the filter predicate above the indexed search at one level higher in the loop nest.

Finally, we consider the case when both equality and inequality constraints apply to an attribute. We observe that equality constraints are a specialisation of inequality constraints where the lower and upper bounds are equal. Therefore, generally, an equality constraint is more restrictive than an inequality constraint and should be preferred for use in an indexed search. However, as with multiple equality constraints, the inequality constraint must also be satisfied for the rule body to hold. Therefore, we perform the following transformation:

```
$valid\_multiplier(Mult, Size)$ :–
    $data(Size),$
    $multiplier(Mult),$
    $Mult = 1, Mult \geq Size.$
```

FIGURE 3.10. Datalog Rule with Equality and Inequality Constraints

```
for all $t_0 \in data$ do
  if $1 \geq t_0(x_0)$ do
    for all $t_1 \in multiplier$ on index $t_1(x_0) = 1$ do
      if $(t_1(x_0), t_0(x_0)) \notin valid\_multiplier$ do
        project $(t_1(x_0), t_0(x_0))$ into $valid\_multiplier$
```

FIGURE 3.11. Filter Predicates Hoisted to Indexed Searches

As discussed, the indexed search always use the equality constraint. Furthermore, as with the transformation of multiple equality predicates, the values for the constraints are compared. Specifically, a check is performed on the value of the equality constraint to ensure that it satisfies the inequality constraint. This strategy allows for the indexed search to be the most effective at pruning the search space by using the most restrictive predicate and exiting early from the loop nest if the inequality constraints are not satisfied.

3.3 Abstracting Search Operations as Spatial Primitive Searches

We have demonstrated how logic programs are translated into equivalent imperative programs by transformation to a loop nest. Crucial to the performance of the loop nest evaluation is the performance of searches on these relations that can be accelerated by indexes. There are a wealth of options for indexes to accelerate search operations such as hash-indexes, B-Trees or R-Trees. However, in essence, each search operation is semantically just a filter on a Datalog relation, reducing all tuples of a relation to only those satisfying some constraints. Therefore, we can abstract these search operations as *spatial primitive searches*, focusing only on the semantics of these searches without concern for the concrete choice of index chosen to evaluate them.

We define a spatial primitive search as follows:

$$\sigma_{l_1 \leq x_1 \leq u_1, \ldots, l_k \leq x_k \leq u_k}(R_i) = \{t \in R_i \mid l_1 \leq t(x_1) \leq u_1, \ldots, l_k \leq t(x_k) \leq u_k\}.$$

We borrow the notation of an equality primitive search except now we introduce $L = \{l_1, l_2, ..., l_k\}$ and $U = \{u_1, u_2, ..., u_k\}$ to represent the lower and upper bounds for each search predicate. All elements of L and U are values which can be constants or values of tuples above the current level of the loop nest.

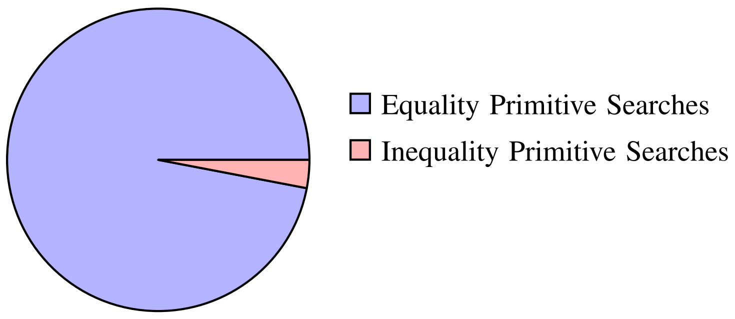


FIGURE 3.12. Classes of Spatial Primitive Searches

Spatial primitive searches generalise the notion of an equality primitive search which can only abstract equality primitive searches, to now support inequality primitive searches. For each attribute x_i, an equality primitive search can have constraints of the form $x_i = v_i$. With spatial primitive searches however, we have constraints of the form $l_i \leq x_i \leq u_i$. Thus, an equality primitive search is a specialisation of a spatial primitive search where for all constrained attributes x_i we have $l_i = u_i$.

Inequality primitive searches are now expressible as spatial primitive searches, where there exists at least one attribute x_i where its lower and upper bounds are not equal, i.e. $l_i \neq u_i$. For simplicity we write r_i to represent a range constraint on an attribute x_i i.e. $r_i = l_i \leq x_i \leq u_i$ where $l_i \neq u_i$. By our definition, every spatial primitive search is therefore either an equality primitive search or an inequality primitive search on a Datalog relation.

3.4 Orthogonal Range Querying

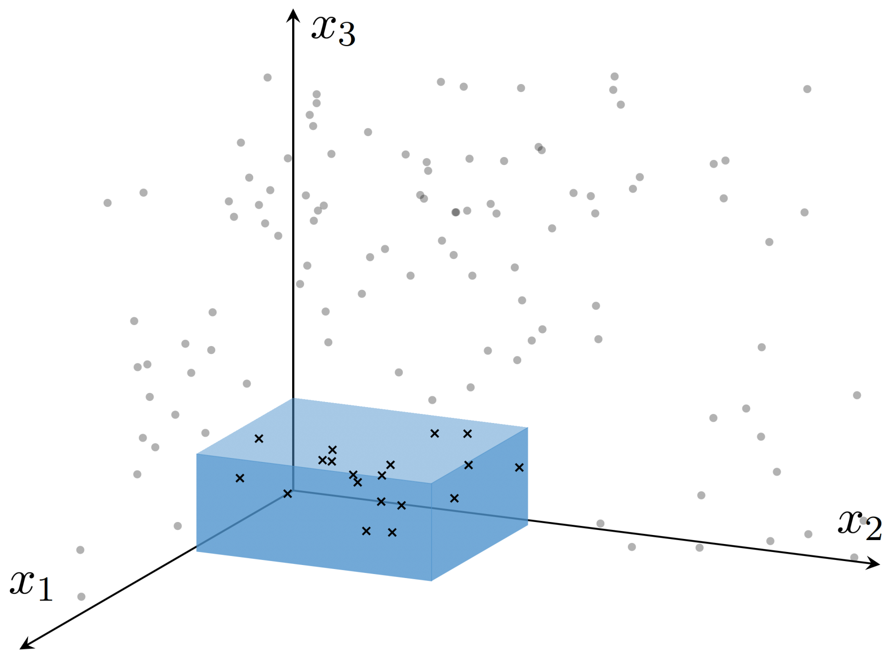


FIGURE 3.13. A Spatial Primitive Search Interpreted as an Orthogonal Range Query

Interestingly, spatial primitive searches are in effect a filter operation applied to a set of tuples in a relation. Figure 3.13 visualises in 3-dimensions the semantics of spatial primitive searches. The tuples of a relation with arity 3 correspond to points in 3-dimensional space, with each attribute mapping to a given coordinate axis. The lower and upper bounds on each attribute correspond to intervals in each dimension, geometrically defining the blue box. Finally, the tuples that satisfy the spatial primitive search (i.e. those that lie within the intervals constraining each attribute value) map to the points bounded by the blue box.

Orthogonal range querying (Lueker, 1978) is the problem of finding a subset Q of a set of points S in a d-dimensional space, that lie within a specified d-dimensional box. Clearly, there is a strong connection between the semantics of spatial primitive searches and orthogonal range queries. We now establish the correspondence and show how to transform spatial primitive searches into equivalent orthogonal range queries.

3.4.1 Evaluating Spatial Primitive Searches as Orthogonal Range Queries

In order to evaluate spatial primitive searches as orthogonal range queries, a formal mapping scheme must be constructed. We have given a relation R of arity m with k constrained attributes in the spatial primitive search: $\sigma_{l_1 \leq x_1 \leq u_1, ..., l_k \leq x_k \leq u_k}$. Define the set of tuples $T = \sigma_{l_1 \leq x_1 \leq u_1, ..., l_k \leq x_k \leq u_k}(R)$ i.e. all tuples in R that satisfy the spatial primitive search.

An instance of the orthogonal range querying problem requires a set of points S in d-dimensional space. We make the simplifying assumption that each attribute x_i only holds numeric values (i.e. signed integers, unsigned integers and floating point numbers). We can then set $d = m$ and map each tuple $t \in R(x_1, ..., x_m)$ to a point $p \in S$ using a bijective mapping function $f\colon R \mapsto S$ such that $p(x_i) = t(x_i)$ i.e. $t = \langle x_1, ..., x_m \rangle \mapsto p = \langle x_1, ..., x_m \rangle$.

The only remaining input to specify is the d-dimensional box that represents the query box. We specify the d-dimensional box geometrically by providing an interval $b_i = [lower_i, upper_i]$ for each dimension $i \in \{1, ..., d\}$. The d-dimensional box is then formed by taking the Cartesian product of all intervals over the provided dimensions i.e. $B = [lower_1, upper_1] \times ... \times [lower_d, upper_d]$. We want to construct the box such that it encloses precisely the points that satisfy the spatial primitive search. For a spatial primitive search $\sigma_{l_1 \leq x_1 \leq u_1, ..., l_k \leq x_k \leq u_k}$ we have lower bounds $L = \{l_1, l_2, ..., l_k\}$ and upper bounds $U = \{u_1, u_2, ..., u_k\}$. In the simplest case where $|L| = |U| = m$, i.e. a lower bound and upper bound constraint is provided for every dimension, then we simply set $b_i = [l_i, u_i]$ for all dimensions.

Suppose an attribute is not constrained by a lower bound in the spatial primitive search. In that case, we must introduce an artificial lower bound to create an interval for the orthogonal range querying problem. Therefore, we set $lower_i = \inf(\mathbb{D}_i)$ since any point must trivially satisfy this lower bound. Likewise we set $upper_i = \sup(\mathbb{D}_i)$ for when an attribute has no specified upper bound. In the case where an attribute is entirely unconstrained it's corresponding interval will be $b_i = [\inf(\mathbb{D}_i), \sup(\mathbb{D}_i)]$ which will be trivially satisfied by all points. Now we wish to prove that the semantics of the orthogonal range query coincide with the spatial primitive search. To do this, we prove that $T = f^{-1}(Q)$, i.e. the set of tuples satisfying the spatial primitive search, is the same as that of all points satisfying the orthogonal range query after mapping the points back to tuples. The proof can be found in Appendix A of this thesis.

3.4.2 Choice of Index for Orthogonal Range Querying in Datalog Engines

Now that we have demonstrated how orthogonal range queries can be employed to perform spatial primitive searches, we need to consider the which index should be used to perform each search. Although orthogonal range queries are well-studied, selecting the data structure that supports the best query time complexity will not suffice. In particular, in a Datalog engine, the workload on indexes is comprised of frequent additions of new tuples to indexes as rules deduce further knowledge. Therefore, a data structure that efficiently supports dynamic updates is required.

3.4.3 k-d-trees

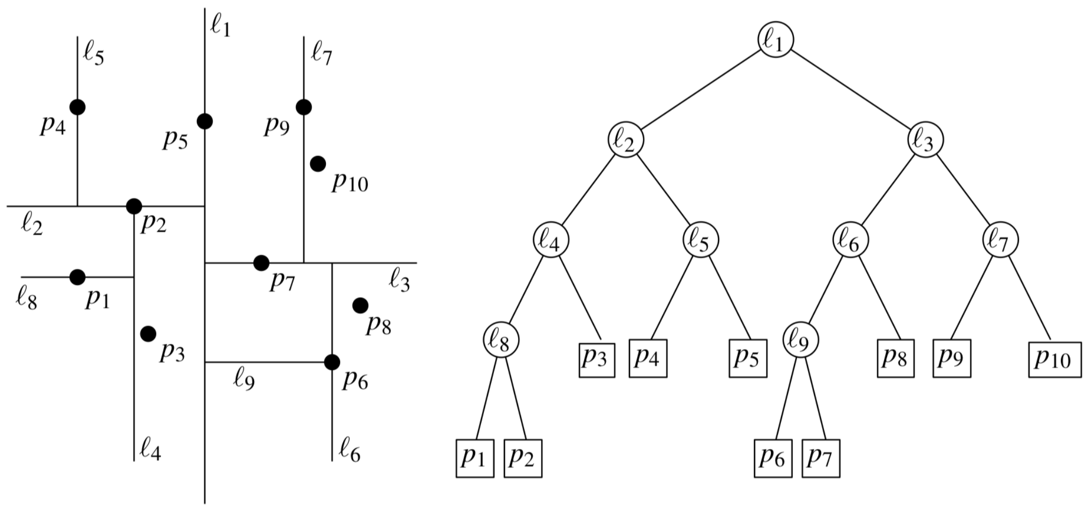


FIGURE 3.14. A k-d-tree cutting the plane into half-spaces with each non-leaf node (De Berg et al., 1997)

The first data structure designed to perform multidimensional searching was the k-d tree by Bentley (1975), an in-memory data structure to query multidimensional point data. Each non-leaf node of the k-d-tree cuts the entire plane into half-spaces where nodes in the left sub-tree lie on the left half-space and nodes in the right sub-tree lie in the other. When performing orthogonal range queries, analysis has shown that the worst-case time complexity is $\mathcal{O}(k \cdot N^{1-1/k})$ where k is the dimensionality of the k-d-tree, which is not very competitive compared to logarithmic query time expected of tree structures. Unfortunately, the main weakness of the k-d-tree is that it is not a balanced tree data structure. As a result, inserting new points will result in the tree becoming unbalanced, and query performance suffers. A naïve approach would be to rebuild the entire tree on each new insertion, but this is far too costly with a workload consisting primarily of updates. Variants of the k-d-tree called adaptive k-d-trees exist such as the KDB-tree (Robinson, 1981) and BKD-tree (Procopiuc et al., 2003) that re-balance themselves when sub-tree heights differ, guaranteeing consistent query performance. BKD-trees further improve on KDB-trees by utilising bulk-loading techniques but they are not applicable to our use case. However, k-d-trees and their variants are still not the most efficient data structures for performing orthogonal range queries.

3.4.4 Range Trees

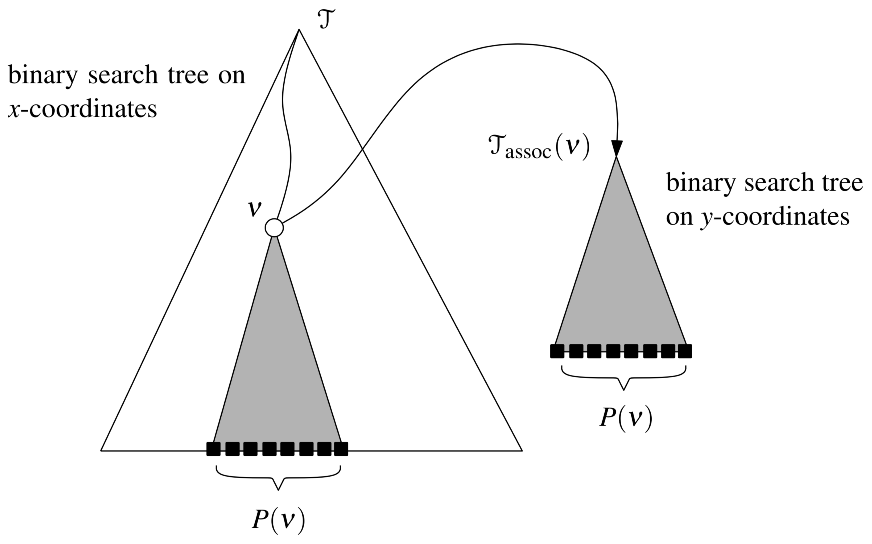


FIGURE 3.15. 2-Dimensional Range Tree (De Berg et al., 1997)

Bentley (1979) further improved on k-d-trees with the invention of range trees. Range trees are recursively defined multi-level search trees where each vertex has an associated $d-1$-dimensional structure to compute the range over the remaining dimensions. Range trees improve upon the query performance of k-d-trees and their variants by offering an impressive $\mathcal{O}(\log^k(n) + |Q|)$ where Q is the set of points satisfying the range query. This can be further improved using fractional cascading (Chazelle and Guibas, 1986) to $\mathcal{O}(\log^{k-1}(n) + |Q|)$. However, the range tree comes with a space trade-off where $\mathcal{O}(n \log^{k-1}(n))$ space is required compared to a traditional k-d-tree's $\mathcal{O}(n)$ space requirement. Range trees are promising since they have strong worst-case guarantees on their query performance; however, a dynamic variant is required if we wish to support frequent insertions during Datalog evaluation. Unfortunately, even though dynamic range tree variations exist that offer strong asymptotic range query performance, practically, dynamic range trees do not perform well as the algorithmic techniques rely on large constant factors (Agarwal et al., 1999).

3.4.5 R-Trees

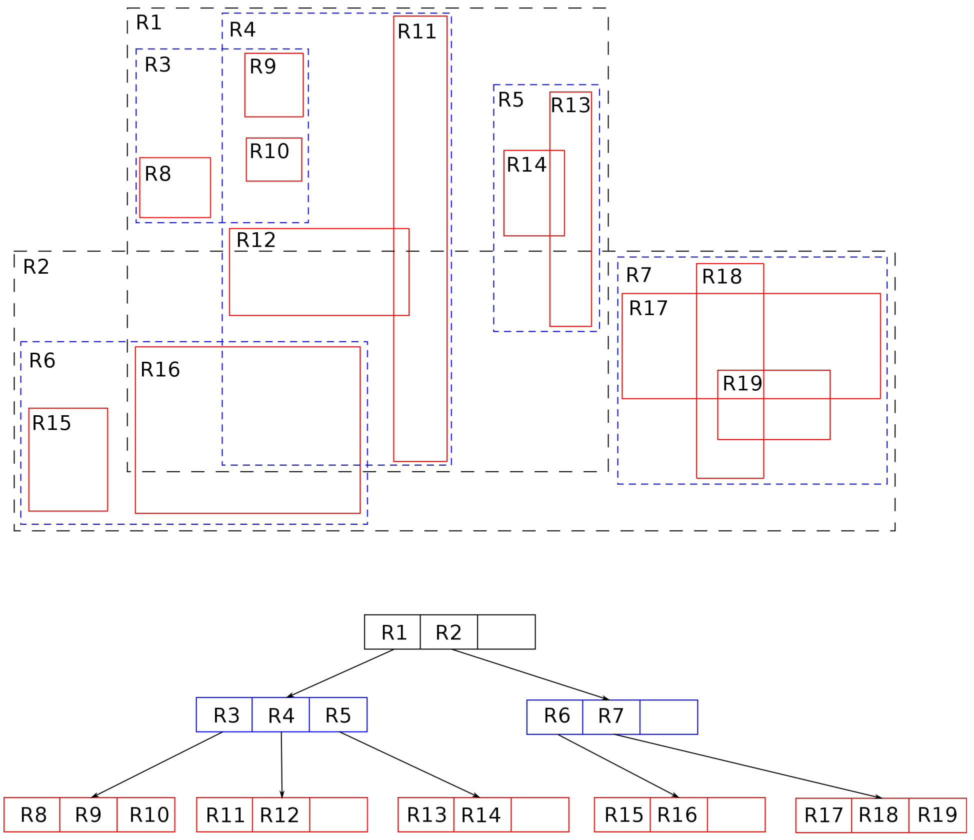


FIGURE 3.16. Tree Structure of the R-Tree and Minimum Bounding Rectangles (MBRs) of Indexed Spatial Data

Guttman (1984) published a groundbreaking paper detailing a new type of dynamic spatial index with support for efficient range queries. Much like a B-Tree, each node stores a collection of at most M and at least $M/2$ entries, ignoring the root node. Each entry appearing in a leaf-node has two components. Firstly, it stores the *minimum bounding rectangle* (MBR) of the spatial object and secondly, a pointer to the spatial object on disk. Entries in non-leaf nodes have their MBRs minimally enclose the MBRs of the entries in their child nodes. Since internal nodes must store the MBR enclosing the entries in their children and MBRs are represented as two corner points in n-dimensional space, the memory overhead of an R-Tree index can be approximately twice that of a B-Tree index.

R-Trees can support orthogonal range queries given a d-dimensional box (known as the query rectangle) by recursively traversing down each sub-tree of any node whose MBRs intersect with the query rectangle. The search continues until reaching the leaf layer and returning any entries in leaf-nodes that intersect with the query rectangle. Unlike k-d-trees which have strong guarantees on worst-case query complexity, R-trees may take $\mathcal{O}(N)$ time to perform a given range query where N is the number of spatial objects in the R-Tree. This worst-case behaviour occurs when there is a high degree of overlap in the MBRs of nodes at the same level. In the above figure, R1 and R2 overlap, so if a query rectangle were to intersect both R1 and R2, then both sub-trees would require exploration. Despite the poor guarantees regarding worst-case query complexity, R-trees have proven to provide impressive practical performance for efficient spatial indexing. In fact, R-trees are the de-facto spatial index employed in a variety of commercial database engines such as Oracle Spatial (Greener and Ravada, 2013), MySQL (Schwartz et al., 2012) and PostGIS (Zhang and Yi, 2010). Although the design of R-trees involves leaf-node entries with pointers to spatial data on disk, there are a variety of in-memory R-Tree implementations which store the spatial data in main memory.

R-Trees offer a strong $\mathcal{O}(\log(N))$ worst-case guarantee for insertion of new spatial objects into the tree. The procedure begins at the root of the R-Tree, finding a sub-tree to insert the spatial object into it. A sub-tree of an entry is a candidate if the entry's MBR encloses the MBR of the new spatial object. If such an entry exists, then the process repeats for the next layer. The sub-tree of the satisfying node entry is visited, and again all of the child entries are inspected to find one that encloses the spatial object. The insertion algorithm continues until reaching the leaf layer and inserting the new entry.

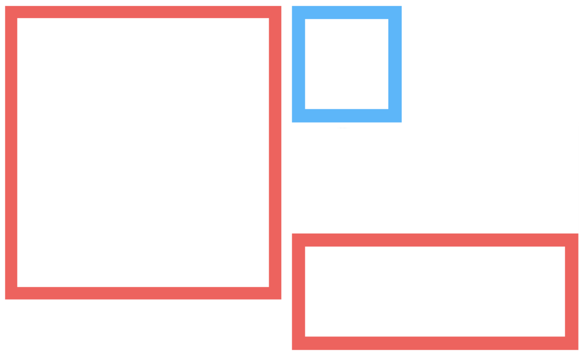


FIGURE 3.17. A Red MBR Must Expand to Accommodate the Blue MBR

However, it is not always the case that a spatial object neatly fits inside a non-leaf node entry's MBR at every level in the tree traversal. Therefore, sometimes the MBR of a non-leaf node must be enlarged to enclose the newly inserted spatial object. Figure 3.17 illustrates this exact scenario where the MBR of the spatial object to be inserted does not fit into any of the red candidate entry's minimum bounding rectangles. Guttman (1984) proposes expanding the candidate rectangle that requires the least increase in area to enclose the new spatial object. In the figure above, the left rectangle would enlarge to enclose the blue rectangle, causing overlap with the left side of the other candidate rectangle. The intuition is that as rectangles expand and occupy more area, there is a higher chance of overlap between rectangles. By reducing overlap between rectangles, it will minimise visiting redundant sub-trees that do not contain the desired spatial objects during range queries. Beckmann et al. (1990) however, introduced a variant of the R-Tree called the R*-Tree which utilises a different heuristic. R* trees minimise the overlap between MBRs rather than minimising the area. Therefore, in the figure above, the R* variant would opt to enlarge the rectangle in the bottom right as it does not create any further overlap with other rectangles.

Another critical parameter affecting the performance of R-Trees is the chosen node splitting algorithm. Much like B-Trees, when a node overflows, a split must occur to divide the overflowing node into two new nodes. When splitting occurs, the B-Tree node entries partition into the two new nodes according to whether they are less or greater than the median of the entry values. After splitting, the links of the parent nodes must be repeatedly adjusted during the traversal up from the overflowed node, ensuring that the ordering invariant of the tree is maintained. However, splitting overflowing nodes as a result of R-Tree insertions is much more complicated. Garcia et al. (1998) demonstrated that computing an optimal node split would require an algorithm with $\mathcal{O}(n^d)$ worst-case time complexity where n is the number of nodes to split and d is the dimensionality of each MBR. Therefore, sub-optimal heuristic splitting strategies are employed to split overflowing nodes to minimise the introduction of overlap.

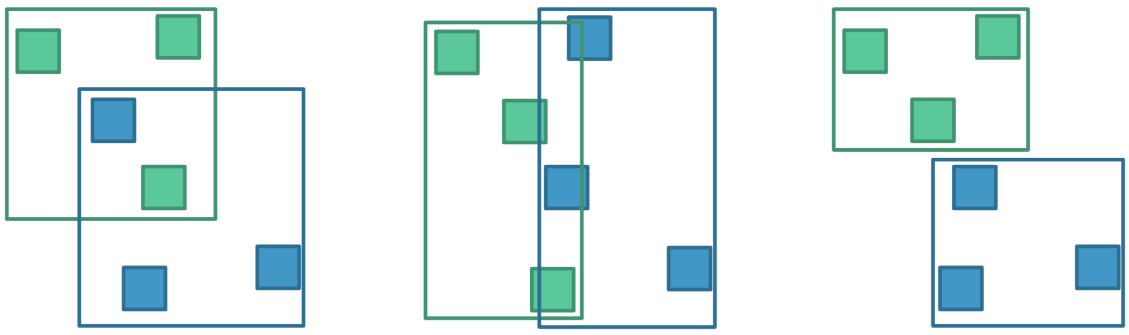


FIGURE 3.18. Potential R-Tree Node Splitting Algorithms: (a) Naive (b) Linear (c) Quadratic

A first idea would be to split the entries in the overflowing node arbitrarily. However, since it takes linear time to perform the split, a simple $\mathcal{O}(n)$ heuristic will outperform a naïve split. The first node splitting algorithm coined by Guttman (1984) is *linear splitting*. First, two candidate rectangles are selected that will belong to opposite nodes after the split. The two rectangles are chosen by picking an arbitrary dimension and finding the two rectangles with the smallest and largest value in that dimension. After this, an assignment of remaining nodes to either partition occurs, whichever results in the least enlargement of the MBR. Another more advanced strategy is *quadratic splitting* (Guttman, 1984). In this approach, from all pairs of rectangles, the two rectangles that would be the most wasteful if put together are chosen. These two rectangles appear in separate partitions as with linear splitting. Next, the remaining rectangles are sorted in decreasing order by the difference in assignment costs to either partition, with the highest difference entries. The intuition is that this heuristic minimises the chance of an entry appearing in the wrong partition and dramatically increasing the cost because it was assigned later. Quadratic splitting as the name implies has a cost of $\mathcal{O}(n^2)$ yet tends to deliver much better splits than linear splitting. Finally, R* Trees employ a unique strategy where after each node split occurs, a percentage of the entries (usually 30%) are forcefully reinserted. The intuition is that since R-Trees are highly sensitive to the order of insertion, the rectangles form clusters based on temporal locality. By reinserting on node overflows the entries are more likely to be reinserted into a region of the tree with entries that are close in space.

```
**for all** $t_1 \in R_1$ **do**
  ...
    **for all** $t_n \in R_n$ **do**
      **project** (...) **into** ...
```

FIGURE 3.19. Loop-Nest Evaluation of a Datalog Rule

The above diagram illustrates the evaluation of a typical loop nest consisting of nested searches over the involved relations. Consider, an index to store each relation which degrades insertion performance by a factor of a constant k. The evaluation of the entire loop nest is, therefore slower by a factor of k. By contrast, consider an index which degrades searching performance by a factor of k. Since searches occur at every level of the loop-nest, the time to evaluate the rule degrades by a factor of k^n. Since real-world programs spend most of their time performing searches in the loop nest, typically with rules involving large numbers of relations, it is crucial that evaluating these searches is fast, even at the cost of insertion performance. Therefore, the R* variant is the most appropriate choice of splitting algorithm for our use case.

There are numerous reasons why R-Trees are suitable to index spatial primitive searches in Datalog. Firstly, R-trees offer high performance for evaluating orthogonal range queries as evidenced by decades of usage in spatial databases. R-Trees also support fast $\mathcal{O}(\log(N))$ insertion performance without the overhead of rebuilding the entire data structure as would happen with a static k-d-tree or range tree. Furthermore, a wide variety of R-Tree splitting algorithms can specialise the index for the workload, making a trade-off between insertion and query performance. Overall, R-trees are the strongest candidate for a dynamic spatial index in a Datalog context for the evaluation of spatial primitive searches.

3.5 A Cluster of B-Trees for Spatial Primitive Searches

In the previous section, we established that R-Trees are a promising candidate for evaluating spatial primitive searches. By mapping each spatial primitive search to an orthogonal range query, every spatial primitive search is covered by a single R-Tree index per relation. However, R-trees do not offer strong guarantees on the time complexity of searches, as in the worst-case, the search complexity is proportional to the size of the relation i.e. $\mathcal{O}(N)$. By comparison, B-Trees have much stronger guarantees on the search complexity, with $\mathcal{O}(|Q| + \log(N))$ where Q is the set of tuples satisfying the search. Therefore when relation sizes are large but the number of tuples satisfying the search is small, B-Trees can offer much stronger performance guarantees. Further, insertion performance is faster for B-Trees as their splitting algorithms are far more straightforward. Finally, the memory overhead of an R-Tree is approximately $2\times$ that of a B-Tree, with each node storing a minimum bounding rectangle alongside each tuple to represent its spatial volume.

For these reasons, we now consider a different index selection strategy, selecting a cluster of B-Trees to cover each spatial primitive search. First, we extend the notion of a lex-search to cover spatial primitive searches, exploiting B-trees to perform these filters. Next, we prove that specific spatial primitive searches cannot be evaluated by a lex-search. Specifically, spatial primitive searches where multiple attributes have inequality constraints cannot be evaluated by a lex-search. We then extend the automatic index selection scheme to cover every *simple spatial primitive search* (an SPS that has at most one attribute with an inequality constraint) for each relation.

3.5.1 Lex Searches

As with equality primitive searches, we wish to translate spatial primitive searches into lex-searches on an index ℓ. Recall that a lex-search has the form:

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

Given a spatial primitive search $\sigma_{l_1 \leq x_1 \leq u_1, \ldots, l_k \leq x_k \leq u_k}$ we want to construct our lower and upper bounds a and b and characterise which indexes ℓ can cover a given spatial primitive search. If $k = m$ and for every attribute x_i there is a provided lower bound l_i and upper bound u_i then we can set $a = \langle l_1, ..., l_m \rangle$ and $b = \langle u_1, ..., u_m \rangle$. However, as with the translation from equality primitive searches to lex-searches, if a lower bound or upper bound is missing then it must be padded with infimum or supremum values.

We define two index mapping functions from the index of an attribute of the relation to the index of the corresponding constrained attribute:

$$\mathit{lower} : \{1, ..., m\} \to \{1, ..., k+1\}$$

$$\mathit{upper} : \{1, ..., m\} \to \{1, ..., k+1\}$$

Two index mapping functions are required with lower mapping indexes from the relation to the set of lower bound constraints L and upper mapping to the set of upper bound constraints U.

If an attribute does not appear in the search predicate then it will be mapped to index $k+1$ where l_{k+1} and u_{k+1} are defined to be Δ, as before. However, now with the introduction of lower and upper bounds, an attribute can be constrained without both bounds having specified values. This occurs when the range is one-sided and either the lower bound l_i or upper bound u_i is unspecified. To handle these one-sided ranges lower will map an index i to index $k+1$ if l_i is left unspecified and similarly upper will do the same when u_i is unspecified.

When constructing the bounds $a = lb(l_1, ..., l_k) = \langle l'_1, ..., l'_k \rangle$ where:

$$l'_j = \left\{ \begin{array}{ll} l_{\mathit{lower}_j}, & \text{if } l_{\mathit{lower}_j} \neq \Delta \\ \inf(\mathbb{D}_j), & \text{otherwise} \end{array} \right\}$$

and $b = ub(u_1, ..., u_k) = \langle u'_1, ..., u'_k \rangle$ where:

$$u'_j = \left\{ \begin{array}{ll} u_{upper_j}, & \text{if } u_{upper_j} \neq \Delta \\ \sup(\mathbb{D}_j), & \text{otherwise} \end{array} \right\}$$

3.5.2 Expressiveness of Lex Searches on Spatial Primitive Searches

Subotić et al. (2018) proved that an index ℓ covers an equality primitive search $\sigma_{x_1=v_1,...,x_k=v_k}(R)$ for all $R \subseteq \mathcal{D}$ **if and only if** the k^{th}-prefix of ℓ is comprised of only attributes in the set $\{x_1, ..., x_k\}$. However, the kinds of spatial primitive searches computable by lex-searches are far more restricted.

LEMMA 1. *Given a spatial primitive search $\sigma_{l_1 \leq x_1 \leq u_1,...,l_k \leq x_k \leq u_k}$ with a range constraint on attribute x_i and a corresponding search set S on a relation R then the search cannot be covered by an index ℓ if x_i does not appear as the last attribute in the k^{th}-prefix of ℓ.*

PROOF. Spatial primitive searches generalise equality primitive searches so the k^{th}-prefix constraint must be satisfied for spatial primitive searches also. Therefore, if S is not a k^{th} prefix then the theorem is true. We assume S is a k^{th} prefix from this point forward. ℓ must be of the form $\ell = ... \prec x_i \prec ...$ where the attributes occurring in the lex-search may be any from the set A_R. Now the k^{th}-prefix must be of the form: $\ell_k = ... \prec x_i \prec ...$ using attributes only from the set S. Since x_i is not the last attribute in the k^{th}-prefix there must be some other attribute at the end i.e. x_j. Therefore, we can write $\ell_k = ... \prec x_i \prec ... \prec x_j$.

Now consider the tuple with the order of attributes written in the order of ℓ_k where:

$$t_1 = \langle ..., l_i, ..., u_j + 1 \rangle$$

$$R = \{t_1\}$$

Considering the result of the spatial primitive search:

$$\sigma_{l_1 \leq x_1 \leq u_1,...,l_k \leq x_k \leq u_k}(R) = \emptyset$$

The set of satisfying tuples is empty because the value of attribute x_j is $u_j + 1$ which exceeds the upper bound in the spatial primitive search.

Now, when considering the lex search: We have:

$$a = \langle ..., l_i ..., l_j \rangle$$

$$b = \langle ..., u_i, ..., u_j \rangle$$

We note that $l_i \neq u_i$ since x_i has a range constraint.

As a result, we have $a \sqsubseteq_\ell t_1 \sqsubseteq_\ell b$ since lexicographically t_1 compares greater than a and smaller than b and so t_1 must be in the result set of the lex-search i.e.

$$\sigma_{\rho(\ell,a,b)}(R) = \{t_1\}$$

The lex-search and spatial primitive search do not have the same semantics i.e.

$$\sigma_{l_1 \leq x_1 \leq u_1, ..., l_k \leq x_k \leq u_k}(R) \neq \sigma_{\rho(\ell,a,b)}(R)$$

Therefore, there does not exist any ℓ that covers the spatial primitive search. □

COROLLARY 1. *If an attribute x_i has an inequality constraint then it must appear at the end of the k^{th}-prefix of ℓ. Since only one attribute can be at the end of the k^{th}-prefix, it is impossible for a spatial primitive search with multiple range constraints to be covered by any index.*

3.5.3 Evaluating Spatial Primitive Searches with Multiple Range Constraints

We have demonstrated that if there are multiple range constraints, then not all of them can be used by a lex-search. If each spatial primitive search has at most one range constraint (i.e. it is a simple spatial primitive search), then no transformation is required. A transformation must occur for spatial primitive searches with multiple range constraints in order to evaluate them with lex-searches. Specifically, range constraints need to be removed from the spatial primitive search until at most one remains for a lex-search to cover it. However, merely removing the range constraint will not preserve the semantics of the spatial primitive search.

The are two distinct strategies to transform a spatial primitive search with multiple range constraints in order to use lex-searches.

(1) Given a spatial primitive search with k range constraints, perform k simple spatial primitive searches each using one of the range constraints in the indexed search. Next, perform a set-wise intersection of all of these results, leaving only tuples satisfying all range constraints.

(2) Given a spatial primitive search, discharge range constraints to nested filter operations repeatedly until at most one range constraint remains in the indexed search.

```
for all $t_0 \in R_0$ do
  ...
    for all $t_k \in R_k$ on index $r_1$ and on index $r_2$ and on index $r_3$ do
      ...
        project (...) into ...
```

FIGURE 3.20. Strategy 1

```
for all $t_0 \in R_0$ do
  ...
    for all $t_k \in R_k$ on index $r_1$ do
      if $r_2 \wedge r_3$ do
        ...
          project (...) into ...
```

FIGURE 3.21. Strategy 2

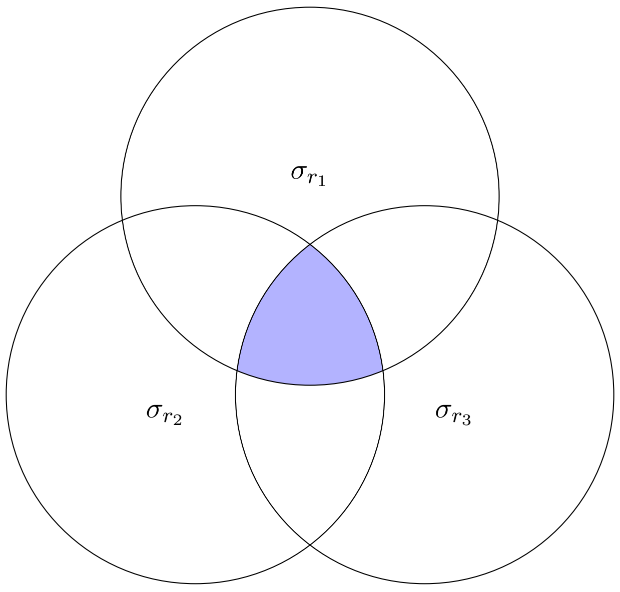


FIGURE 3.22. Venn Diagram for Sets of Tuples Satisfying Spatial Primitive Searches

A spatial primitive search with multiple range constraints can be decomposed into the intersection of different simple spatial primitive searches, each with a single range constraint. Figure 3.22 illustrates the sets of tuples satisfying each decomposed simple spatial primitive search. The intersection of these sets contains all tuples satisfying the original spatial primitive search. Strategy 1 requires computing each set separately and then performing a set wise intersection. Strategy 2, by contrast, computes a single one of these sets and filters the tuples for those satisfying the remaining inequality constraints.

To prove that Strategy 2 outperforms Strategy 1 we consider their respective evaluation complexity in the worst-case. For Strategy 1, a lex-search would need to be performed for each σ_{r_i} which would consume $\mathcal{O}(\sum\limits_i |\sigma_{r_i}(R)| + \log(|R|))$ time. After computing each lex-search individually, the set-wise intersection would then need to be computed. By utilising hash-sets, the intersection can be computed in $\mathcal{O}(\sum\limits_i |\sigma_{r_i}(R)|)$. Therefore, we can bound the overall complexity of this approach by $\mathcal{O}(\sum\limits_i |\sigma_{r_i}(R)| + \log(|R|))$.

Now for Strategy 2 we pick the first range constraint r_i to use in the indexed search and then perform a filter on the remaining range constraints. Assume for the worst-case analysis that we select the least constraining range predicate i.e. $\arg\max\limits_i(|\sigma_{r_i}(R)|)$. When performing the corresponding lex-search this would take $\mathcal{O}(\arg\max\limits_i(|\sigma_{r_i}(R)| + \log(|R|)))$. Finally, the result set of the lex-search must be iterated to filter for the tuples satisfying all of the remaining range constraints which takes $\mathcal{O}(\arg\max\limits_i(|\sigma_{r_i}(R)|))$. Therefore, the overall time complexity is bounded by $\mathcal{O}(\arg\max\limits_i(|\sigma_{r_i}(R)| + \log(|R|)))$ which grows slower asymptotically than $\mathcal{O}(\sum\limits_i |\sigma_{r_i}(R)| + \log(|R|))$. Thus, Strategy 2 will outperform Strategy 1.

We note that during index selection time, the distribution of tuples for a relation is not known. Therefore, there is no way to determine which attribute has the most selective inequality constraint and retain it in the simple spatial primitive search. For simplicity, the first attribute with a range constraint is retained in the simple spatial primitive search using the attribute order as it is written in the relation's declaration in the Datalog program. For all subsequent range constraints, they are discharged to filter operations further down in the loop nest.

3.5.4 The Minimum Index Selection Problem for Simple Spatial Primitive Searches

We have now demonstrated that in order to evaluate spatial primitive searches with multiple range constraints using lex-searches, only one range constraint can be used in the search. Therefore, for every spatial primitive search we use the strategy outlined previously to discharge all range constraints to filter operations, transforming the original spatial primitive search into a simple spatial primitive search.

The focus now is to extend the framework of Subotić et al. (2018) to cover all simple spatial primitive searches. We reuse the same objective of the original MISP to minimise the number of indexes such that all simple spatial primitive searches for a given relation are covered. If all simple spatial primitive searches for a given relation have no range constraints, then the original technique can be utilised with no issue. However, to use indexes to speed up inequality primitive searches, we must consider the case where inequality constraints appear in some of the simple spatial primitive searches for a relation. We now extend the original auto-index selection technique by defining a partial ordering over simple spatial primitive searches and reducing our minimum index selection problem to a minimum chain covering problem.

First, we introduce the notation:

In the original formulation of the MISP, each equality primitive search has a corresponding search set S. i.e.

$$\sigma_{x_1=v_1,\ldots,x_k=v_k} \mapsto \{x_1, \ldots, x_k\}$$

Likewise, we define a mapping of a simple spatial primitive search to a search set S. The attribute x_i appears in the search set if at least one of its lower bound l_i or upper bound u_i values are specified. i.e.

$$\sigma_{l_1\leq x_1\leq u_1,\ldots,l_k\leq x_k\leq u_k} \mapsto \{x_1, \ldots, x_k\}$$

However, a single search set is not enough to abstract the simple spatial primitive search since an attribute appearing in the search set does not indicate whether an equality or inequality constraint is acting on the attribute. Therefore, we introduce a mapping function from a simple spatial primitive search to a pair of search sets (S_{EQ}, S_{INEQ}). S_{EQ} contains the attributes in the simple spatial primitive search with equality constraints i.e. $l_i = u_i$ for a constrained attribute x_i. Whereas S_{INEQ} contains the attributes with range constraints i.e. $l_i \neq u_i$. For convenience, we define $S = S_{EQ} \cup S_{INEQ}$, the search set containing all attributes with either an equality or inequality constraint.

In the index selection scheme for equality primitive searches, Subotić et al. (2018) used the fact that for a pair of searches S and S', if $S \subset S'$ and an index ℓ covers S', then S can also be covered by ℓ. However, simple spatial primitive searches are far more restrictive. If the search has a range constraint then it is no longer sufficient for a search to be the k^{th} prefix of an index ℓ for it to be covered.

3.5.5 Characterising Indexes which Cover Simple Spatial Primitive Searches

LEMMA 2. *Given a simple spatial primitive search $\sigma_{l_1 \leq x_1 \leq u_1, \ldots, l_k \leq x_k \leq u_k}$ and its corresponding pair of search sets (S_{EQ}, S_{INEQ}), and $x_i \in S_{INEQ}$ then the pair of search sets can be covered by an index ℓ* ***if and only if*** *S is the k^{th}-prefix of ℓ and x_i appears as the last attribute in the k^{th}-prefix.*

The proof for Lemma 2 can found in Appendix B of this thesis.

Now that we have fully characterised the requirements for a simple spatial primitive search to be evaluated as a lex-search, we must modify the automatic index selection scheme to accommodate these new requirements. Therefore, we want to characterise the precise conditions under which an index ℓ can cover two simple spatial primitive searches.

3.5.6 Covering Simple Spatial Primitive Searches with a Common Index

LEMMA 3. *Given two search pairs (S_{EQ}, S_{INEQ}) and (S'_{EQ}, S'_{INEQ}) satisfying $|S_{INEQ}| \leq 1, |S'_{INEQ}| \leq 1$, there exists an index ℓ that covers both (S_{EQ}, S_{INEQ}) and (S'_{EQ}, S'_{INEQ})* ***if and only if****:*

(1) *$S \subseteq S'$ where $S = (S_{EQ} \cup S_{INEQ})$ and $S' = (S'_{EQ} \cup S'_{INEQ})$.*
(2) *If $x_i \in S'_{INEQ}$ then $x_i \notin S$.*

The proof for Lemma 3 can found in Appendix C of this thesis.

3.5.7 Defining the Minimum Index Selection Problem (MISP)

Now that we have defined the circumstances under which two search pairs derived from simple spatial primitive searches can be covered by a common index, we can formally define the minimum index selection problem for simple spatial primitive searches. First, we redefine the notion of an **l-cover** from Subotić et al. (2018), to verify whether a selection of indexes covers every search pair derived from a collection of simple spatial primitive searches. We define an **l-cover** as follows:

Given a set of search pairs $\mathcal{S}$ where every $(S_{EQ}, S_{INEQ}) \in \mathcal{S}$ satisfies $|S_{INEQ}| \leq 1$, and a set $\mathcal{L}$ of indexes on a relation R, we define a predicate **l-cover**$_{\mathcal{S}}(\mathcal{L})$ which holds true if for every search pair $(S_{EQ}, S_{INEQ}) \in \mathcal{S}$, there exists an index $\ell \in \mathcal{L}$ that covers (S_{EQ}, S_{INEQ}).

Given the notion of an **l-cover** we would like to find the minimum set of indexes $\mathcal{L}$ to cover all search pairs in $\mathcal{S}$. Therefore, we redefine the minimum index selection problem as follows:

The Minimum Index Selection Problem (MISP) for Simple Spatial Primitive Searches. Given a set of search pairs $\mathcal{S}$ where every $(S_{EQ}, S_{INEQ}) \in \mathcal{S}$ satisfies $|S_{INEQ}| \leq 1$ on a relation R, the minimum index selection problem is to find the minimum cardinality set of indexes $\mathcal{L}$ such that all search pairs are covered by the index set i.e.

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

3.5.8 Reduction to The Minimum Chain Covering Problem (MCCP)

We have shown that there exists an index ℓ to cover any two search pairs derived from simple spatial primitive searches if:

(1) $S \subseteq S'$ where $S = (S_{EQ} \cup S_{INEQ})$ and $S' = (S'_{EQ} \cup S'_{INEQ})$.
(2) If $x_i \in S'_{INEQ}$ then $x_i \notin S$.

Now, we wish to construct such an index, given a chain of searches satisfying these constraints by reduction to a minimum chain covering problem. For simplicity, for two search pairs if they satisfy the above criteria we write $(S_{EQ}, S_{INEQ}) < (S'_{EQ}, S'_{INEQ})$. In effect, we are defining a new partial ordering over search pairs derived from simple spatial primitive searches. Intuitively, by defining a new partial order which coincides with our requirements for two search pairs to share an index, we can reuse the original technique by Subotić et al. (2018), minimising the cardinality of a corresponding chain covering over the lattice to retrieve a minimal index selection. By our new partial ordering, we redefine a search chain as:
$C = \{(S^1_{EQ}, S^1_{INEQ}), ..., (S^k_{EQ}, S^k_{INEQ})\}$ such that $(S^i_{EQ}, S^i_{INEQ}) < (S^{i+1}_{EQ}, S^{i+1}_{INEQ})$

LEMMA 4. *Given a search chain $C = (S^1_{EQ}, S^1_{INEQ}) < ... < (S^k_{EQ}, S^k_{INEQ})$, we can construct an index ℓ to cover all simple spatial primitive searches that map to search pairs in C **if** $\ell = S^1 \prec (S^2 - S^1) \prec ... \prec (S^k - S^{k-1})$ where the attributes of S^1 and $S^i - S^{i-1}$ are ordered arbitrarily except for any attribute appearing in S^i_{INEQ} it appears later in the sub-order than any attribute in S^i_{EQ} i.e. $S^i_{EQ} \prec S^i_{INEQ}$ with respect to ℓ.*

PROOF.

Following the chain, each of the attributes in a pair of searches is a subset of the next. Therefore, all of the searches are k^{th}-prefixes of ℓ and can be covered by if they have no range constraints.

Next, we consider if a search has a range constraint i.e. there exists an attribute $x \in S^i_{INEQ}$. If so then x appears at the end of the sub-order induced by $S^i - S^{i-1}$. For search set pair, if $x \in S^i_{INEQ}$, we know that $(S^{i-1}_{EQ}, S^{i-1}_{INEQ}) < (S^i_{EQ}, S^i_{INEQ})$ and hence $x \notin S^{i-1}$ and so attribute x appears later in the lex-order than all attributes in S^{i-1}. Additionally, for an attribute with an inequality constraint, it appears at the end of the sub-order within $S^i - S^{i-1}$. Therefore, x occurs at the end of the k^{th}-prefix of ℓ, and by Lemma 2 can be covered by ℓ. The above proof holds for all elements in the chain, and thus all search pairs can be covered by the same index ℓ. □

We have demonstrated that given a search chain C, we can construct an index ℓ that covers all of the search pairs in the chain. Since this can be done for every chain, we can build indexes in this way to cover the searches within all search chains. However, we now wish to formalise whether a set of chains $\mathcal{C}$ can cover all search pairs in the set $\mathcal{S}$. We redefine the notion of a **c-cover** from Subotić et al. (2018). We define a **c-cover** as follows:

Given a set of search pairs $\mathcal{S}$ where every $(S_{EQ}, S_{INEQ}) \in \mathcal{S}$ satisfies $|S_{INEQ}| \leq 1$, and a set $\mathcal{C}$ of search chains on a relation R, we define a predicate $\textbf{c-cover}_{\mathcal{C}}(\mathcal{S})$ which holds true if for every search pair $(S_{EQ}, S_{INEQ}) \in \mathcal{S}$, there exists a search chain $C \in \mathcal{C}$ that covers (S_{EQ}, S_{INEQ}). i.e.

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

We can therefore define the minimum chain covering problem as follows:

The Minimum Chain Covering Problem (MCCP) for Simple Spatial Primitive Searches. Given a set of search pairs $\mathcal{S}$ where every $(S_{EQ}, S_{INEQ}) \in \mathcal{S}$ satisfies $|S_{INEQ}| \leq 1$ on a relation R, the minimum chain covering problem is to find the minimum cardinality set of chains $\mathcal{C}$ such that all search pairs are covered by the set of chains i.e.

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

We now demonstrate that there is a one-to-one correspondence between solutions of MISP for Simple Spatial Primitive Searches and MCCP for Simple Spatial Primitive Searches.

LEMMA 5. *Given any search set $\mathcal{S}$ where each search pair $(S_{EQ}, S_{INEQ}) \in \mathcal{S}$ satisfies $|S_{INEQ}| \leq 1$, there is a one-to-one correspondence between search chains $\mathcal{C}$ that cover $\mathcal{S}$, such that $|\mathcal{C}| = |\mathcal{S}|$.*

PROOF.
By Lemma 4 we have shown that given a set $\mathcal{C}$ of search chains covering all search pairs derived by simple spatial primitive searches in the set $\mathcal{S}$, one can construct an index set $\mathcal{L}$ of cardinality $|\mathcal{C}|$ which also covers $\mathcal{S}$. We now prove the correspondence in the opposite direction i.e. given a set of indexes $\mathcal{L}$ that cover all search pairs derived by simple spatial primitive searches in $\mathcal{S}$ that one can construct a set of chains $\mathcal{C}$ of cardinality $|\mathcal{L}|$ that cover $\mathcal{S}$. We do this by proving that for any index ℓ, one can construct a search chain C that covers all the search pairs covered by ℓ.

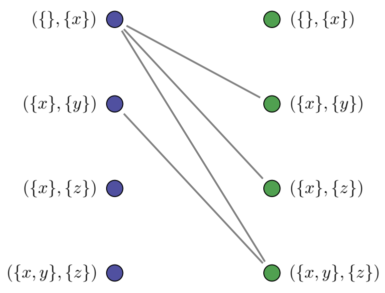
Following the notation of Subotić et al. (2018), let $\mathcal{S}_\ell$ denote the subset of search pairs from $\mathcal{S}$ that are covered by ℓ. By Lemma 3, for any two search pairs $(S_{EQ}, S_{INEQ}), (S'_{EQ}, S'_{INEQ})$ covered by the same index ℓ, they must satisfy:

(1) $S \subseteq S'$ where $S = (S_{EQ} \cup S_{INEQ})$ and $S' = (S'_{EQ} \cup S'_{INEQ})$.
(2) If $x_i \in S'_{INEQ}$ then $x_i \notin S$.

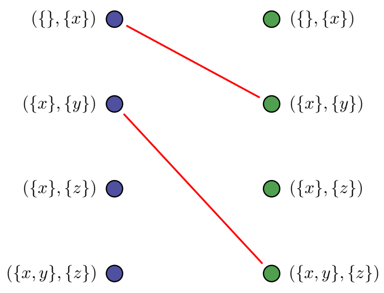
The above two conditions are precisely those needed to conclude $(S_{EQ}, S_{INEQ}) < (S'_{EQ}, S'_{INEQ})$. Therefore, one can construct a search chain C of these search pairs which covers $\mathcal{S}_\ell$. Since the above procedure can be performed for all indexes ℓ in the index set $\mathcal{L}$, one can construct a set of chains $\mathcal{C}$ of cardinality $|\mathcal{L}|$ that cover $\mathcal{S}$, completing the proof. $\square$

COROLLARY 2. *Given any set of search pairs derived from simple spatial primitive searches $\mathcal{S}$ on a relation R, a minimal index selection can be found by computing a minimum chain covering.*

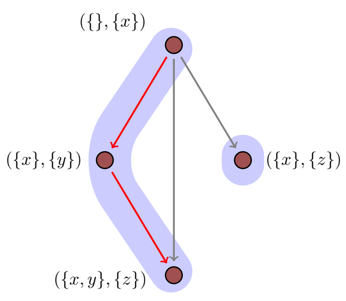
3.5.9 Algorithms to Solve the MISP in Polynomial Time



Bipartite Graph



Maximum Matching



Chain Cover

We have demonstrated the equivalence between solutions of the Minimum Index Selection Problem (MISP) and the Minimum Chain Covering Problem (MCCP) with our construction. Therefore, to compute our minimum index selection, we only need to solve the corresponding minimum chain covering problem and translate the corresponding search chains to indexes. We follow a similar approach to Subotić et al. (2018), solving the MCCP by Dilworth's Theorem (Dilworth, 2009). We reuse the reduction to a maximum matching over a bipartite graph (Fulkerson, 1956) to solve the MCCP. The above figure illustrates the algorithm.

Algorithm 3 MinChainCover ($\mathcal{S}$)

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

$\mathcal{M} \leftarrow$ MaximumMatching $(\mathcal{S}, \mathcal{S}, \{((S_{EQ}, S_{INEQ}), (S'_{EQ}, S'_{INEQ})) \in \mathcal{S} \times \mathcal{S} \mid (S_{EQ}, S_{INEQ}) < (S'_{EQ}, S'_{INEQ})$
and $(S_{EQ}, S_{INEQ}) \neq (S'_{EQ}, S'_{INEQ})\})$
Initialize $\mathcal{C}$ to be the empty set
for all $u_1 \in \mathcal{S}$ s.t. $\nexists(u_0, u_1) \in \mathcal{M}$ do
    Find maximal path $(u_1, u_2), (u_2, u_3), \ldots, (u_{k-1}, u_k) \subseteq \mathcal{M}$
    Add $u_1 < u_2 < u_3 < \cdots < u_{k-1} < u_k$ to $\mathcal{C}$
return $\mathcal{C}$
```

To compute a minimum chain covering, we construct a bipartite graph where each node represents a search pair. Each side of the bi-partition contains all search pairs as vertices. An edge is drawn from a node on the left side to a node on the right side when the search pair on the left compares lower than the search pair on the right by our partial ordering. Each maximal-path in the max matching then corresponds to a search chain in the lattice. We note that we only consider pairs of search pairs that are distinct to ensure a well-defined partial ordering over the elements of the lattice.

Algorithm 4 MinIndex ($\mathcal{S}$)

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

$\mathcal{C} \leftarrow$ MinChainCover($\mathcal{S}$)
Initialize $\mathcal{L}$ to be the empty set
for all $(S^1_{EQ}, S^1_{INEQ}) < (S^2_{EQ}, S^2_{INEQ}) < \cdots < (S^{k-1}_{EQ}, S^{k-1}_{INEQ}) < (S^k_{EQ}, S^k_{INEQ}) \in \mathcal{C}$ do-
    Add to $\mathcal{L}$ an arbitrary index conforming with
      [1] $S^1 \prec (S^2 - S^1) \prec \ldots \prec (S^k - S^{k-1})$
      [2] $S^i_{EQ} \prec S^i_{INEQ}$ for all search pairs

return $\mathcal{L}$
```

To compute a minimum index selection, we first construct a minimum chain covering over the set of search pairs. We then follow the procedure described previously to construct an index that covers the same search pairs covered by the chain. Finally, we return this minimum index selection.

The correctness of our algorithm follows from our previous lemmas. With regards to the time complexity, to construct the bipartite graph, we still consider all pairs of searches of which there are $|\mathcal{S}|^2$ where $S = S_{EQ} \cup S_{INEQ}$, and compare them concerning the partial order which requires considering in the worst case all m attributes of the relation for each pair of search pairs. Further, the complexity of the maximum matching remains the same. Therefore the time complexity of our auto-index selection algorithm is identical to the state-of-the-art index selection technique i.e. $\mathcal{O}(|\mathcal{S}|^{2.5} + |\mathcal{S}|^2 \cdot m)$.

3.5.10 Robustness and Efficiency of B-Tree SPS

We denote this new index selection strategy as B-Tree SPS (Spatial Primitive Search), automatically building a minimal cluster of B-Tree indexes to accelerate all simple spatial primitive searches. We note that any binary search tree index can replace B-Trees in our index selection scheme since any binary search tree can perform one-dimensional range queries with the desired complexity bounds. However, in our implementation, we use B-Tree indexes, so we adopt this naming convention. To the best of our knowledge, the B-Tree SPS technique is novel in its approach in computing a minimum index selection for inequality primitive searches, automatically and in polynomial time.

B-Tree SPS has several advantages over the state-of-the-art technique. The primary advantage is that both equality primitive searches and inequality primitive searches for a given relation are considered together when performing the index selection, allowing for inequality primitive searches to appear in the same search chains as equality primitive searches. The effect is that even though an index must now cover new inequality primitive searches, it is likely for these extra searches to be covered without building any more indexes than the original technique. We suspect given how infrequent inequality primitive searches occur, that covering the extra inequality searches without additional indexes is likely to occur.

If no new extra indexes are built, there will be zero memory overhead relative to the original technique as no additional maintenance is needed compared to the original technique. More importantly, evaluation time will be improved considerably when inequality constraints restrict the set of satisfying tuples heavily. The index selection time will be only slightly longer than the original technique since the complexity of B-Tree SPS is identical, with only a few new searches adding to the index selection computation. The compilation time is also not increased if no additional B-Tree indexes are required. Therefore, the B-Tree SPS technique is highly efficient, accelerating inequality primitive searches with indexes and potentially incurring zero overhead in both memory overhead and compilation time.

Furthermore, even when new extra indexes are required, because the same search chains can cover inequality primitive searches as equality primitive searches, the number of additional indexes would also be small. The impact on compilation time and memory overhead would be small while still potentially improving the evaluation time of inequality primitive searches significantly. The only noticeable overhead would be the maintenance of these extra indexes. Typically, real-world Datalog applications consume the majority of their run-time evaluating searches rather than inserting into indexes. Hence, we expect the overhead to maintain the additional indexes to be overshadowed by the performance improvement of evaluating inequality primitive searches efficiently. Therefore, the B-Tree SPS index selection technique is robust for real-world Datalog programs.

CHAPTER 4

Experiments and Results

For our experiments we compare three index selection schemes and evaluate their performance using various data sets and programs. The following index selection schemes are used for our comparison:

- **R-Tree SPS (Spatial Primitive Searches):** This index scheme uses an R-Tree for a relation, if there exists at least one inequality primitive search. Otherwise, a cluster of B-Tree indexes covers the equality primitive searches of the relation.
- **B-Tree SPS (Spatial Primitive Searches):** This indexing scheme uses a cluster of B-Tree structures to cover all simple spatial primitive searches for a relation. If there are multiple attributes with inequality constraints in an SPS, then it is transformed into a simple spatial primitive search and remaining attributes with inequality constraints are discharged to filter operations.
- **Base:** This index selection strategy uses a cluster of B-Trees for a relation to cover all of its equality primitive searches. All inequalities are executed as filter operations.

We evaluate these index strategies under different workloads to show their efficiency and effectiveness. In this chapter, we first present the aims of the experimental analysis, and our rationale for our choice of experiments. Next, we discuss the empirical results and answer these aims. Finally, we present the outcome of these experiments and discuss their implications.

Our aims are to answer the following experimental questions:

(Q1) What proportion of spatial primitive searches are inequality primitive searches in real-world applications? In particular, how often do inequality primitive searches appear in Datalog programs and how often are they executed?

(Q2) Is R-Tree SPS a better index selection strategy than Base? In particular, how does it compare with regards to compilation time, memory consumption, and evaluation time?

(Q3) Is B-Tree SPS a better index selection strategy than Base? In particular, how does it compare with regards to compilation time, memory consumption, and evaluation time?

(Q4) Which index selection strategy is better for real world applications? R-Tree SPS or B-Tree SPS?

The first strategy, R-Tree SPS, has the benefit that a *single* R-Tree index can speed up every spatial primitive search over a relation. Furthermore, R-Trees can evaluate spatial primitive searches with any number of attributes with inequality constraints. However, R-Trees are multi-dimensional data structures, and thus insertion and searching are more involved with $\mathcal{O}(n)$ worst-case search performance where n is the size of the relation. Only when the data within the R-Tree is well organised can the search be performed efficiently. Overall, in the worst-case R-Trees require time proportional to the size of the relation when performing searches. Another drawback is that R-Trees are complex structures that require storage of minimum bounding rectangles (MBRs) alongside each entry in the tree. Since two corner points define each MBR, the memory overhead of the R-Tree index is approximately $2\times$ that of a single B-Tree index.

The second strategy, B-Tree SPS, has the benefit that search performance is $\mathcal{O}(|Q| + \log(n))$ in the worst-case; therefore the evaluation complexity is always bounded to take time proportional to the size of the output. The primary disadvantage of B-Tree indexes for covering spatial primitive searches is that multiple B-Tree indexes may be required to cover every search. Extra B-Tree indexes may be constructed compared to Base to cover new inequality primitive searches. New tuples must be inserted into all B-Tree indexes to keep them consistent, degrading insertion performance and increasing memory usage. Additionally, constructing extra B-Tree indexes adds to the overall compilation time. The other drawback of a B-Tree index is that searches can only support simple spatial primitive searches, discharging any extra attributes with inequality constraints to filter operations. Finally, the third strategy, Base, acts as an experimental control representing the current state of the art index selection scheme used in SOUFFLÉ.

A key observation that can be made is that if no inequalities are present in a Datalog program, then all three strategies will perform identically. R-Tree SPS constructs R-Trees only when inequalities are present in a search. Therefore, when there are none, B-Trees will index all relations. Similarly, the treatment of searches is identical between B-Tree SPS and Base when the searches contain no inequalities, and thus the selection of indexes will be identical. Since both strategies are robust in the case where no inequality constraints are present, our experimental analysis will explore the effectiveness of either technique to accelerate inequality primitive searches while trading off compile time and memory overhead.

For our R-Tree index, we use the C++ Boost R-Tree implementation using version 1.71 of Boost. The implementation is highly optimised, heavily utilising compile-time specialisation through templates to enhance evaluation time performance. For a B-Tree index, we use the existing B-Tree implementation in SOUFFLÉ (Jordan et al., 2019b).

For our experiments, we use an AMD Ryzen Threadripper 2990WX 32-Core processor with a base clock speed of 3.0GHz and a maximum boost clock speed of 4.2GHz. The machine has 128GB of RAM, and runs Ubunutu version 20.10 as an OS. Version 9.3.0 of GCC compiles the C++ programs generated by SOUFFLÉ into binaries. GCC compiles the programs with the -O3 flag enabled and with assertions disabled. Additionally, SOUFFLÉ constructs all program to run with a single thread since the Boost R-Tree implementation is not thread-safe. Tree traversal hints are disabled for B-Tree indexes since the Boost R-Tree does not use this technique in its implementation. For Boost's R-Tree parameter, we employ the R* splitting algorithm and select the maximum entries per node to be a fixed size of 16 to control the frequency of node splitting and re-insertions.

We used the Linux 'time' command to measure compilation time, memory consumption and evaluation time of programs with the various index selection strategies. Since SOUFFLÉ is an in-memory Datalog engine, excessive memory consumption can result in the machine running out of memory and crashing during evaluation. Therefore, we measure the maximum resident set size during program evaluation since this represents the memory required to perform the analysis without crashing. To summarise the performance characteristics of each technique relative to the original scheme, we use the following statistics:

$$\text{Compilation Overhead} = \frac{\text{Compilation Time using New Strategy}}{\text{Compilation Time using Base}}$$

$$\text{Memory Overhead} = \frac{\text{Maximum RSS using New Strategy}}{\text{Maximum RSS using Base}}$$

$$\text{Evaluation time Speedup} = \frac{\text{Evaluation Time using Base}}{\text{Evaluation Time using New Strategy}}$$

In order to demonstrate the effectiveness of each technique to accelerate inequality primitive searches we use a combination of synthesised and real-world benchmarks.

- **Nearby Natural Numbers** is a synthesised benchmark to compute all pairs of numbers within a given range of each other. The benchmark exploits indexed inequality primitive searches to prune the search space eagerly.
- **Nearby Points** is another synthesised benchmark which is a two dimensional analogue of the previous benchmark. The program computes all points that are within a given range of each other, serving to demonstrate how each strategy performs when multiple attributes have inequality constraints.
- **Tax** is the next synthesised benchmark. This benchmark showcases a spatial primitive search with two different attributes with inequality constraints. Each attribute with an inequality constraint is weakly selective but when used in conjunction they are highly selective. The benchmark stresses evaluation performance for multi-dimensional searches.
- **Insert** is the final synthesised benchmark, stressing the maintenance cost of each index selection strategy by inserting $10,000,000$ entries into a relation. The searches on the relation can only be covered by constructing $n\times$ the number of B-Tree indexes for the B-Tree SPS technique compared to Base.
- **DOOP** (Bravenboer and Smaragdakis, 2009) is a static program analysis framework deploying pointer-analysis for Java programs. The included DaCapo suite of Java programs serve as input in our experimental evaluation of the context-insensitive analysis. DOOP is a large scale benchmark, taking millions of tuples as input and generating tens of millions of output tuples. The implementation of DOOP is highly optimised to take advantage of the underlying evaluation engine of SOUFFLÉ. In particular, loop schedules which are most crucial to achieving high performance are user annotated to achieve optimal performance. Therefore, the effect on the indexing strategy will be most apparent in DOOP as other factors affecting the evaluation time are already highly optimised. Since inequality primitive searches occur very infrequently, we expect no improvement in evaluation time for DOOP. However, DOOP serves as a critical benchmark to demonstrate the robustness of the index selection strategies as any extra overhead introduced by either technique will be immediately apparent.
- **Amazon Virtual Private Cloud (VPC) Security Analysis** (Backes et al., 2019) is the next real-world benchmark, detecting security vulnerabilities in Amazon's Virtual Private Cloud service. Unlike DOOP, which is handcrafted to take advantage of the evaluation strategies

of SOUFFLÉ, a domain-specific language generates the Datalog program for VPC, resulting in unoptimised output. Therefore, inefficient loop schedules will be present and be responsible for the majority of the evaluation time. Additionally, VPC has few inequality primitive searches and therefore we expect to see no improvement from either index selection strategy.

- **DDISASM** (Flores-Montoya and Schulte, 2020) is a dis-assembler which takes as input a stripped binary program and produces assembly that compiles down to the same binary. The tool represents state of the art in disassembly, outperforming and producing more accurate output than other tools. DDISASM uses inequality constraints infrequently, yet inequality primitive searches consume a large proportion of the evaluation time. We, therefore, expect considerable performance gains with our indexing strategies for DDISASM. We note that loop schedules were hand optimised to eliminate their influence from our performance evaluation and instead showcase the performance impact of the new indexing schemes.

4.1 (Q1) Frequency and Distribution of Spatial Primitive Searches

We want to evaluate the extent to which inequality primitive searches occur statically in Datalog programs and to what extent they contribute to their evaluation time. In order to collect this data, we use the Base index selection scheme on a suite of real-world programs, i.e. DOOP, VPC and DDISASM. We label spatial primitive searches that could use an inequality constraint as inequality primitive searches. Using this labelling approach, we can then consider the frequency of these inequality primitive searches in both a static and dynamic context.

We begin by inspecting the static distribution of spatial primitive searches as they appear in real-world Datalog programs. Specifically, how often do inequality primitive searches appear in these Datalog programs?

Program	**Equality Primitive Searches (%)**	**Inequality Primitive Searches (%)**
DOOP	99.4	0.6
VPC N-1075 (sec1)	98.7	1.3
VPC N-1075 (sec2)	98.9	1.1
VPC N-1075 (sec3)	98.7	1.3
VPC N-2340 (sec1)	98.4	1.6
VPC N-2340 (sec2)	98.7	1.3
VPC N-2340 (sec3)	98.3	1.7
VPC N-3500 (sec1)	98.8	1.2
VPC N-3500 (sec2)	98.9	1.1
VPC N-3500 (sec3)	98.7	1.3
VPC N-3511 (sec1)	98.8	1.2
VPC N-3511 (sec2)	98.9	1.1
VPC N-3511 (sec3)	98.7	1.3
VPC N-9087 (sec1)	98.8	1.2
VPC N-9087 (sec2)	98.9	1.1
VPC N-9087 (sec3)	98.7	1.3
DDISASM	96.9	3.1

TABLE 4.1. Static Distribution of Spatial Primitive Searches

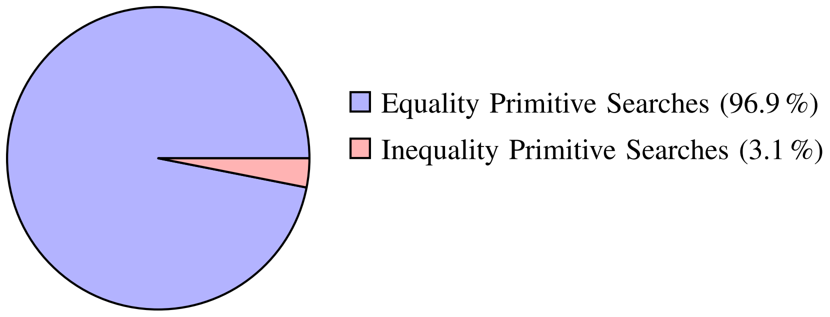


FIGURE 4.1. Static Distribution of Spatial Primitive Searches (DDISASM)

The table above details the distribution of spatial primitive searches in real-world programs. We find that inequality primitive searches are very infrequent, comprising at most 3.1% of the distribution of searches in real-world programs, with the highest frequency occurring in DDISASM.

Another helpful statistic is to inspect the attributes of every atom and see the distribution of constraints for each atom. The distribution divides into three types of constraints: no constraint, equality constraint or inequality constraint.

Program	**No Constraint (%)**	**Equality Constraint (%)**	**Inequality Constraint (%)**
DOOP	23.7	76.1	0.2
VPC N-1075 (sec1)	23.3	76.1	0.6
VPC N-1075 (sec2)	24.7	74.9	0.4
VPC N-1075 (sec3)	21.8	77.6	0.6
VPC N-2340 (sec1)	27.4	71.9	0.7
VPC N-2340 (sec2)	28.4	71.0	0.6
VPC N-2340 (sec3)	25.5	73.7	0.8
VPC N-3500 (sec1)	23.3	76.1	0.6
VPC N-3500 (sec2)	24.7	74.9	0.5
VPC N-3500 (sec3)	21.8	77.6	0.6
VPC N-3511 (sec1)	23.3	76.1	0.6
VPC N-3511 (sec2)	24.7	74.9	0.5
VPC N-3511 (sec3)	21.8	77.6	0.6
VPC N-9087 (sec1)	23.3	76.1	0.6
VPC N-9087 (sec2)	24.7	74.9	0.5
VPC N-9087 (sec3)	21.8	77.6	0.6
DDISASM	32.5	66.0	1.5

TABLE 4.2. Attribute Constraint Distribution in Spatial Primitive Searches

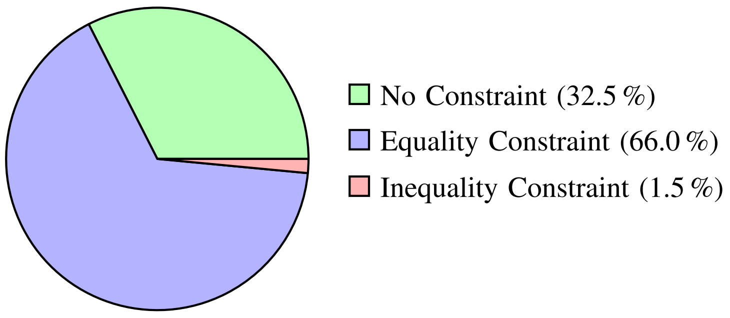


FIGURE 4.2. Distribution of Attribute Constraints in Spatial Primitive Searches (DDISASM)

Table 4.2 demonstrates how infrequently inequality constraints apply to attributes in searches. All programs have less than 1% of their attribute constraints as inequality constraints except for DDISASM (1.5%). Overall, it is quite clear that inequality primitive searches are very infrequent compared to equality primitive searches when looking at their composition in programs.

The final statistic that we use to analyse the static frequency of inequality primitive searches is their appearance in rules. We consider the proportion of rules that contain at least one inequality primitive search. By considering the frequency of rules with inequality primitive searches, we can gain further intuition for what proportion of programs are comprised by inequality primitive searches.

Program	**Rules without Inequality Primitive Searches (%)**	**Rules with Inequality Primitive Searches (%)**
DOOP	95.54	4.46
VPC N-1075 (sec1)	96.88	3.12
VPC N-1075 (sec2)	96.92	3.08
VPC N-1075 (sec3)	96.79	3.21
VPC N-2340 (sec1)	95.30	4.70
VPC N-2340 (sec2)	95.39	4.61
VPC N-2340 (sec3)	95.10	4.90
VPC N-3500 (sec1)	96.88	3.12
VPC N-3500 (sec2)	96.92	3.08
VPC N-3500 (sec3)	96.79	3.21
VPC N-3511 (sec1)	96.88	3.12
VPC N-3511 (sec2)	96.92	3.08
VPC N-3511 (sec3)	96.79	3.21
VPC N-9087 (sec1)	96.88	3.12
VPC N-9087 (sec2)	96.92	3.08
VPC N-9087 (sec3)	96.79	3.21
DDISASM	96.71	3.29

TABLE 4.3. Proportion of Rules with Inequality Primitive Searches

As can be seen from Table 4.3, we find that there are very few rules with inequality primitive searches, with VPC N-2340 having 4.90% of its rules containing at least one inequality primitive search. Overall, when considering the appearance of inequality primitive searches statically in real-world Datalog programs, they are very infrequent, being comprised almost exclusively by equality primitive searches instead.

We now shift our focus to the dynamic distribution of inequality primitive searches in Datalog programs. Specifically, how often do inequality primitive searches occur during evaluation time?

The first dynamic statistic to consider is the distribution of the total atom frequency that inequality primitive searches comprise. Within each rule, atoms are evaluated by performing spatial primitive searches and iterating through the tuples retrieved by each search, with each iteration over the atom in a rule, its atom frequency increases by one. The atom frequency statistics derive from the profiler tool provided with SOUFFLÉ, evaluated on real-world benchmarks run with different fact folders.

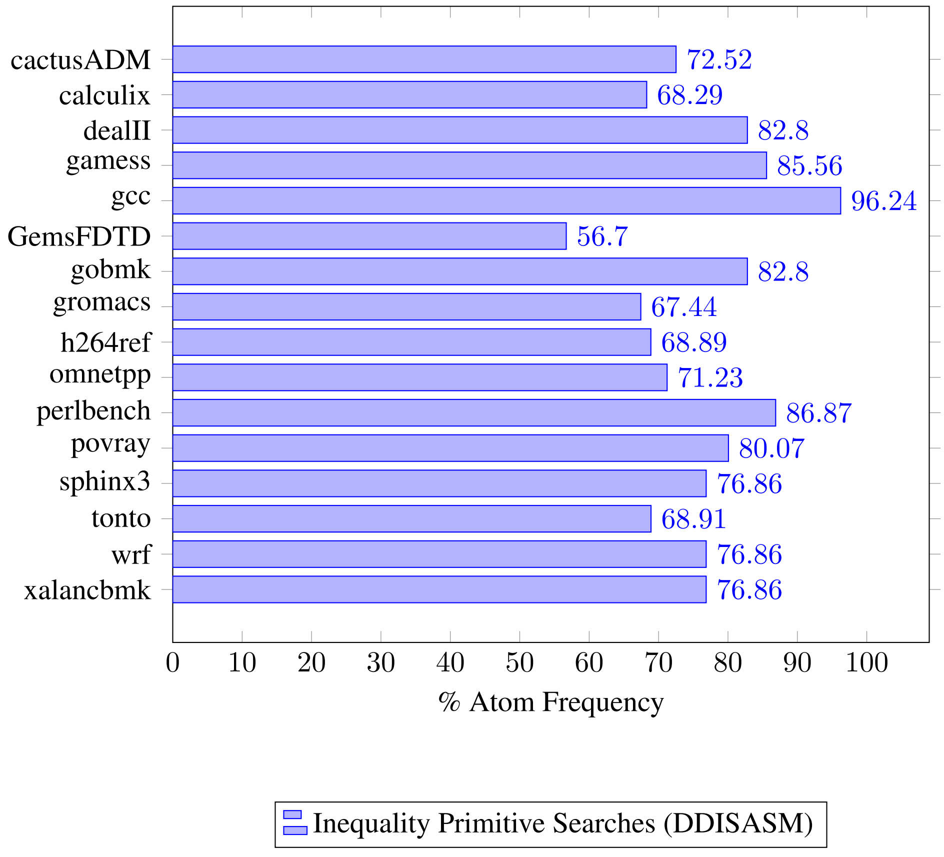
Fact Folder	**Equality Primitive Searches (%)**	**Inequality Primitive Searches (%)**
antlr	99.73	0.27
bloat	99.64	0.36
chart	99.75	0.25
eclipse	99.61	0.39
fop	99.73	0.27
hsqldb	99.61	0.39
jython	99.55	0.45
luindex	99.60	0.40
lusearch	99.60	0.40
pmd	99.71	0.29
xalan	99.71	0.29

TABLE 4.4. Dynamic Atom Frequency Distribution of Spatial Primitive Searches (DOOP)

Network and Analysis	**Equality Primitive Searches (%)**	**Inequality Primitive Searches (%)**
VPC N-1075 (sec1)	99.99983	0.00017
VPC N-1075 (sec2)	99.99985	0.00015
VPC N-1075 (sec3)	99.99983	0.00017
VPC N-2340 (sec1)	99.99950	0.00050
VPC N-2340 (sec2)	99.99950	0.00050
VPC N-2340 (sec3)	99.99950	0.00050
VPC N-3500 (sec1)	99.99999	0.00010
VPC N-3500 (sec2)	99.99999	0.00010
VPC N-3500 (sec3)	99.99999	0.00010
VPC N-3511 (sec1)	99.99999	0.00010
VPC N-3511 (sec2)	99.99999	0.00010
VPC N-3511 (sec3)	99.99999	0.00010
VPC N-9087 (sec1)	99.99999	0.00010
VPC N-9087 (sec2)	99.99999	0.00010
VPC N-9087 (sec3)	99.99999	0.00010

TABLE 4.5. Dynamic Atom Frequency Distribution of Spatial Primitive Searches (VPC)

Tables 4.4 and 4.5 indicate that for two of the three real-world benchmarks tested, the atom frequency of inequality primitive searches is below 0.1% of the total. Since atoms with inequality primitive searches occur very infrequently, the statistics indicate that inequality primitive searches do not comprise much of the evaluation time for either DOOP or VPC.



For DDISASM, we find that inequality primitive searches contribute significantly to the total atom frequency ranging from 56.7% up to 96.24% of the total frequency. Considering that inequality primitive searches only comprise 3.1% of all spatial primitive searches in DDISASM, the statistic indicates that inequality primitive searches are not evaluated efficiently compared to equality primitive searches.

Another useful dynamic statistic to explore is the proportion of the overall evaluation time consumed by inequality primitive searches. It is challenging to measure this directly since searches that appear at lower levels in the loop-nest execute more frequently and contribute more to the evaluation time than those positioned higher in the loop-nest. Therefore, we use a coarse-grained statistic by instead considering the distribution of the evaluation time contributed by each type of rule. We again divide rules into two groups: rules without inequality primitive searches and rules with inequality primitive searches.

Fact Folder	**Rules without Inequality Primitive Searches (%)**	**Rules with Inequality Primitive Searches (%)**
antlr	99.00	1.00
bloat	98.90	1.10
chart	99.07	0.93
eclipse	98.82	1.18
fop	99.00	1.00
hsqldb	98.71	1.29
jython	98.78	1.22
luindex	98.84	1.16
lusearch	98.88	1.12
pmd	99.01	0.99
xalan	98.93	1.07

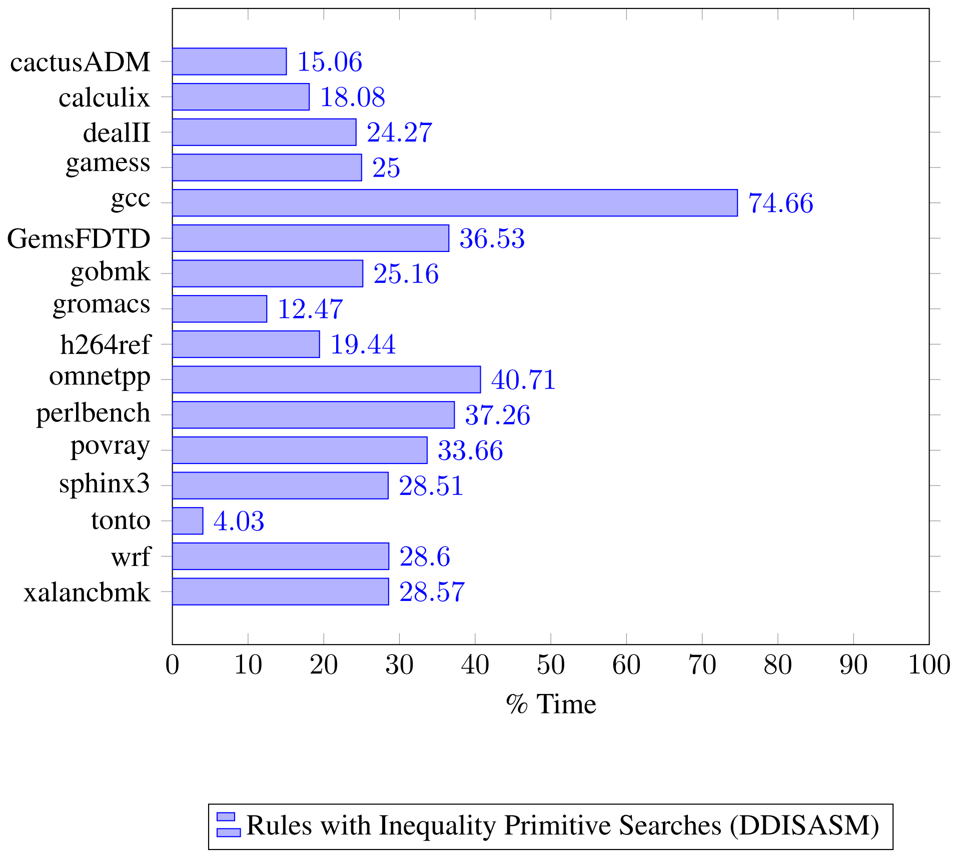
TABLE 4.6. Dynamic Run-Time Distribution of Rules (DOOP)

From Table 4.6, we can see that less than 1.29% of the evaluation time consists of rules containing inequality primitive searches. In other words, even if inequality primitive searches were instantaneous, the overall program evaluation time could only be reduced by 1.29%. DOOP, therefore, illustrates a scenario which does not stand to benefit significantly from indexing inequality primitive searches.

Network and Analysis	**Rules without Inequality Primitive Searches (%)**	**Rules with Inequality Primitive Searches (%)**
N-1075 sec1	99.99995	0.00005
N-1075 sec2	99.99995	0.00005
N-1075 sec3	99.99995	0.00005
N-2340 sec1	99.99991	0.00009
N-2340 sec2	99.99984	0.00016
N-2340 sec3	99.99991	0.00009
N-3500 sec1	99.99999	0.00001
N-3500 sec2	99.99999	0.00001
N-3500 sec3	99.99999	0.00001
N-3511 sec1	99.99999	0.00001
N-3511 sec2	99.99999	0.00001
N-3511 sec3	99.99999	0.00001
N-9087 sec1	99.99999	0.00001
N-9087 sec2	99.99999	0.00001
N-9087 sec3	99.99999	0.00001

TABLE 4.7. Dynamic Run-Time Distribution of Rules (VPC)

Table 4.7 illustrates an even less ideal scenario for indexed inequalities as less than one ten-thousandth of one per cent of the evaluation time is spent on rules with inequality primitive searches. Therefore, we expect zero evaluation time improvement from VPC.



The above figure indicates that a significant proportion of time in DDISASM is spent evaluating rules with inequality primitive searches. The proportion of the evaluation time spent on these rules ranges from 4.03% up to as high as 74.66%. Given that the number of rules with inequality primitive searches is only 3.29%, this indicates that inequality primitive searches are not evaluated efficiently.

Overall, the statistics gathered indicate that although inequality primitive searches occur relatively infrequently compared to equality primitive searches in Datalog programs viewed statically, there exist real-world benchmarks that spend a significant proportion of their evaluation time evaluating them. The evidence suggests that speeding up inequality primitive searches can dramatically reduce the time to evaluate such programs.

4.2 (Q2) R-Tree SPS vs Base

We have demonstrated that although inequality primitive searches are infrequent in Datalog programs, they can sometimes comprise a large portion of the evaluation time. Therefore, we now move on to evaluating our first proposed index selection strategy, R-Tree SPS. We consider R-Tree SPS compared to the current state of the art index selection technique, Base, evaluating its ability to speed up inequality primitive searches across a range of synthesised and real-world Datalog benchmarks.

$$
\begin{aligned}
&.decl\ \ natural(x : number)\\
&.input\ \ natural\\
\\
&.decl\ \ nearby\_naturals(x : number, y : number)\\
&.output\ \ nearby\_naturals\\
\\
&nearby\_naturals(x, y) \mathrel{:-} natural(x), natural(y),\\
&\qquad\qquad\qquad\qquad\quad x < y, y \leq x + 10.
\end{aligned}
$$

FIGURE 4.3. Nearby Naturals Program

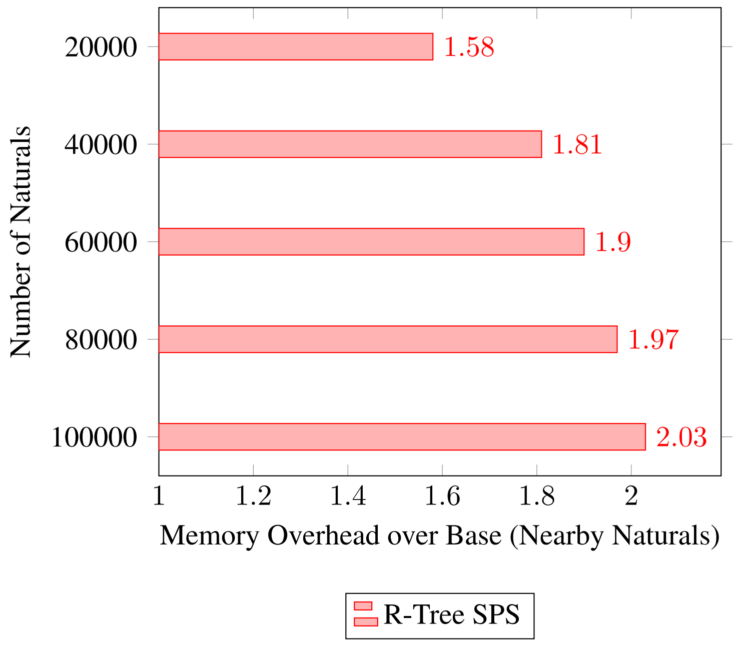
Index Selection Strategy	Number of B-Tree Indexes	Number of R-Tree Indexes
Base	2	0
R-Tree SPS	1	1

TABLE 4.8. Number of Indexes (Nearby Naturals)

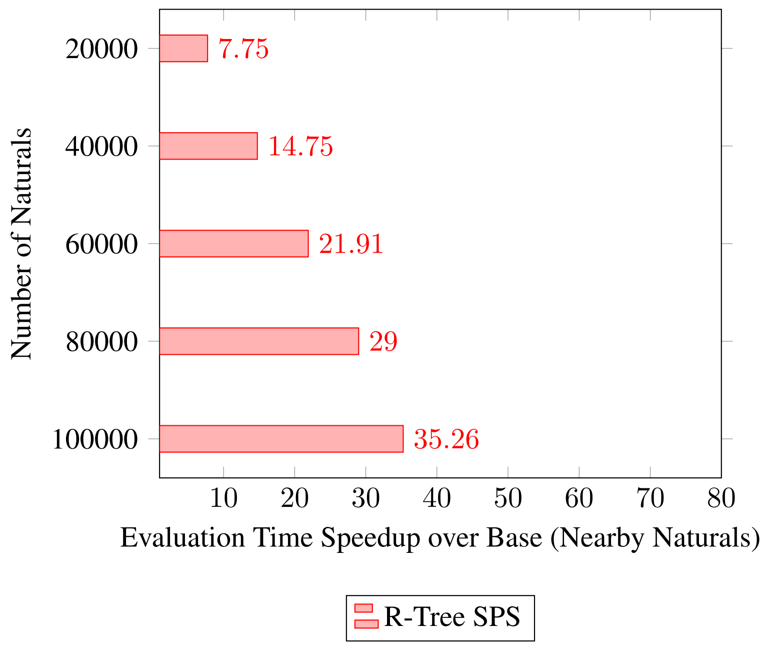
For our first synthesised benchmark, we use the Nearby Naturals program, where the inequality constraint on y heavily limits the number of satisfying tuples. Table 4.8 illustrates that both techniques construct the same number of indexes. R-Tree SPS uses an R-Tree index to cover the $natural$ relation since we apply an inequality primitive search on it to evaluate our search. For the other relation, $nearby\_naturals$, R-Tree SPS uses a B-Tree index since it has no inequality primitive searches. Base covers the two relations with a single B-Tree index each.

Number of Naturals	Memory Usage (Base)	Memory Usage (R-Tree SPS)	Evaluation Time (Base)	Evaluation Time (R-Tree SPS)
20000	8952	14168	0.62	0.08
40000	12856	23216	2.36	0.16
60000	17148	32576	5.26	0.24
80000	21164	41772	9.28	0.32
100000	25212	51120	14.46	0.41

Table 4.9. Maximum Resident Set Size (KB) and Evaluation Time (s) (Nearby Naturals)



The above figure illustrates the memory overhead of R-Tree SPS relative to Base. The R-Tree SPS strategy incurs a significant memory overhead, consuming $1.58\times$ to $2.03\times$ the memory of Base. In terms of memory consumption, Base is superior to R-Tree SPS.



The above figure demonstrates that the R-Tree SPS technique provides significant speedups ranging from a $7.75\times$ improvement to a $35.26\times$ improvement in evaluation time compared to the Base selection scheme. As the input size, n increases linearly; the speedup also increases almost linearly. The performance improvement is a natural consequence of the fact that using the inequality constraint in the indexed scan prunes the relation for satisfying tuples eagerly compared to Base which naïvely generates all pairs of naturals and filters them. Therefore, since evaluation time is most important, and R-Tree SPS delivers dramatic speedups over Base, we can accept a memory overhead of $2\times$ for the improved performance. Overall, R-Tree SPS significantly outperforms Base for this benchmark.

$$\begin{array}{l}
.decl\ \ point(x : number, y : number)\\
.input\ \ point\\
\\
.decl\ \ nearby\_points(x_1 : number, y_1 : number, x_2 : number, y_2 : number)\\
.output\ \ nearby\_points\\
\\
nearby\_points(x_1, y_1, x_2, y_2) :- point(x_1, y_1), point(x_2, y_2),\\
\qquad\qquad\qquad\qquad\qquad\qquad x_1 < x_2, x_2 \leq x_1 + 10,\\
\qquad\qquad\qquad\qquad\qquad\qquad y_1 < y_2, y_2 \leq y_1 + 10,
\end{array}$$

FIGURE 4.4. Nearby Points Program

The Nearby Points program generalises the previous synthesised benchmark to two dimensions. A number of points n are generated in the range $[0, \sqrt{n}] \times [0, \sqrt{n}]$ and pairs of points whose x and y coordinates both differ by at most $k = 10$ are queried. The benchmark uses the ability of an R-Tree index to natively performing searches with any number of attributes with inequality constraints.

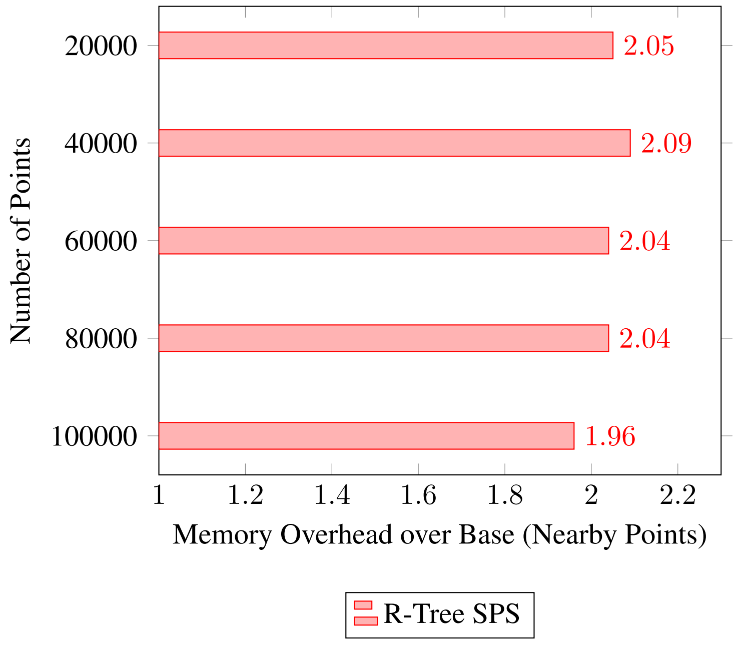
Index Selection Strategy	**Number of B-Tree Indexes**	**Number of R-Tree Indexes**
Base	2	0
R-Tree SPS	1	1

TABLE 4.10. Number of Indexes (Nearby Points)

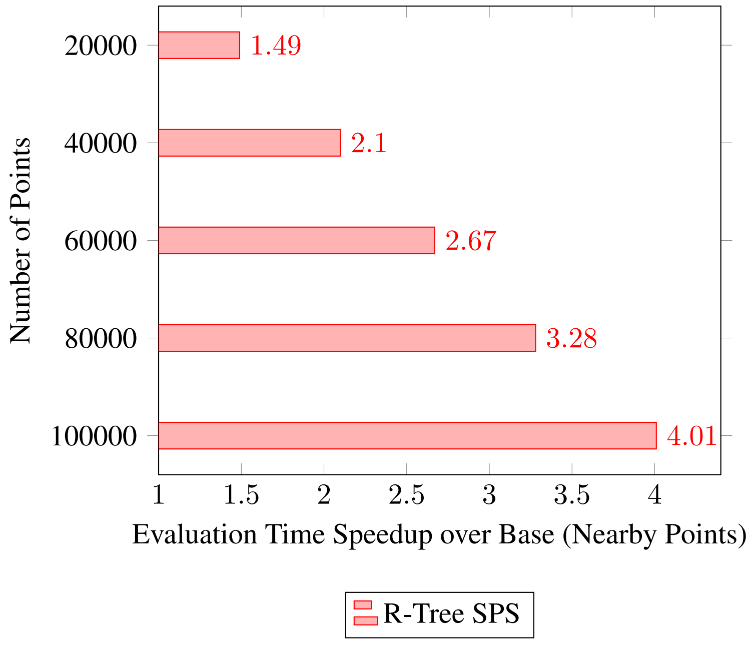
As with the previous benchmark, Table 4.10 showcases that R-Tree SPS uses an R-Tree index to cover the $point$ relation. R-Tree SPS constructs an R-Tree index as an inequality primitive search applies to $point$. The other relation has no inequality primitive searches, so a B-Tree index covers the other relation. We note that this is possible since the R-Tree index can natively support any number of attributes with inequality constraints in a spatial primitive search and can cover all spatial primitive searches of a relation with a single R-Tree index.

Number of Points	**Memory Usage (Base)**	**Memory Usage (R-Tree SPS)**	**Evaluation Time (Base)**	**Evaluation Time (R-Tree SPS)**
20000	78100	160108	1.36	0.91
40000	154900	323932	3.93	1.87
60000	229872	468128	7.64	2.86
80000	307340	626280	12.58	3.83
100000	385888	757156	19.01	4.73

TABLE 4.11. Maximum Resident Set Size (KB) and Evaluation Time (s) (Nearby Points)



The above figure demonstrates again approximately a $2\times$ memory overhead of R-Tree SPS over Base.



The above figure showcases similar behaviour to the previous benchmark with speedup almost linearly increasing as the input size n grows linearly (ranging from $1.49\times$ to $4.01\times$). The R-Tree SPS strategy again proves to be better than Base, dramatically improving the evaluation speed even with a memory overhead of $2\times$.

$$
\begin{array}{l}
.decl\ \ employee(name : symbol, salary : number, tax : number) \\
.input\ \ employee \\
\\
.decl\ \ tax\_fraud(name_1 : symbol, name_2 : symbol) \\
.output\ \ tax\_fraud \\
\\
tax\_fraud(name_1, name_2) :- employee(name_1, salary_1, tax_1), \\
\qquad\qquad\qquad\qquad\qquad\quad employee(name_2, salary_2, tax_2), \\
\qquad\qquad\qquad\qquad\qquad\quad salary_2 > salary_1, tax_2 < tax_1.
\end{array}
$$

FIGURE 4.5. Tax Program

The next synthesised benchmark represents a best-case scenario for the R-Tree strategy. For an experimental data set, we generate employees earning a salary in the range $30,000$ to $30,000 + n$ that pay 30% of their salary as tax. We then introduce a single anomalous record of an employee committing tax fraud.

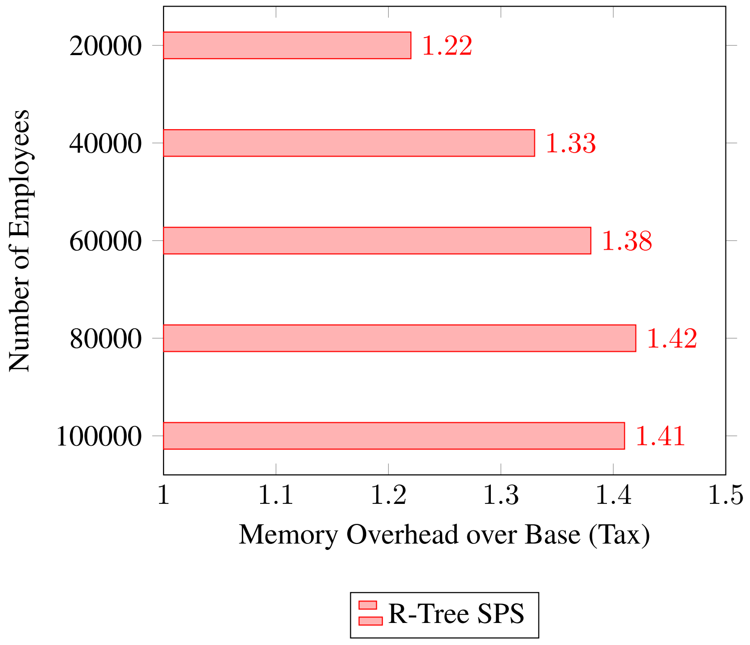
Index Selection Strategy	**Number of B-Tree Indexes**	**Number of R-Tree Indexes**
Base	2	0
R-Tree SPS	1	1

TABLE 4.12. Number of Indexes (Tax)

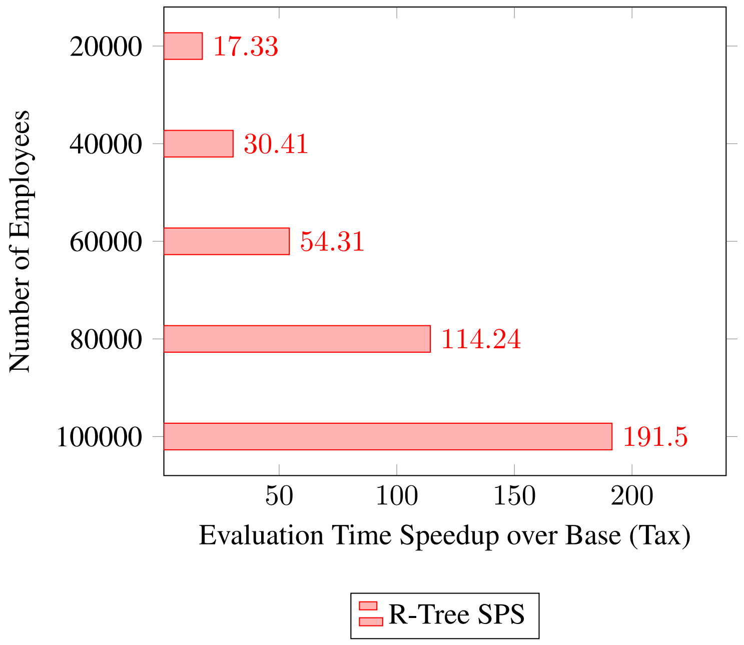
Table 4.14 shows that exactly like the previous two benchmarks, an R-Tree index covers the relation queried with inequality constraints and the B-Tree index covers the other.

Number of Employees	**Memory Usage (Base)**	**Memory Usage (R-Tree SPS)**	**Evaluation Time (Base)**	**Evaluation Time (R-Tree SPS)**
20000	7640	9284	1.04	0.06
40000	10184	13572	3.65	0.12
60000	13040	17956	10.32	0.19
80000	15776	22368	28.56	0.25
100000	19140	27060	61.28	0.32

TABLE 4.13. Maximum Resident Set Size (KB) and Evaluation Time (s) (Tax)



The figure above illustrates the least memory overhead for R-Tree SPS using between 22% and 42% more memory than the original scheme. The relative lack of memory overhead for R-Tree SPS in this benchmark compared to Base is likely because less filtering of tuples is done in memory, as irrelevant tuples are not visited during the search over the R-Tree index. Overall, Base is still superior in terms of memory consumption.



The above figure demonstrates a scenario where R-Tree SPS performs outstandingly well compared to Base providing an evaluation time speedup ranging from $17.33\times$ to $191.5\times$ the evaluation time of Base. Since R-Trees can use both inequality constraints in the indexed scan, they can evaluate the search dramatically faster than Base, which naïvely generates all pairs of employees and filters them on the inequality constraints. As the input size increases, the speedup increases linearly for the R-Tree index. The R-Tree SPS scheme is much better than the Base scheme, outperforming it by orders of magnitude with at most a 41% increase in memory consumption.

$$
\begin{array}{l}
.decl\ \ A(x : number, y : number, z : number) \\
.input\ \ A \\
.printsize\ \ A \\
\\
A(0,0,0) :\!-\ A(x,\_,\_), x > 0. \\
A(0,0,0) :\!-\ A(\_,y,\_), y > 0. \\
A(0,0,0) :\!-\ A(\_,\_,z), z > 0.
\end{array}
$$

FIGURE 4.6. Insert Program with Arity $n = 3$

The Insert program is our final synthesised benchmark. The relation A has arity n with inequality primitive searches on each attribute. A fixed set of $10,000,000$ shuffled facts for A in the range $[0, (10,000,000)^{1/n}]^n$ are inserted into the relation to illustrate the insertion performance of each technique.

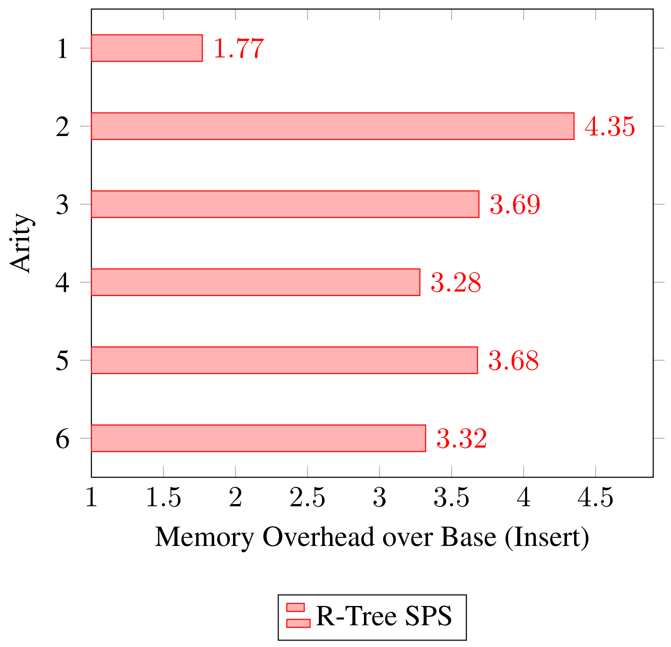
Index Selection Strategy	**Number of B-Tree Indexes**	**Number of R-Tree Indexes**
Base	3	0
R-Tree SPS	0	3

TABLE 4.14. Number of Indexes (Insert)

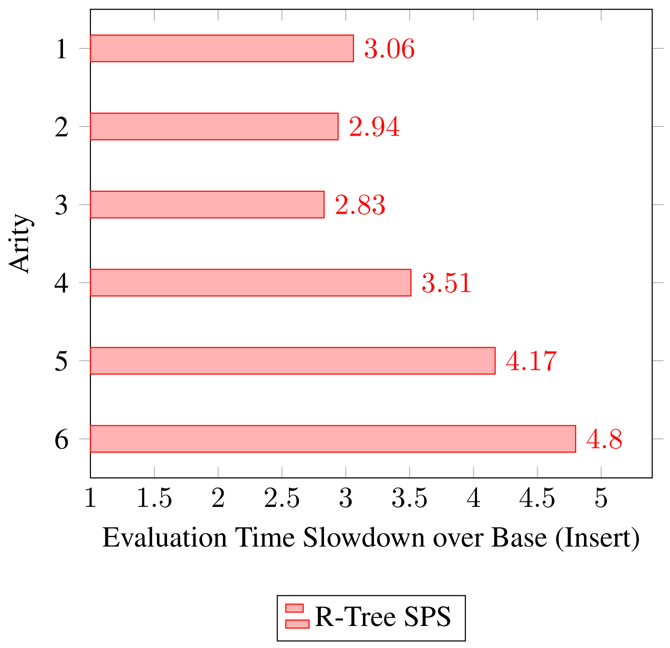
Due to the semi-naïve evaluation strategy, when recursive rules occur in the program, 3 indexes must be constructed for the relation, A, $delta_A$ and new_A. We note that new_A only ever contains IDB tuples, therefore the overall storage requirement is $10,000,000$ tuples on each of A and $delta_A$.

Arity	Memory Usage (Base)	Memory Usage (R-Tree SPS)	Evaluation Time (Base)	Evaluation Time (R-Tree SPS)
1	239836	425272	6.69	20.46
2	484584	2109544	10.06	29.60
3	721104	2658072	12.99	36.72
4	1108120	3631748	16.92	59.41
5	1464816	4920528	23.35	97.40
6	1698144	5634816	24.77	118.91

TABLE 4.15. Maximum Resident Set Size (KB) and Evaluation Time (s) (Insert)



The figure illustrates the memory overhead of R-Tree SPS compared to Base. We observe an overhead ranging from 1.77× to 4.35×. We can attribute this to two main factors. Firstly, R-Trees have roughly a 2× memory overhead compared to B-Tree indexes and secondly, the process of forcefully reinserting tuples with the R* splitting technique materialises more tuples into memory.



We observe in the above figure, a slowdown ranging from $2.83\times$ to $4.80\times$ that appears to increase as the arity increases. The observation makes sense given that as the arity increases, the R-Tree has more difficulty placing tuples into sub-trees without introducing overlap. With more overlap, more splitting occurs causing further re-insertions at the top of the tree, degrading performance. Overall, the R-Tree SPS strategy is worse than Base here, evaluation time slowdown of up to $4.8\times$ as well as a significant memory overhead of over $4\times$.

Our synthesised benchmarks have showcased a dramatic improvement in evaluation time over Base for a number of synthesised benchmarks with the R-Tree SPS technique. However, it is essential that our technique can accelerate the evaluation time performance of real-world programs when selective inequality constraints are present and not hamper performance when less selective inequality constraints are present. Therefore, we now shift attention to three real-world benchmarks: DOOP (Antoniadis et al., 2017; Bravenboer and Smaragdakis, 2009), Amazon's Virtual Private Cloud (VPC) Network Security Analysis and the Datalog Disassembler (DDISASM) (Antoniadis et al., 2017).

Index Selection Strategy	Number of B-Tree Indexes	Number of R-Tree Indexes	Compilation Time
Base	594	0	140.8
R-Tree SPS	585	8	147.7

TABLE 4.16. Number of Indexes and Compilation Time (s) (DOOP)

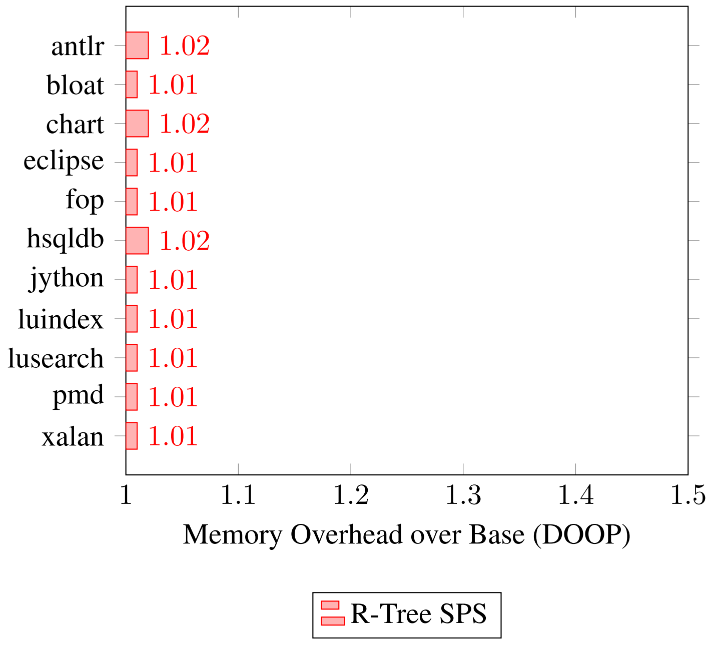
We observe that for DOOP, there are 585 B-Tree indexes and 8 R-Tree indexes constructed by R-Tree SPS. The lack of R-Tree indexes indicates that very few relations have inequality primitive searches. We note that the total number of indexes used by R-Tree SPS is one less than Base, indicating that a single R-Tree index replaced a relation which Base would cover with two B-Tree indexes.

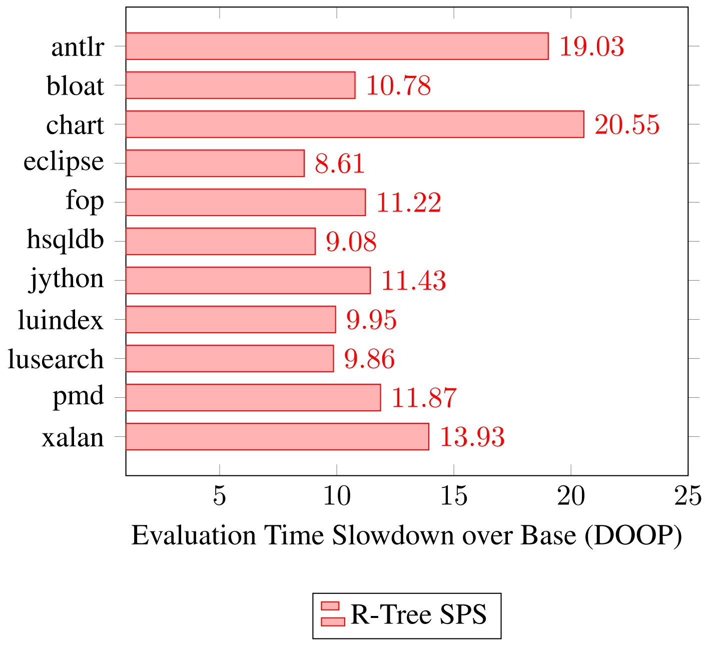
We observe a 5% increase in the overall compilation time for DOOP using R-Tree SPS. Boost's R-Tree index relies heavily on the evaluation of many layers of templates, constructing the index with template classes provided by other Boost dependencies. Therefore, R-Trees add to the compilation time of the generated program and add time to the linking stage for the binary to resolve these extra dependencies. Overall a 5% overhead in compilation time is not very significant since programs are infrequently compiled and once compiled, run frequently with different inputs.

Fact Folder	**Memory Usage (Base)**	**Memory Usage (R-Tree SPS)**	**Evaluation Time (Base)**	**Evaluation Time (R-Tree SPS)**
antlr	2046448	2078336	40.81	776.81
bloat	1445884	1464600	27.33	294.63
chart	2149824	2182168	42.85	880.39
eclipse	1443628	1461236	29.31	252.29
fop	2120208	2151592	41.75	468.3
hsqldb	2189184	2223080	44.41	403.04
jython	1627420	1649008	31.53	360.31
luindex	1443508	1461864	27.83	277.00
lusearch	1441388	1459636	27.85	274.61
pmd	2089748	2120232	41.61	493.73
xalan	2140004	2170860	43.06	599.98

TABLE 4.17. Maximum Resident Set Size (KB) and Evaluation Time (s) (DOOP)

The figure above highlights that R-Tree SPS has an insignificant memory overhead compared to the original scheme. This indicates that the few relations which have inequality primitive searches do not contain a large number of tuples, otherwise the approximate $2\times$ memory overhead of maintaining an R-Tree index would be visible.





The figure illustrates the slowdown of the R-Tree SPS technique as it degrades performance heavily in DOOP compared to Base. The technique slows down the evaluation time by $8.61\times$ to $20.55\times$. In the worst-case, searching an R-Tree takes time proportional to the size of the relation. B-Tree indexes, by comparison, are specialised for each search, consuming time proportional to the size of the output of the search. Therefore, when the relation sizes are large, and the output size of searches is small, the performance of the R-Tree is significantly worse than a B-Tree index.

Given that the R* splitting strategy organises the data within the R-Tree effectively, the poor search performance indicates that the performance limitations are inherent to the R-Tree data structure. The performance of indexed searches is most crucial when the loop schedule is optimally selected, as is the case in DOOP, the effect of the R-Tree's slow evaluation of these indexed searches becomes apparent. DOOP is sufficient evidence to conclude that the R-Tree SPS strategy is not robust, causing dramatic slowdown if the R-Tree covers crucial spatial primitive searches. Overall, Base is far superior to R-Tree SPS as it is roughly an order of magnitude faster across all fact folders for DOOP.

Index Selection Strategy	**Number of B-Tree Indexes**	**Number of R-Tree Indexes**
Base	304	0
R-Tree SPS	302	1

TABLE 4.18. Number of Indexes (VPC N-1075 sec1)

Index Selection Strategy	**Number of B-Tree Indexes**	**Number of R-Tree Indexes**
Base	315	0
R-Tree SPS	313	1

TABLE 4.19. Number of Indexes (VPC N-1075 sec2)

Index Selection Strategy	**Number of B-Tree Indexes**	**Number of R-Tree Indexes**
Base	298	0
R-Tree SPS	296	1

TABLE 4.20. Number of Indexes (VPC N-1075 sec3)

Tables 4.18, 4.19 and 4.20 showcase the number of constructed indexes for each technique. Across all analyses (sec1, sec2 and sec3), only a single R-Tree index is constructed by R-Tree SPS, indicating that inequality primitive searches are extremely sparse in VPC. We observed the same behaviour across VPC-2340, VPC-3500, VPC-3511 and VPC-9087, so we omit these tables for brevity. Overall, we expect little to no effect on VPC from using the R-Tree SPS strategy.

Program	Compilation Time (Base)	Compilation Time (R-Tree SPS)	Compilation Time Overhead over Base (Overhead over Base)
VPC N-1075 sec1	127.1	130.0	1.00
VPC N-1075 sec2	134.6	136.0	1.01
VPC N-1075 sec3	126.0	129.0	1.02
VPC N-2340 sec1	52.64	55.20	1.05
VPC N-2340 sec2	55.38	60.57	1.09
VPC N-2340 sec3	52.04	54.93	1.06
VPC N-3500 sec1	128.54	132.45	1.03
VPC N-3500 sec2	134.98	137.88	1.02
VPC N-3500 sec3	128.67	130.98	1.02
VPC N-3511 sec1	127.98	136.88	1.07
VPC N-3511 sec2	132.49	138.11	1.04
VPC N-3511 sec3	136.95	139.86	1.02
VPC N-9087 sec1	127.59	130.28	1.02
VPC N-9087 sec2	133.14	136.51	1.03
VPC N-9087 sec3	128.14	131.62	1.03

TABLE 4.21. Compilation Time (s) of Index Selection Schemes (VPC)

The above figure illustrates the compilation time overhead of R-Tree SPS relative to Base. We find that the overhead is at most an added 9% to the overall compilation time. We can attribute this to the extra time to compile the single R-Tree index for VPC. Overall, a compilation time overhead of 9% is noticeable but not significant for the R-Tree SPS strategy.

Network and Analysis	Memory Usage (Base)	Memory Usage (R-Tree SPS)	Evaluation Time (Base)	Evaluation Time (R-Tree SPS)
VPC N-1075 (sec1)	1631804	1632692	150.97	154.01
VPC N-1075 (sec2)	1962056	1962732	137.55	139.62
VPC N-1075 (sec3)	4195416	4195884	168.91	169.07
VPC N-2340 (sec1)	727508	727980	202.76	210.40
VPC N-2340 (sec2)	727628	727988	198.84	201.37
VPC N-2340 (sec3)	727060	727476	207.15	196.67
VPC N-3500 (sec1)	7706920	7707700	1862.17	1826.11
VPC N-3500 (sec2)	14359900	14360692	1605.51	1588.42
VPC N-3500 (sec3)	7497628	7498320	1619.32	1632.57
VPC N-3511 (sec1)	7706952	7707644	1846.28	1822.34
VPC N-3511 (sec2)	14359824	14360724	1618.12	1579.51
VPC N-3511 (sec3)	7497612	7498472	1639.53	1632.97
VPC N-9087 (sec1)	6506056	6506904	1814.89	1790.38
VPC N-9087 (sec2)	11633248	11634224	1557.48	1522.85
VPC N-9087 (sec3)	7567900	7568980	1587.98	1593.71

TABLE 4.22. Maximum Resident Set Size (KB) and Evaluation Time (s) (VPC)

Table 4.22 details the memory usage of the analyses with each index selection scheme. We observe that across all networks, the memory overhead of of the new strategy is less than 1%. We can conclude that the R-Tree index constructed to cover the inequality primitive searches in VPC does not contain many tuples. Overall, since the memory overhead is not significant, there is no harm in adopting the R-Tree SPS strategy over Base.

The table also showcases the absolute evaluation time of the network security analysis for all three analyses across multiple networks. We find that the R-Tree SPS strategy has little effect on the overall evaluation time. We observe a speed-up of up to 5% with R-Tree SPS as well as slowdowns of up to 4%. Since a domain-specific language generates the Datalog program for each network analysis, poorly scheduled loop nests are responsible for the majority of the evaluation time. Furthermore, there are few inequality constraints in VPC, resulting in only a single constructed R-Tree index. Therefore, the different indexing strategies do not significantly impact evaluation time. Overall, Base offers more consistent performance than R-Tree SPS and is the better strategy for VPC.

Index Selection Strategy	Number of B-Tree Indexes	Number of R-Tree Indexes	Compilation Time
Base	692	0	178.7
R-Tree SPS	638	34	187.7

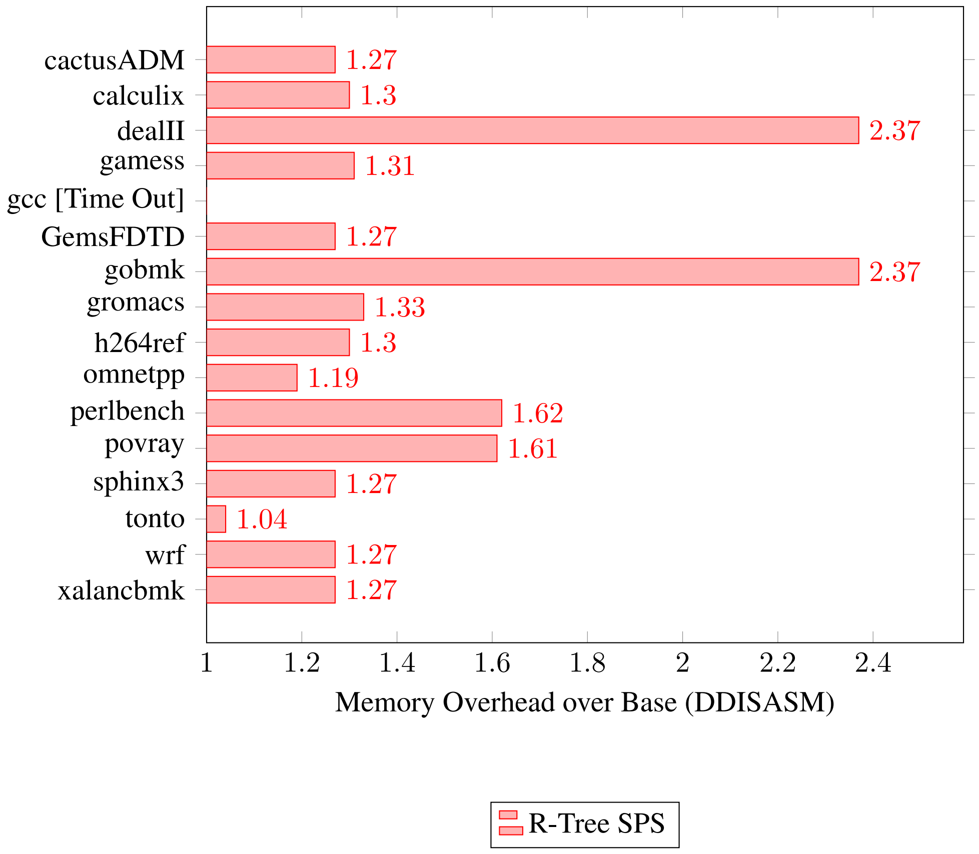
TABLE 4.23. Number of Indexes and Compilation Time (s) (DDISASM)

Unlike previous benchmarks, Table 4.23, makes clear that R-Tree indexes cover a much more significant number of relations in DDISASM. We can attribute this to the larger number of inequality primitive searches present in DDISASM compared to other benchmarks. Therefore, the effect of the R-Tree SPS strategy should be very apparent compared to previous benchmarks. Interestingly, the total number of indexes constructed by R-Tree SPS is 20 less than Base. We can attribute this difference to the fact that each individual R-Tree index replaces a cluster of B-Tree indexes containing multiple B-Trees.

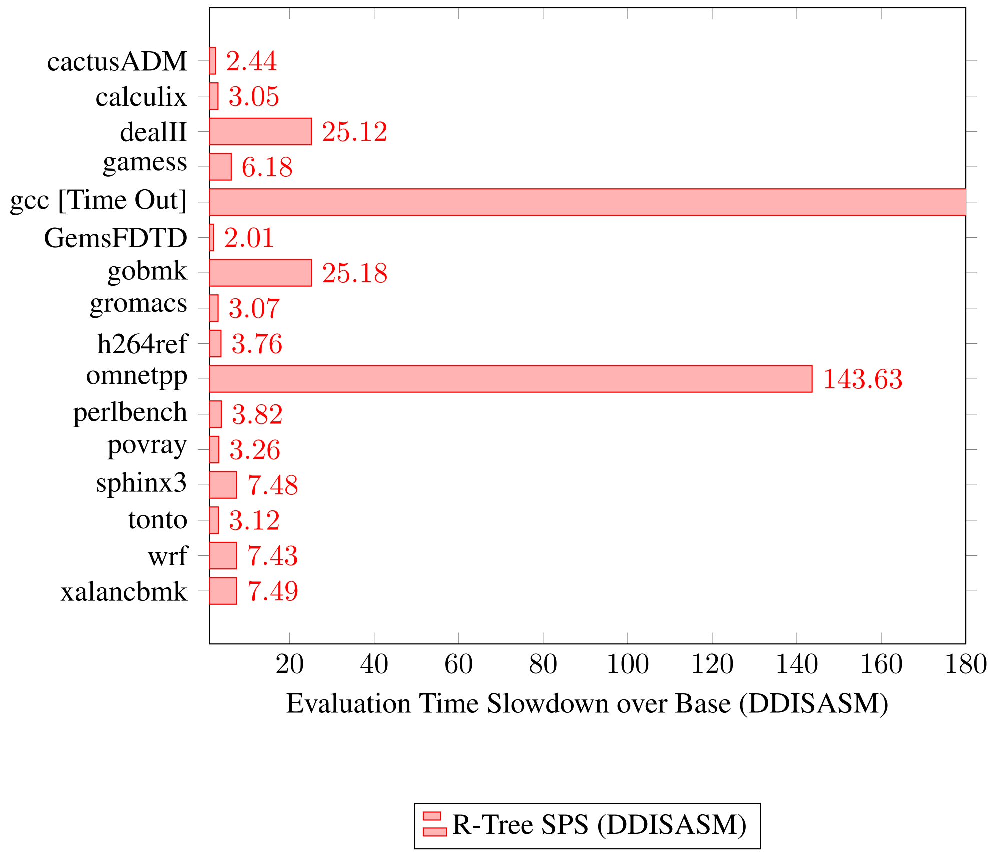
We find by Table 4.23 that there is a 5% increase in compilation time using the R-Tree SPS strategy compared to Base. As with DOOP, we can explain this difference as the time required to synthesise the 34 R-Tree indexes, compile the generated R-Tree indexes in the C++ source and link against Boost's dependencies. An overhead of 5% is not considered significant for the R-Tree since users compile Datalog programs infrequently and execute them with different inputs frequently.

Fact Folder	Memory Usage (Base)	Memory Usage (R-Tree SPS)	Evaluation Time (Base)	Evaluation Time (R-Tree SPS)
cactusADM	445400	564808	10.21	24.87
calculix	1004012	1304052	26.61	81.05
dealII	609620	1447632	47.71	1198.62
gamess	5286504	6899952	160.2	990.38
gcc	2595832	1471208	177.88	Time Out
GemsFDTD	278572	352816	9.38	18.9
gobmk	609556	1446812	47.55	1197.16
gromacs	569472	756468	13.34	40.96
h264ref	330768	428424	9.05	34.07
omnetpp	346472	410600	16.12	2315.31
perlbench	1328780	2158668	25.79	98.62
povray	1400756	2249388	19.07	62.10
sphinx3	2795504	3562080	81.17	607.27
tonto	22435464	23417856	522.41	1628.63
wrf	2796084	3562240	81.40	605.02
xalancbmk	2795532	3562180	80.70	604.41

TABLE 4.24. Maximum Resident Set Size (KB) and Evaluation Time (s) (DDISASM)



The figure highlights the memory overhead of R-Tree SPS for DDISASM. We find that the indexing strategy results in a memory overhead ranging from an additional 4% to up to a 137% increase in memory usage. Considering that less than 5% of all indexes are R-Tree indexes, this indicates that these few indexes contain large numbers of tuples. Overall, the additional memory overhead is fairly significant here, and the Base strategy would be preferable unless R-Tree SPS can deliver considerable speedups in evaluation time.



The figure illustrates the slowdown of the R-Tree index for the DDISASM benchmark. The slowdown ranges from 2.01× to 143.63× in the worst case and timing out for the GCC benchmark (exceeding 3 hours). As with DOOP, the R-Tree SPS strategy does not accelerate performance, but instead it consistently degrades performance. Even ignoring the 2× memory overhead, R-Tree SPS is utterly infeasible for real-world benchmarks, causing orders of magnitude levels of a slowdown compared to Base.

Overall, even though R-Tree SPS shows considerable speed-ups for synthesised benchmarks, it results in dramatic slowdowns in real-world benchmarks, including both DOOP and DDISASM. Therefore, Base is far superior to R-Tree SPS as an index selection strategy for the evaluation of real-world Datalog programs.

4.3 (Q3) B-Tree SPS vs Base

In the previous section, we concluded that R-Tree SPS failed to accelerate the performance of inequality primitive searches in real-world programs compared to Base. The poor evaluation performance is a consequence of the lack of specialisation of R-Tree indexes, resulting in searches that may consume time proportional to the size of the relation. Therefore, we now explore B-Tree SPS as an alternative index selection strategy, using B-Tree indexes which offer strong search performance, bounding the search complexity by the size of the search output.

Index Selection Strategy	Number of B-Tree Indexes
Base	2
B-Tree SPS	2

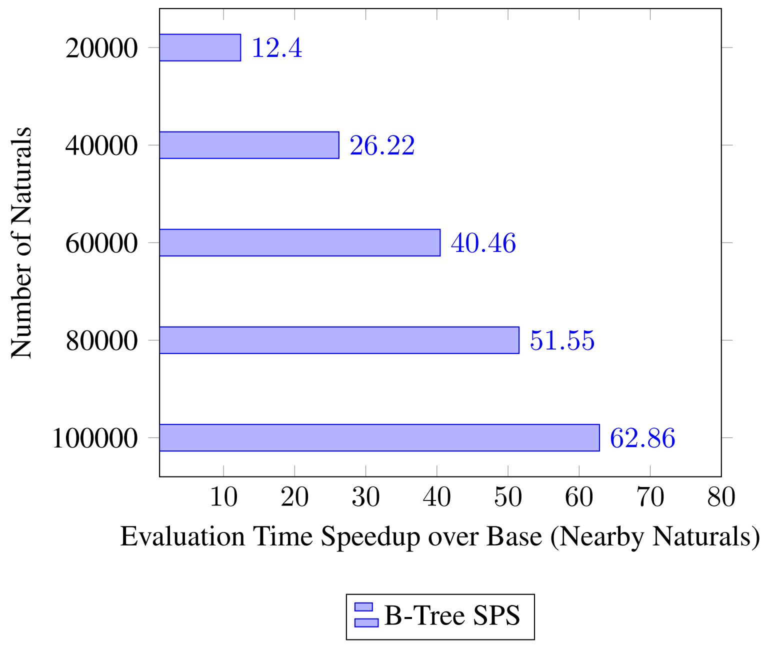
TABLE 4.25. Number of B-Tree Indexes (Nearby Naturals)

For the Nearby Naturals benchmark, Table 4.25, showcases that both B-Tree SPS and Base, construct a single B-Tree index to cover each relation. Since, the B-Tree SPS index selection technique is efficient at packing many spatial primitive searches into long search chains, even though there are extra spatial primitive searches to be indexed, no extra B-Trees are required to cover them. Therefore, since the number of B-Tree indexes for each technique is identical, we expect zero compilation time or memory overhead of B-Tree SPS compared to Base.

Number of Naturals	Memory Usage (Base)	Memory Usage (B-Tree SPS)	Evaluation Time (Base)	Evaluation Time (B-Tree SPS)
20000	8952	8952	0.62	0.05
40000	12856	12980	2.36	0.09
60000	17148	16968	5.26	0.13
80000	21164	21312	9.28	0.18
100000	25212	25156	14.46	0.23

TABLE 4.26. Maximum Resident Set Size (KB) and Evaluation Time (s) (Nearby Naturals)

The table above details the memory usage of B-Tree SPS and Base. We find that B-Tree SPS shows zero memory overhead up to experimental error (within 1%). The lack of memory overhead is intuitive since no extra B-Trees need to be constructed by B-Tree SPS to cover the new inequality primitive searches (as seen in Table 4.25).



The figure above showcases the B-Tree SPS technique's significant speedups of $12.40\times$ to $62.86\times$ over the Base selection scheme. By using the inequality constraint in the indexed scan, the B-Tree SPS technique dramatically improves performance with the evaluation time speedup over Base doubling as the input size doubles. We can attribute this dramatic speedup with B-Tree SPS to the eager pruning of the relation's tuples with inequality constraints as opposed to Base's naïve generate and test approach. Overall, B-Tree SPS is superior to Base as it is dramatically faster in terms of evaluation time and requires zero extra memory consumption.

Index Selection Strategy	**Number of B-Tree Indexes**
Base	2
B-Tree SPS	2

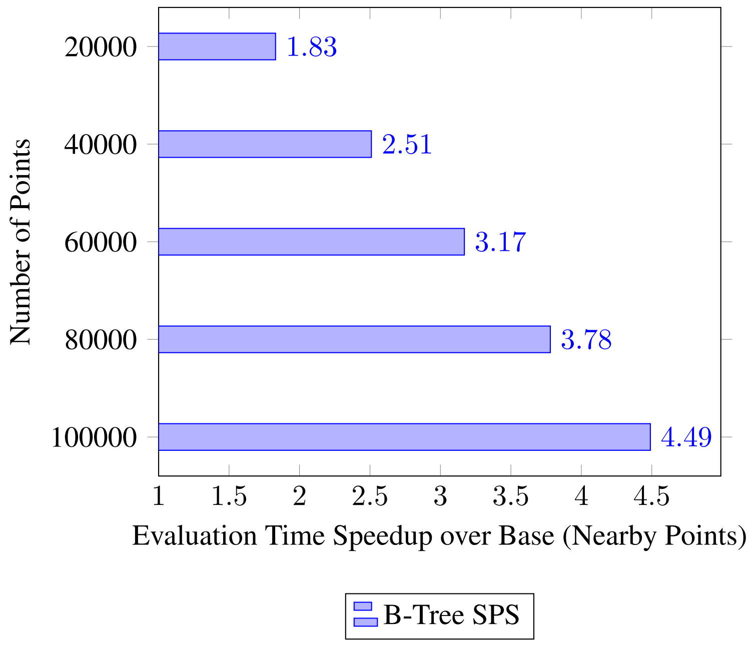
TABLE 4.27. Number of Indexes (Nearby Points)

As with the previous benchmark, B-Tree SPS uses the same number of B-Tree indexes to cover the extra inequality primitive searches present in the benchmark.

Number of Points	**Memory Usage (Base)**	**Memory Usage (B-Tree SPS)**	**Evaluation Time (Base)**	**Evaluation Time (B-Tree SPS)**
20000	78100	78140	1.36	0.74
40000	154900	154956	3.93	1.56
60000	229872	230128	7.64	2.41
80000	307340	307236	12.58	3.32
100000	385888	385676	19.01	4.23

TABLE 4.28. Maximum Resident Set Size (KB) and Evaluation Time (s) (Nearby Points)

As with the previous benchmark, since the same number of B-Tree indexes are constructed for both B-Tree SPS and Base, we observe effectively no difference in memory usage between the two techniques.



Much like the previous benchmark, we observe tremendous speedups over Base, ranging from 1.83× to 4.49×. The speedup increases further as the input size increases. Ultimately, B-Tree SPS is the clear choice compared to Base, consistently outperforming Base by large margins in evaluation time with effectively zero memory overhead.

Index Selection Strategy	**Number of B-Tree Indexes**
Base	2
B-Tree SPS	2

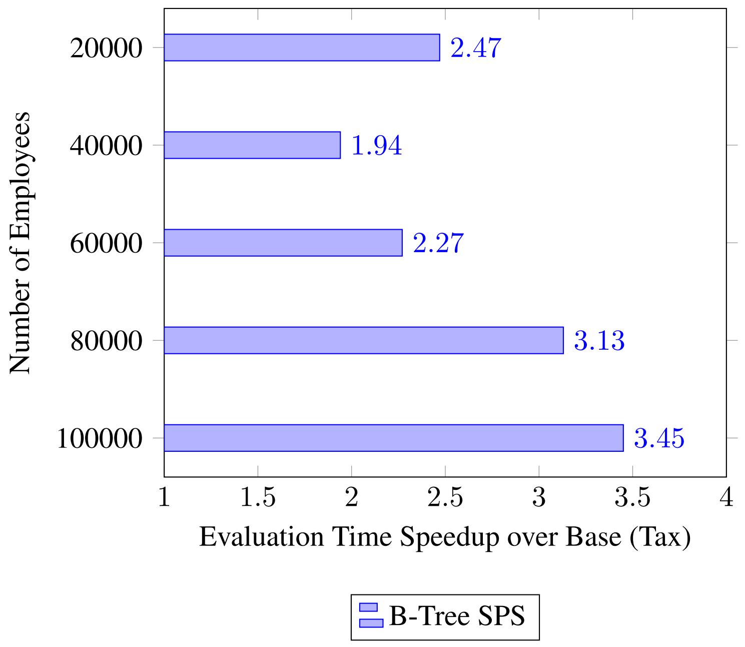
TABLE 4.29. Number of Indexes (Tax)

As before, the number of B-Tree indexes constructed by B-Tree SPS is identical to that of Base, since a single search chain can cover all simple spatial primitive searches.

Number of Employees	**Memory Usage (Base)**	**Memory Usage (B-Tree SPS)**	**Evaluation Time (Base)**	**Evaluation Time (B-Tree SPS)**
20000	7640	7556	1.04	0.42
40000	10184	10272	3.65	1.88
60000	13040	13328	10.32	4.54
80000	15776	15964	28.56	9.10
100000	19140	19324	61.28	17.76

TABLE 4.30. Maximum Resident Set Size (KB) and Evaluation Time (s) (Tax)

Much like the previous benchmarks, there is zero memory overhead up to experimental error for the B-Tree SPS strategy since no new indexes need to be constructed relative to Base.



The above figure demonstrates a scenario where B-Tree SPS can accelerate performance fairly significantly compared to Base, but not as dramatically as previous benchmarks. B-Tree indexes cannot natively evaluate searches multiple attributes with inequality constraints and therefore, the indexed scan can only compute efficiently, employees with a greater salary or a lower tax but not both. The resultant set of pairs is then filtered manually. Since each inequality constraint individually is not very selective, less pruning of the search space can be done by the B-Tree SPS technique, with the complexity of the evaluation being $\mathcal{O}(n^2)$. However, as with prior benchmarks the B-Tree SPS benchmark is far superior to Base, still outperforming the Base strategy by significant margins with zero memory overhead.

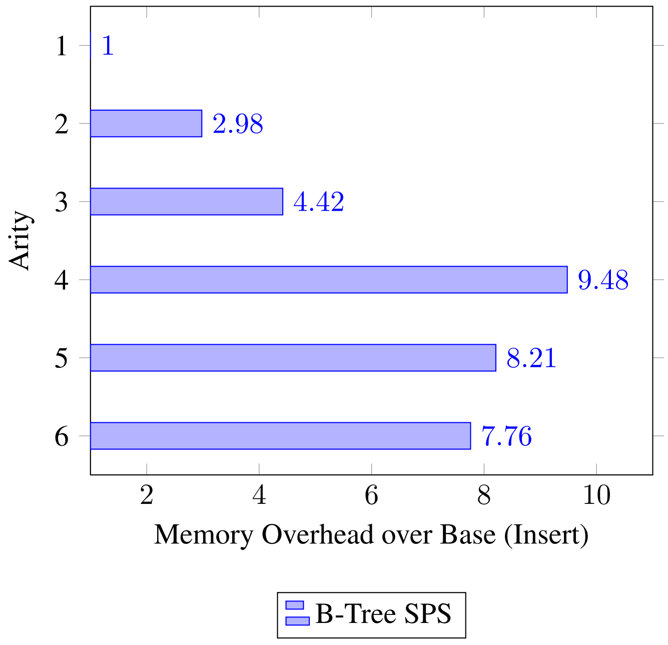
Index Selection Strategy	Number of B-Tree Indexes
Base	3
B-Tree SPS	3×Arity

TABLE 4.31. Number of Indexes (Insert)

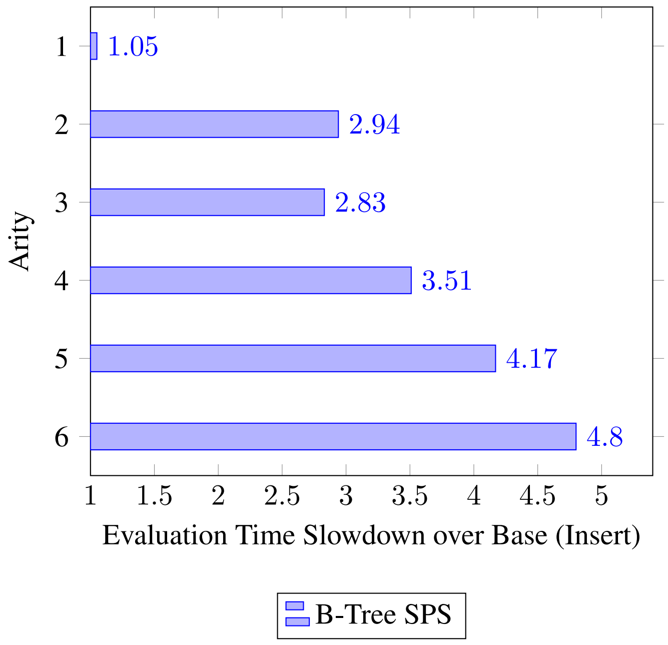
Due to semi-naïve evaluation we require three relations A, $delta_A$ and new_A. For each of these relations, the B-Tree SPS strategy must maintain n indexes where n is the arity of relation. A B-Tree must be constructed in order to efficiently support each inequality primitive search and since the searches are done on separate attributes, they cannot share an index. By contrast, the Base strategy does not cover any of these inequality primitive searches and only a single B-Tree index is sufficient to hold the tuples for A.

Arity	Memory Usage (Base)	Memory Usage (B-Tree SPS)	Evaluation Time (Base)	Evaluation Time (B-Tree SPS)
1	239836	239760	6.69	7.00
2	484584	1442616	10.06	29.60
3	721104	3189128	12.99	36.72
4	1108120	10504848	16.92	59.41
5	1464816	12019220	23.35	97.40
6	1698144	13183000	24.77	118.91

TABLE 4.32. Maximum Resident Set Size (KB) and Evaluation Time (s) (Insert)



There is a significant memory overhead as shown in the above figures when using B-Tree SPS. Since $n\times$ more B-Tree indexes are required to cover all inequality primitive searches compared to Base, the overall memory overhead is considerable. The memory overhead is significant enough that the B-Tree SPS strategy would potentially cause the evaluation to crash for large enough input sizes and Base is much more preferable.



The above figure showcases a worst-case scenario for the B-Tree SPS strategy. There is a clear linear relationship between the arity of the relation and the evaluation time slowdown over Base. Since the B-Tree SPS strategy must maintain $n\times$ the number of B-Tree indexes of Base and the workload is entirely comprised of insertions; we observe an $n\times$ slowdown. Overall, we can characterise the worst-case scenario of B-Tree SPS as when there are large numbers of non-overlapping inequality primitive searches, and the workload is insertion heavy. Although, this scenario is uncommon in practice, clearly for this benchmark, the Base strategy is far superior as it does not incur a large memory overhead or evaluation time slowdown.

We have found that the B-Tree SPS strategy demonstrates dramatic speedups while incurring minimal memory overhead for all synthesised benchmarks bar Insert, an unrealistic benchmark designed to highlight the theoretical weaknesses of the technique. However, most crucial to the adoption of B-Tree SPS as a replacement index selection strategy compared to Base is its performance for real-world benchmarks. Therefore, we now shift our attention to real-world applications, considering the evaluation time speedup of B-Tree SPS compared to the memory overhead and compilation time increase.

Index Selection Strategy	**Number of B-Tree Indexes**	**Compilation Time**
Base	594	140.8
B-Tree SPS	594	141.5

TABLE 4.33. Number of Indexes and Compilation Time (s) (DOOP)

Table 4.33, showcases the number of B-Tree indexes for DOOP, a large scale, real-world benchmark. The same number of B-Tree indexes can be used by B-Tree SPS to cover all of the extra inequality primitive searches introduced. Although only 0.6% of the search operations in DOOP are inequality primitive searches, the fact that B-Tree SPS creates no additional B-Tree indexes gives strong guarantees. We can be confident that there is zero overhead concerning the compilation time or the memory overhead. Most importantly, however, even if the evaluation time does not decrease, it will not be noticeably larger as no extra B-Tree indexes need to be maintained to speed up inequality primitive searches.

The above table illustrates that the increase in compilation time of B-Tree SPS compared to Base is zero within the range of experimental error (1%). From Table 4.33, the reasoning here is that B-Tree SPS does not synthesise any extra indexes and therefore no extra time is required to generate the C++ program or compile it. There is some overhead with regards to the index selection overhead since there are inequality primitive searches not present with the Base strategy. However, the index selection phase consumes less than 1% of the overall compilation time and so we can dismiss this overhead. Overall, the B-Tree SPS technique is superior to Base in that it has negligible overhead to the compilation time while covering more searches.

Fact Folder	Memory Usage (Base)	Memory Usage (B-Tree SPS)	Evaluation Time (Base)	Evaluation Time (B-Tree SPS)
antlr	2046448	2046560	40.81	41.20
bloat	1445884	1446020	27.33	26.96
chart	2149824	2149840	42.85	43.14
eclipse	1443628	1443788	29.31	29.37
fop	2120208	2120476	41.75	41.89
hsqldb	2189184	2189288	44.41	44.35
jython	1627420	1627544	31.53	31.77
luindex	1443508	1443572	27.83	27.75
lusearch	1441388	1441260	27.85	27.94
pmd	2089748	2089940	41.61	41.77
xalan	2140004	2139920	43.06	42.55

TABLE 4.34. Maximum Resident Set Size (KB) and Evaluation Time (s) (DOOP)

The above table showcases the experimental memory overhead of B-Tree SPS. Since SOUFFLÉ constructs no additional B-Tree indexes, no extra memory is needed since both B-Tree SPS and Base maintain the same number of B-Tree indexes.

In terms of evaluation time, we observe not much improvement and not much overhead, either. We see no effect within the margin of experimental error (1%), indicating that no crucial inequalities exist in DOOP.

Index Selection Strategy	Number of B-Tree Indexes
Base	304
B-Tree SPS	304

TABLE 4.35. Number of Indexes (VPC N-1075 sec1)

Index Selection Strategy	Number of B-Tree Indexes
Base	315
B-Tree SPS	315

TABLE 4.36. Number of Indexes (VPC N-1075 sec2)

Index Selection Strategy	**Number of B-Tree Indexes**
Base	298
B-Tree SPS	298

TABLE 4.37. Number of Indexes (VPC N-1075 sec3)

Tables 4.35, 4.36 and 4.37 illustrate that zero extra B-Tree indexes are constructed by the B-Tree SPS technique. Again, with no extra B-Trees created, there are strong guarantees on the compilation time, memory overhead and performance overhead of B-Tree SPS relative to Base.

Program	**Compilation Time (Base)**	**Compilation Time (B-Tree SPS)**	**Compilation Time (Overhead over Base)**
VPC N-1075 sec1	127.1	129.9	1.00
VPC N-1075 sec2	134.6	136.0	1.01
VPC N-1075 sec3	126.0	126.0	1.01
VPC N-2340 sec1	52.64	52.49	1.00
VPC N-2340 sec2	55.38	56.20	1.01
VPC N-2340 sec3	52.04	51.83	1.01
VPC N-3500 sec1	128.54	130.68	1.02
VPC N-3500 sec2	134.98	135.83	1.01
VPC N-3500 sec3	128.67	129.61	1.01
VPC N-3511 sec1	127.98	130.47	1.02
VPC N-3511 sec2	132.49	133.90	1.01
VPC N-3511 sec3	136.95	137.25	1.00
VPC N-9087 sec1	127.59	127.86	1.00
VPC N-9087 sec2	133.14	135.38	1.02
VPC N-9087 sec3	128.14	129.97	1.01

TABLE 4.38. Compilation Time (s) of Index Selection Schemes (VPC)

As with DOOP, since the number of B-Tree indexes is identical for both B-Tree SPS and Base, the compilation time is roughly within the margin of experimental error. Therefore, there is no disadvantage in terms of compilation time overhead to using the B-Tree SPS strategy.

Network and Analysis	Memory Usage (Base)	Memory Usage (B-Tree SPS)	Evaluation Time (Base)	Evaluation Time (B-Tree SPS)
VPC N-1075 (sec1)	1631804	1631940	150.97	151.27
VPC N-1075 (sec2)	1962056	1961916	137.55	139.07
VPC N-1075 (sec3)	4195416	4195248	168.91	172.47
VPC N-2340 (sec1)	727508	727340	202.76	207.76
VPC N-2340 (sec2)	727628	727436	198.84	197.08
VPC N-2340 (sec3)	727060	727160	207.15	198.81
VPC N-3500 (sec1)	7706920	7706864	1862.17	1888.08
VPC N-3500 (sec2)	14359900	14359764	1605.51	1586.53
VPC N-3500 (sec3)	7497628	7497548	1619.32	1642.51
VPC N-3511 (sec1)	7706952	7707228	1846.28	1878.21
VPC N-3511 (sec2)	14359824	14359612	1618.12	1591.3
VPC N-3511 (sec3)	7497612	7497624	1639.53	1632.26
VPC N-9087 (sec1)	6506056	6505960	1814.89	1810.58
VPC N-9087 (sec2)	11633248	11633324	1557.48	1533.89
VPC N-9087 (sec3)	7567900	7568032	1587.98	1577.34

TABLE 4.39. Maximum Resident Set Size (KB) and Evaluation Time (s) (VPC)

Table 4.39 details the memory used by the B-Tree SPS strategy and the Base strategy for VPC. We find that the memory usage is within the range of experimental error (1%) which aligns with our expectation as both techniques synthesise the same number of B-Tree indexes. Therefore, there is no harm in opting for the B-Tree SPS strategy over the Base strategy.

The above figures detail the evaluation time of the VPC network analysis using the B-Tree SPS technique with different analyses performed across different networks. We find that VPC is up to 4% faster with the B-Tree SPS technique relative to Base but also up to 2% slower than Base. Given that the same number of B-Tree indexes are used by each method, it is likely that our observations are due to experimental variance. Overall, both B-Tree SPS and Base have no significant difference in compilation time or memory overhead and there is no clear winner in terms of evaluation speed. Therefore, there is no harm in opting for the B-Tree SPS strategy compared to Base.

Index Selection Strategy	Number of B-Tree Indexes	Compilation Time
Base	692	178.7
B-Tree SPS	698	190.2

TABLE 4.40. Number of Indexes and Compilation Time (s) (DDISASM)

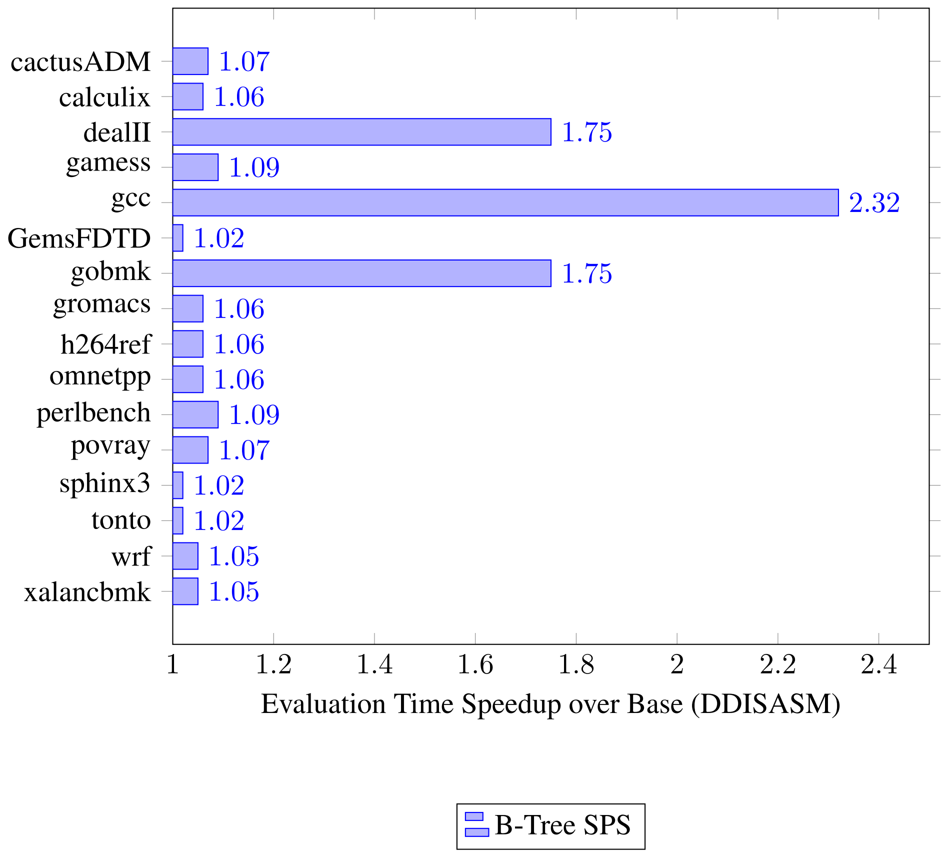
Table 4.40, showcases that DDISASM is the first benchmark where new B-Tree indexes are needed to cover all of the extra inequality primitive searches. Since only 3.1% of the search operations in DDISASM are inequality primitive searches, some relations are dense enough with new inequality primitive searches that extra search chains are required to cover them. However, B-Tree SPS builds only 6 additional indexes in order to cover these new searches, less than a 1% increase in the total number of indexes. Therefore we expect, B-Tree SPS will incur a compilation time overhead to construct the additional indexes and memory overhead to maintain them.

The B-Tree SPS technique represents the most considerable compilation time overhead observed so far, adding an overhead of 6% to Base and increasing the overall compilation time from 178.7s to 190.2s. Since the index selection phase of the compilation pipeline consumes less than 1% of the total time, we can conclude that this overhead is a consequence of constructing the 6 additional B-Tree indexes necessary to cover the extra inequality primitive searches in the program. Constructing each new B-Tree index requires evaluation of templates for each lexicographical order, therefore adding to the overhead as GCC compiles the C++ program. Overall, 6% is not a significant overhead of the B-Tree SPS technique and is acceptable to use instead of Base.

Fact Folder	**Memory Usage (Base)**	**Memory Usage (B-Tree SPS)**	**Evaluation Time (Base)**	**Evaluation Time (B-Tree SPS)**
cactusADM	445400	445140	10.21	9.53
calculix	1004012	1006824	26.61	25.15
dealII	609620	610232	47.71	27.22
gamess	5286504	5291832	160.2	147.31
gcc	2595832	2592636	177.88	76.68
GemsFDTD	278572	278200	9.38	9.16
gobmk	609556	610424	47.55	27.10
gromacs	569472	570452	13.34	12.54
h264ref	330768	331548	9.05	8.51
omnetpp	346472	347028	16.12	15.17
perlbench	1328780	1327216	25.79	23.76
povray	1400756	1399564	19.07	17.89
sphinx3	2795504	2803804	81.17	79.49
tonto	22435464	22442636	522.41	513.13
wrf	2796084	2803628	81.40	77.34
xalancbmk	2795532	2803772	80.70	77.11

TABLE 4.41. Maximum Resident Set Size (KB) and Evaluation Time (s) (DDISASM)

Table 4.41 highlights the lightweight nature of the B-Tree SPS strategy. Although 6 additional indexes had to be constructed, relative to those used by Base, we observe less than a 1% memory overhead compared to Base. We observed that DDISASM has large relations with inequality primitive searches when evaluating the R-Tree SPS technique, yet we observe nearly zero memory overhead with B-Tree SPS. Among all the relations with inequality primitive searches, some of them may be large, and some may be small. Since there is negligible memory overhead for B-Tree SPS then for the large relations with inequality primitive searches, it must use the same number of B-Tree indexes as Base. In other words, the 6 additional indexes constructed by B-Tree SPS only cover smaller relations with inequality primitive searches. Ultimately, the B-Tree SPS technique incurs negligible memory overhead, making it the superior strategy to Base as it can accelerate the evaluation time of inequality primitive searches dramatically.



The above figure illustrates the speedup of the B-Tree SPS indexing scheme. The results demonstrate an impressive speedup accelerating evaluation time by up to 2.32×. The observation validates our original hypothesis that inequality primitive searches were very unoptimised in SOUFFLÉ compared to equality primitive searches. Notably, the technique is robust and shows no degradation in performance across all input fact folders. The B-Tree SPS strategy is superior to Base, demonstrating dramatic speedups with a negligible overhead in compilation time and memory usage.

4.4 (Q4) R-Tree SPS vs B-Tree SPS

We have experimented using a range of constructed and real-world benchmarks to determine the performance characteristics of both the R-Tree SPS and B-Tree SPS indexing strategies.

In synthesised benchmarks, we observed effectively zero overhead in terms of memory using B-Tree SPS and up to $2\times$ more memory usage for the R-Tree SPS technique. We showcased that both of the proposed strategies could significantly speed up the evaluation time of synthesised programs, noting that B-Tree SPS provided approximately $2\times$ higher performance than R-Tree SPS for almost all synthesised benchmarks. We note that the "Insert" benchmark, a pathological program designed to disadvantage the B-Tree SPS strategy, illustrated a dramatic memory overhead and an $n\times$ evaluation time slowdown. However, since this benchmark would require a heavy insertion workload, on a relation with large numbers of inequality primitive searches using separate attributes, it is not a scenario that we expect ever to occur in practice. Overall, ignoring this pathological scenario, B-Tree SPS beats out R-Tree SPS for synthesised benchmarks, proving to be superior in terms of memory overhead and evaluation time speedup.

For real-world benchmarks, we observed that both techniques show negligible compilation time overhead, with R-Tree SPS increasing compilation time by at most 9% and B-Tree SPS with an overhead of 6%. Regarding memory usage, the R-Tree SPS technique consumes up to 137% more memory than Base, as shown in the DDISASM benchmark. By contrast, B-Tree SPS is far more lightweight, incurring zero memory overhead for both DOOP and VPC, since it uses the same number of B-Tree indexes as Base and increasing the memory overhead by less than 1% for DDISASM. However, when observing the evaluation time, the R-Tree SPS technique heavily degraded performance in real-world programs, evaluating up to $20.55\times$ slower than Base in DOOP and up to $143.63\times$ slower than Base in DDISASM. The B-Tree SPS strategy, however, dramatically improved evaluation time in the DDISASM benchmark, accelerating evaluation time by up to $2.32\times$ relative to Base, demonstrating its ability to speed up inequality primitive searches significantly. Furthermore, the B-Tree SPS technique proved to be far more robust than R-Tree SPS with no degradation in performance within the margin of experimental error.

Overall, the R-Tree SPS strategy is not robust and not safe to implement as a general index selection strategy as it can heavily degrade performance in real-world benchmarks. By contrast, the B-Tree SPS strategy is highly robust and showcases dramatic improvements in evaluation time for real-world benchmarks with negligible overhead compared to Base in both compilation time and memory usage. Therefore, we can conclude that the B-Tree SPS is a better index selection strategy than R-Tree SPS for real-world Datalog programs.

CHAPTER 5

Conclusion and Future Work

Crucial to the high performance of Datalog engines is the choice of indexes to store logical relations and accelerate the evaluation of search operations. The state of the art technique in the literature uses an automatic index selection technique to select a minimal cluster of B-Tree indexes to accelerate all equality primitive searches on a Datalog relation. However, for inequality primitive searches, their evaluation is performed naïvely without using an index, consuming excessive amounts of time for crucial logic programs.

In this thesis, we have extended the state of the art auto-index selection technique to accelerate both equality and inequality primitive searches. We introduced the notion of a spatial primitive as an abstract device to encapsulate both equality and inequality primitive searches. We showed an equivalence between the semantics of Datalog searches as spatial primitive searches and orthogonal range queries. We then explored high-performance data structures for evaluating orthogonal range queries and selected the R-Tree index as the most appropriate choice for a Datalog engine. We then presented two index selection strategies for evaluating spatial primitive searches in practice and implemented prototypes of both approaches in SOUFFLÉ.

The first approach, R-Tree SPS, builds an R-Tree index whenever inequality primitive searches are applied to Datalog relations and defers to the existing auto-index selection strategy for all other relations. The approach is feasible for synthesised benchmarks but is dramatically slower than the state of the art technique when evaluating real-world Datalog programs.

The second is B-Tree SPS, extending the existing auto-index selection technique to cover every simple spatial primitive search. The new strategy is the primary contribution of our research, showcasing the ability to dramatically accelerate the evaluation of inequality primitive searches without the need for any user intervention.

Our empirical experiments demonstrated the feasibility of B-Tree SPS on real-world Datalog programs, accelerating the evaluation-time of DDISASM by up to $2.32\times$, consuming less than 1% additional memory and incurring only a 6% increase in compilation time compared to the state-of-the-art. For other real-world Datalog programs such as DOOP and VPC, B-Tree SPS was robust, with no practical effect on compilation time, memory overhead or evaluation time. Overall B-Tree SPS is a robust and lightweight auto-index selection scheme, either matching or dramatically improving real-world benchmarks while R-Tree SPS fails to meet real-world demands.

5.1 Future Work

There are many areas of future work to explore, both regarding the practical applications of the new index selection technique and other strategies to improve the evaluation of logic programs.

5.1.1 Efficient Evaluation of Min/Max Aggregates

Aggregate functions perform a calculation on one or more values and return a single value. Aggregate functions are popular language extensions used in relational databases such as MySQL and PostgreSQL but are also common in Datalog engines including SOUFFLÉ and LogicBlox. Common aggregate functions include $count$, sum, $average$, min and max.

Consider for example the following aggregate expression in SOUFFLÉ:

$$B(y) \coloneq y = sum\ \ z : A(0, z).$$

We express in a declarative manner that we wish to compute the sum of the second attribute of all tuples in relation A under the constraint that each tuple's first attribute must have the value of 0. Internally, the aggregate is evaluated on relation A by first evaluating the spatial primitive search: $\sigma_{x_1=0}(A)$, and then summing the value of the second attribute from the result set.

Two of the most common aggregates, min and max can be accelerated using our extended auto-index selection technique. Given an aggregate expression of the form:

$$y = min/max\ \ v : A(z_1, ..., z_{k-1}, v, z_{k+1}, ..., z_n)$$

We permit $z_1, z_2, ..., z_n$ to be constants or variables/expressions that are supplied by other atoms in the rule. We also note that any variable/expression z_i which does not appear elsewhere the rule may be replaced by an _ to indicate that the attribute value is unused.

Currently to evaluate aggregates of this form, SOUFFLÉ will evaluate the spatial primitive search:

$$\sigma_{z_1 \le x_1 \le z_1,\ ...,\ z_{k-1} \le x_{k-1} \le z_{k-1},\ z_{k+1} \le x_{k+1} \le z_{k+1},\ ...,\ z_n \le x_n \le z_n}(A)$$

Or more simply:

$$\sigma_{x_1 = z_1,\ ...,\ x_{k-1} = z_{k-1},\ x_{k+1} = z_{k+1},\ ...,\ x_n = z_n}(A)$$

Firstly, the lower and upper bounds of the range need to be computed which has a worst-case time complexity of $\mathcal{O}(\log(|A|))$. After generating the resulting range, a loop iterates through the resulting tuples, keeping track of the minimum or maximum value attained by v which consumes $\mathcal{O}(|\sigma(A)| + |\log(A)|)$ time where $\sigma(A)$ is the set of tuples in relation A satisfying the spatial primitive search. Therefore, to evaluate aggregate expressions of the above form, the complexity is dominated by iterating through the range, consuming $\mathcal{O}(|\sigma(A)|)$ time.

A naïve scan of the resulting range for the minimum/maximum value of v is necessary since there is no defined ordering of tuples and thus any tuple in the range could attain the minimum/maximum value. However, if we could guarantee that the range is ordered by increasing values of v, then we would guarantee that the first tuple in the range would attain the minimum value and the last tuple would attain the maximum. With this guarantee, the range would not require iteration and the lower or upper bound of the range would satisfy the aggregate, reducing the complexity of evaluating aggregates of this form to $\mathcal{O}(|\log(A)|)$.

Therefore, to guarantee this ordering among tuples in the range we add the constraint $\inf(\mathbb{D}_k) \le x_k \le \sup(\mathbb{D}_k)$, an inequality constraint that every tuple satisfies trivially, to the spatial primitive search as follows:

$$\sigma_{x_1 = z_1,\ ...,\ x_{k-1} = z_{k-1},\ \inf(\mathbb{D}_k) \le x_k \le \sup(\mathbb{D}_k),\ x_{k+1} = z_{k+1},\ ...,\ x_n = z_n}(A)$$

Now after introducing an inequality constraint on x_k the corresponding search sets will be:

$$S_{EQ} = \{x_1, ..., x_{k-1}, x_{k+1}, ..., x_n\}$$

$$S_{INEQ} = \{x_k\}$$

Now, our extended index selection strategy guarantees that any index ℓ covering the search must conform with $S_{EQ} \prec S_{INEQ}$ or more simply $S_{EQ} \prec x_k$. Now for all tuples in the range, the values of their attributes $x_i \in S_{EQ}$ are all fixed i.e. $x_i = z_i$. Tuples in the range will all compare equal up until the last attribute of the lex-order x_k. Therefore, since the index using ℓ uses the value of x_k to determine the order of tuples, the range will be ordered by increasing value of x_k.

In effect, by expressing the min/max aggregates as spatial primitive searches, we guarantee that their evaluation is done with an index and is efficient. The advantage of this technique is that adding extra spatial primitive searches rarely requires the construction of additional indexes for the relation. Even if adding this new spatial primitive search results in an additional index for the relation, reducing the complexity of these min/max aggregates from $\mathcal{O}(|\sigma(A)| + |\log(A)|)$ to $\mathcal{O}(|\log(A)|)$ is dramatic enough that it should outweigh the extra maintenance costs of the new index. Overall, implementing this technique into SOUFFLÉ is a high priority as it can result in the automatic acceleration in the evaluation of min/max aggregates in a variety of Datalog applications.

5.1.2 Improving Performance of Datalog Provenance

Logic programs are expressed declaratively without specifying the control flow as one would with imperative programs. Therefore, unlike imperative programs where debugging can be performed by setting breakpoints, debugging Datalog programs is considerably more challenging. The state-of-the-art technique in debugging logic programs is data provenance, providing the origins of a tuple. The technique works by providing a proof tree, where the leaf level of the proof tree consists of only the derived tuple, and each level of the tree above it contains the EDB/IDB tuples in the rule bodies used to derive it.

The key idea is that the Datalog engine stores extra information about the derivation of each tuple during evaluation time. After evaluation time, if the program produces a tuple that should not exist, then the user can query the Datalog engine for the provenance of the tuple. The Datalog engine will then use the extra state computed during evaluation time to build a proof tree and present it to the user.

The challenge of practical Datalog provenance for large-scale applications is to have the ability to answer provenance queries quickly while minimising the overhead during evaluation time. The state-of-the-art Datalog provenance technique deployed in SOUFFLÉ (Zhao et al., 2020) offers minimal run-time overhead of $1.31\times$ and memory-overhead of $1.71\times$; however, proof construction time for large-scale benchmarks is quite costly. The crucial cost is to efficiently generate a minimal height proof-tree using only the small amount of extra state kept during evaluation time. The current evaluation of these queries for minimum height proof-trees is naïve, considering large numbers of potential tuples that will not appear in the final proof-tree.

For future work, we aim to integrate the new index selection technique for rules with inequality constraints with provenance. The key idea would be to perform the index selection twice: firstly to cover the queries performed at evaluation time and secondly to speed-up provenance queries. By leveraging indexes to perform these queries for minimal height proof-trees, we expect a tangible reduction in the proof-tree construction time.

5.1.3 Index Selection with Partial Indexes

Our index selection technique operates by constructing a cluster of B-Tree indexes to accelerate the evaluation of all of the spatial primitive searches performed on a Datalog relation. The technique is geared toward an in-memory Datalog engine, storing tuples of a relation in a clustered index. Therefore, when querying the index, the satisfying tuples can be retrieved directly without requiring a lookup into an unclustered in-memory or on-disk table containing the relation's tuples. It is often the case that only a single clustered index is required to cover all of the searches performed on a relation. However, if multiple indexes are required, replica indexes are constructed with different lexicographical orderings.

The key disdvantage of storing replica indexes is that not all of the relation's attributes require storage to satisfy the queries on the index. Consider the following Datalog program:

$$
\begin{array}{l}
.decl\ \ A(x : number, y : number)\\
.input\ \ A\\
\\
.decl\ \ B(x : number)\\
.input\ \ B\\
\\
.decl\ \ B(x : number)\\
.output\ \ C\\
\\
C(z) :- B(x), A(x, \_).\\
C(y) :- B(y), A(\_, y).
\end{array}
$$

Relation A has two spatial primitive searches corresponding to atoms $A(x, \_)$ and $A(\_, y)$. Every relation in a Datalog program also has an added spatial primitive search on all of its attributes. In this case, the search would correspond to the $A(x, y)$. The search is an existence check to determine whether a tuple already exists in a relation before insertion. If the tuple already exists then the tuple is not inserted in order to maintain the set property of Datalog relations. Therefore, the search sets $\{x_1\}, \{x_2\}$ and $\{x_1, x_2\}$ represent spatial primitive searches on the first, second and on both attributes respectively. Two indexes are required to cover all of these searches on relation A with a possible set of indexes being $\ell_1 = x_1 \prec x_2$ and $\ell_2 = x_2$.

For the index ℓ_1 of A, both attributes must be stored within the index as both attributes appear in the lex-order. We call this index which covers the existence check on a relation the *master index*. Index ℓ_2, by contrast, does not require storage of x_2 as the corresponding Datalog rule never references this attribute. Although only one of the attributes requires storage, since the index selection strategy creates replica indexes with different orders, all attributes of the relation are stored by ℓ_2 regardless. Any attribute referenced in the body of an atom must be stored in the index even if it is unused in the spatial primitive search. For example: in the atom $B(y)$, the attribute y has no constraint yet since it is referenced in $C(y)$, it still must be stored. We denote the complete set of attributes, utilised by the spatial primitive search and referenced by other atoms in the rule, the *auxiliary set* of the atom, i.e. $aux(B(y)) = \{y\}$.

Therefore, a *partial index* could be deployed for ℓ_2 instead, storing only the attribute x_2 for every tuple. We note that the utilisation of partial indexes would only come into effect when a relation requires multiple indexes since the master index for every relation is always full. The most obvious advantage of storing replace indexes as partial indexes is the reduction in memory usage as for every tuple in ℓ_2 it would require half the storage as a tuple in a full index.

More interestingly, there is a much greater advantage to using a partial index to store relations. Consider the tuples $t_1 = \langle 1, 1 \rangle$ and $t_2 = \langle 2, 1 \rangle$. When employing a regular index, these tuples are distinct, and both would be stored. However, the partial index for ℓ_2 would only store the x_2 value, which is 1 for both tuples. Therefore, to maintain the set property over the partial index, both of these tuples can be represented by ℓ_2 by storing a single sliced tuple, $t_{sliced} = \langle 1 \rangle$. By slicing off unused attributes from each tuple, the partial index performs a *dimensionality reduction* on the full index, no longer requiring the storage of any tuples which only vary on unused dimensions.

The primary benefit we expect if a partial index were to store sliced tuples would be this dimensionality reduction, reducing the atom frequency of searches covered by partial indexes. To integrate this partial indexing technique into SOUFFLÉ, we can apply the following algorithm after the index selection is made. For every atom whose corresponding spatial primitive search is covered by an index ℓ, we compute its auxiliary set $aux(A_i)$, representing all attributes referenced in the atom or used in the search. The union of the auxiliary sets of all the atoms corresponding to searches in the search chain then represents the set of all attributes that require storage by the partial index. Therefore, we can construct all of the necessary partial indexes from these sets, cutting down on both the memory overhead and performing a dimensionality reduction on the set of tuples. The primary challenge in the implementation is a bijective mapping between the position of attributes in sliced tuples and regular tuples at the interface boundary of the index. Providing this mapping would then enable a partial index to be used in the same way as a full index, requiring no changes to consumers of the interface. Overall, despite the engineering challenges involved, the partial indexing technique can only reduce memory consumption and improve evaluation performance, so we aim to employ partial indexes in SOUFFLÉ soon.

5.1.4 Dynamic Query Scheduling

The efficient evaluation of logic programs is primarily dependent on two factors: query scheduling and index selection. In this thesis, we assumed a fixed query schedule for each rule and then computed a minimal index selection to cover each of the searches within the rules. In practice, the choice of query schedule can affect performance even more dramatically than the index selection. However, including the assumption of a fixed query schedule is reasonable given that experimenting to find a satisfactory loop schedule is much less time-intensive than finding the right index selection.

Unfortunately, there is still considerable time invested in order to find query schedules that deliver satisfactory performance. The last step towards Datalog being a performance declarative language would be for Datalog engines to automatically select the best query schedules without the need for user intervention. The primary challenge is that information known only at run-time is needed to determine an optimal query schedule. For instance, placing smaller sized relations first in the schedule can lead to fewer iterations of the loop nest, dramatically reducing the evaluation time. Datalog engines such as SOUFFLÉ operate by first translating the logic program into an imperative one without access to the program input. As a consequence, selecting an optimal query schedule statically is not possible since these crucial statistics are not available.

One possible solution would be to execute the program once and gather run-time statistics during evaluation and provide these to the Datalog engine for use when selecting the loop schedule. SOUFFLÉ programs are optimised similarly, first running the profiler on a representative input and then adjusting the loop schedule in the Datalog program manually. However, using fixed loop schedules introduces a few problems. Firstly, modern Datalog applications such as static program analysis tools are translated to imperative programs once and then run on a variety of different inputs. Selecting a representative input to use when selecting the loop schedule is not always possible given that different schedules may perform better when evaluating the program with different inputs.

Furthermore, even for the same input, the optimal loop schedule changes throughout the evaluation. For example, on some iterations of a rule, a relation's size may be small, but on another, the size is much larger. Consequently, evaluation time can improve by using different query schedules across different iterations of the same rule.

For future work, we aim to research into a dynamic query scheduler which can change the order of the loop nest on the fly. It would keep track of vital run-time statistics such as the sizes of relations and re-order the loop-nest as required to achieve satisfactory performance. Not only would this free users from the manual effort of experimenting with different loop schedules, but the technique could offer dramatically higher performance evaluation, specialising the schedule for the current input.

Bibliography

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*, volume 8. Addison-Wesley Reading.

Pankaj K Agarwal, Jeff Erickson, et al. 1999. Geometric range searching and its relatives. *Contemporary Mathematics*, 223:1–56.

Sheldon B. Akers. 1978. Binary decision diagrams. *IEEE Transactions on computers*, (6):509–516.

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting doop to soufflé: a tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 25–30.

Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382.

John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, et al. 2019. Reachability analysis for aws-based networks. In *International Conference on Computer Aided Verification*, pages 231–241. Springer.

Francois Bancilhon. 1986. Naive evaluation of recursively defined relations. In *On Knowledge Base Management Systems*, pages 165–178. Springer.

Francois Bancilhon and Raghu Ramakrishnan. 1989. An amateur's introduction to recursive query processing strategies. In *Readings in Artificial Intelligence and Databases*, pages 376–430. Elsevier.

Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The r*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331.

JL Bentley. 1979. Decomposable searching problems information processing letters.

Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.

Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. 2003. Points-to analysis using bdds. *ACM SIGPLAN Notices*, 38(5):103–114.

Spyros Blanas, Yinan Li, and Jignesh M Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 37–48.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262.

Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *PLDI*, pages 454–469.

Dario Campagna, Beata Sarna-Starosta, and Tom Schrijvers. 2012. Optimizing inequality joins in datalog with approximated constraint propagation. In *International Symposium on Practical Aspects of Declarative Languages*, pages 108–122. Springer.

Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43.

Bernard Chazelle and Leonidas J Guibas. 1986. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162.

Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. 1997. Computational geometry. In *Computational geometry*, pages 1–17. Springer.

Robert P Dilworth. 2009. A decomposition theorem for partially ordered sets. In *Classic Papers in Combinatorics*, pages 139–144. Springer.

Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

Edward Fredkin. 1960. Trie memory. *Communications of the ACM*, 3(9):490–499.

Delbert Ray Fulkerson. 1956. Note on dilworths decomposition theorem for partially ordered sets. In *Proc. Amer. Math. Soc*, volume 7, pages 701–702.

Yoshihiko Futamura. 1999. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391.

Yvh J Garcia, R Mario, A Lpez, and Scott T Leutenegger. 1998. On optimal node splitting for r-trees. In *In Proc. of VLDB 98*. Citeseer.

Sergio Greco and Cristian Molinaro. 2015. Datalog and logic databases. *Synthesis Lectures on Data Management*, 7(2):1–169.

Todd J Green, Molham Aref, and Grigoris Karvounarakis. 2012. Logicblox, platform and language: A tutorial. In *International Datalog 2.0 Workshop*, pages 1–8. Springer.

Simon Gerard Greener and Siva Ravada. 2013. *Applying and Extending Oracle Spatial*. Packt Publishing Ltd.

Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57.

Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. 2011. μz–an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, pages 457–462. Springer.

Sangyong Hwang, Keunjoo Kwon, Sang K Cha, and Byung S Lee. 2003. Performance evaluation of main-memory r-tree variants. In *International Symposium on Spatial and Temporal Databases*, pages

10–27. Springer.

Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer.

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019a. Brie: A specialized trie for concurrent datalog. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 31–40.

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019b. A specialized b-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 327–339.

Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2020. Specializing parallel data structures for datalog. *Concurrency and Computation: Practice and Experience*, page e5643.

Ibrahim Kamel and Christos Faloutsos. 1992. Parallel r-trees. *ACM SIGMOD Record*, 21(2):195–204.

Maurizio Lenzerini. 2002. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246.

Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108.

George S Lueker. 1978. A data structure for orthogonal range queries. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 28–34. IEEE.

Christoph Meinel and Thorsten Theobald. 2012. *Algorithms and Data Structures in VLSI Design: OBDD-foundations and applications*. Springer Science & Business Media.

Patrick Nappa, David Zhao, Pavle Subotić, and Bernhard Scholz. 2019. Fast parallel equivalence relations in a datalog compiler. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 82–96. IEEE.

Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases*, pages 46–65. Springer.

John T Robinson. 1981. The kdb-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18.

Kenneth A Ross. 1990. Modular stratification and magic sets for datalog programs with negation. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 161–171.

Konstantinos Sagonas, Terrance Swift, and David S Warren. 1994. Xsb as an efficient deductive database engine. *ACM SIGMOD Record*, 23(2):442–453.

Isaac Arch Samuel. 2020a. Benchmarking. `https://github.com/SamArch27/souffle/tree/rtree`.

Isaac Arch Samuel. 2020b. Indexed inequalities. `https://github.com/souffle-lang/souffle/pulls?q=is%3Apr+author%3ASamArch27+is%3Aclosed+is%3Amerged%3A%3C2020%3A11%3A22+`.

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 196–206.

Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. 2012. *High performance MySQL: optimization, backups, and replication.* " O'Reilly Media, Inc.".

Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic index selection for large-scale datalog computation. *Proceedings of the VLDB Endowment*, 12(2):141–153.

Hisao Tamaki and Taisuke Sato. 1986. Old resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98. Springer.

K Tuncay Tekle and Yanhong A Liu. 2011. More efficient datalog queries: subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 661–672.

Jeffrey D Ullman. 1989. Bottom-up beats top-down for datalog. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 140–149.

Todd L Veldhuizen. 2012. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems*, pages 97–118. Springer.

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*, pages 131–144.

Lijing Zhang and Jing Yi. 2010. Management methods of spatial data based on postgis. In *2010 Second Pacific-Asia Conference on Circuits, Communications and System*, volume 1, pages 410–413. IEEE.

David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging large-scale datalog: A scalable provenance evaluation strategy. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(2):1–35.

APPENDIX A

Spatial Primitive Searches and Orthogonal Range Queries

We now prove the equivalence between spatial primitive searches and their corresponding orthogonal range queries from Section 3.5.

THEOREM 1. *Given a spatial primitive search $\sigma_{l_1 \leq x_1 \leq u_1, \ldots, l_k \leq x_k \leq u_k}$ over a relation R yielding the set of satisfying tuples T, and its corresponding orthogonal range query returning points Q we have $T = f^{-1}(Q)$.*

PROOF. We prove this theorem by proving that $f^{-1}(Q) \subseteq T$ and $T \subseteq f^{-1}(Q)$.

$\Rightarrow$

For every tuple $t \in T$ we have that $l_1 \leq x_1 \leq u_1, \ldots, l_k \leq x_k \leq u_k$. When an attribute has no lower bound or upper bound constraint then an artificial lower or upper bound is introduced in the interval i.e. $lower_i = \inf(\mathbb{D}_i)$ and/or $upper_i = \sup(\mathbb{D}_i)$. All points in S trivially satisfy these bounds on the intervals for the corresponding intervals. If a lower bound appears in the spatial primitive search then $lower_i = l_i$. We know that $t(x_i) \geq l_i$ from the spatial primitive search and since f maps the attribute values to the same values in the corresponding point, $f(t)(x_i) \geq l_i$ and lies above the lower end of the interval. Similarly, when an upper bound appears in the spatial primitive search then $upper_i = x_i$ and $f(t)(x_i) \leq u_i$ and lies within the upper end of the interval. Therefore, for all tuples satisfying the spatial primitive search T, after being mapped by f to a point in d-dimensional space lie within the corresponding intervals b_i of the box B. Therefore, all mapped points $f(T)$ lie within the query box i.e. $f(T) \subseteq Q$. Since f is bijective we can apply the inverse f^{-1} to both sides and find that $T \subseteq f^{-1}(Q)$.

$\Leftarrow$

Consider a point $q \in Q$ satisfying the orthogonal range query. Since every point $q \in Q$ lies within the box B, for every dimension i, $q(x_i) \in b_i = [lower_i, upper_i]$. When $lower_i = \inf(\mathbb{D}_i)$ or $upper_i = \sup(\mathbb{D}_i)$ then by construction there is no corresponding lower/upper bound constraint in the spatial primitive search on x_i. Now considering the case where $lower_i = l_i$ then $t(x_i) \geq l_i$ where $t = f^{-1}(q)$ since the lower bound of the interval b_i is l_i. Likewise, if $upper_i = u_i$ then $t(x_i) \leq u_i$. Hence, for every point $q \in Q$ that satisfies the orthogonal range query, the corresponding tuple $f^{-1}(q)$ must satisfy the spatial primitive search i.e. $f^{-1}(Q) \subseteq T$.

Since $T \subseteq f^{-1}(Q)$ and $f^{-1}(Q) \subseteq T$ then $T = f^{-1}(Q)$. $\square$

APPENDIX B

Attributes with Inequalities Must be at the End of the k^{th} Prefix of ℓ

We prove Lemma 2 from Section 3.5.5

PROOF.

$\Rightarrow$

We prove the claim by showing that the semantics of the simple spatial primitive search always coincide with the corresponding lex-search. Firstly, simple spatial primitive searches generalise equality primitive searches so the k^{th}-prefix constraint must be satisfied for simple spatial primitive searches also. Now x_i must appear as the last attribute in the k^{th}-prefix. Consider a lex-order $\ell = ... \prec x_i \prec ...$ where the attributes occurring in the lex-search may be any from the set A_R. Now the k^{th}-prefix must be of the form: $\ell_k = ... \prec x_i \prec ...$ using attributes only from the set S. Since x_i is the last attribute in the k^{th}-prefix we can write $\ell_k = ... \prec x_i$. Note that since x_i is the only attribute with an inequality constraint that for all other attributes in the search predicate $x_i \in S_{EQ}$ i.e. $l_j \leq x_j \leq u_j$ where $l_j = u_j$. Therefore, as a short-hand we write $v_j = l_j = u_j$ for all $j \neq i$.

When constructing the lower and upper bounds for the lex-search we have:

$$a = \langle v_1, v_2, ..., v_{k-1}, l'_i \rangle$$

$$b = \langle v_1, v_2, ..., v_{k-1}, u'_i \rangle$$

Next, we consider all of the tuples that satisfy the lex-search $\sigma_{\rho(\ell,a,b)}(R)$ and ensure that the semantics coincide with the simple spatial primitive search by considering each attribute constraint. We prove by induction that precisely the set of tuples satisfying all constraints excluding those on x_i in the search satisfy the lex-search.

For a base case we consider the first attribute constraint $x_1 = v_1$ in the simple spatial primitive search, clearly this is satisfied precisely when a tuple $\{t \in R \mid a \sqsubseteq_\ell t \sqsubseteq_\ell b\}$ as $a(x_1) = v_1 \leq t(x_1) = v_1$

and $b(x_1) = v_1 \geq t(x_1) = v_1$. By the inductive hypothesis we assume that all constraints in the search predicate are satisfied up to equality constraint on attribute x_{n-1}. For attribute x_j we have that $t(x_j) = v_j$ for all $j < n$. For $j = n$ we also have that $a(x_n) = v_n$ and $b(x_n) = v_n$ and therefore, the only tuples satisfying the lex-search must satisfy $t(x_j) = v_j$ for all $j \leq n$. Therefore, by induction the semantics of the simple spatial primitive search match those of the search predicates up until x_i.

Firstly, if l_i is not specified in the search predicate then l'_i is $\inf(\mathbb{D}_i)$ and $x_i \geq \inf(\mathbb{D}_i)$ for any value of x_i. Similarly, if u_i is unspecified then $u'_i = \sup(\mathbb{D}_i)$ which is also satisfied by any x_i. Now if l_i is specified then only tuples satisfying $t(x_i) \geq a(x_i) = l_i$ will satisfy the lex-search. Likewise if u_i is specified then only tuples satisfying $t(x_i) \leq b(x_i) = u_i$ will satisfy the lex-search. Therefore, by a case-by-case analysis, any tuples satisfying the lex-search are precisely those satisfying the inequality predicate on x_i. Overall, since we know all of the predicates for a simple spatial primitive search are satisfied precisely when the tuples satisfy the corresponding lex search we have proven that:

$$\sigma_{l_1 \leq x_1 \leq u_1, \ldots, l_k \leq x_k \leq u_k}(R) = \sigma_{\rho(\ell,a,b)}(R)$$

and ℓ when x_i is at the end of the k^{th}-prefix of ℓ.

$\Leftarrow$

By Lemma 1, we know that if x_i is not at the end of the k^{th}-prefix of ℓ then the pair of search sets cannot be covered by ℓ. Therefore by transposition, if a simple spatial primitive search is coverable by an index ℓ, then it must have x_i at the end of the k^{th}-prefix of ℓ. $\square$

APPENDIX C

Covering Search Set Pairs with B-Trees

We prove Lemma 3 from Section 3.5.6

PROOF.

$\Rightarrow$

By Lemma 2, an index ℓ covers (S'_{EQ}, S'_{INEQ}) if S' is the k^{th}-prefix of ℓ and if $x_i \in S'_{INEQ}$ then x_i is at the end of the k^{th}-prefix. We know that $S \subseteq S'$ and therefore S is also a k^{th}-prefix. We know that if $x_i \in S'_{INEQ}$ then $x_i \notin S$.

Therefore, with respect to the k^{th}-prefix of S', x_i can always be placed at the end since x_i does not appear in the k^{th}-prefix of S. Since $S \subseteq S'$, then S is also a k^{th}-prefix of ℓ and therefore covers both simple spatial primitive searches. Thus, there exists an index ℓ such that (S_{EQ}, S_{INEQ}) and (S'_{EQ}, S'_{INEQ}) can both be covered by ℓ.

$\Leftarrow$

Assume now that (S_{EQ}, S_{INEQ}) and (S'_{EQ}, S'_{INEQ}) can both be covered by an index ℓ and it is known that $S \subseteq S'$ holds. Next, by Lemma 2, for (S_{EQ}, S_{INEQ}) and (S'_{EQ}, S'_{INEQ}) to both be covered by the same index ℓ, if $x_i \in S'_{INEQ}$ then x_i must appear at the end of the corresponding k^{th}-prefix for S'. Additionally, for any two search pairs to be covered by the same index ℓ they must both be k^{th}-prefixes of ℓ and thus $S \subseteq S'$. $\square$