PrefixSolve: Efficiently Solving Multi-Source Multi-Destination Path Queries on RDF Graphs by Sharing Suffix Computations

Sidan Gao
Department of Computer Science
North Carolina State University
Raleigh, NC 27606, USA
sgao@ncsu.edu

Kemafor Anyanwu
Department of Computer Science
North Carolina State University
Raleigh, NC 27606, USA
kogan@ncsu.edu

ABSTRACT

Uncovering the "nature" of the connections between a set of entities e.g. passengers on a flight and organizations on a watchlist can be viewed as a *Multi-Source Multi-Destination (MSMD) Path Query* problem on labeled graph data models such as RDF. Using existing graph-navigational path finding techniques to solve MSMD problems will require queries to be decomposed into multiple single-source or destination path subqueries, each of which is solved independently. Navigational techniques on disk-resident graphs typically generate very poor I/O access patterns for large, disk-resident graphs and for MSMD path queries, such poor access patterns may be repeated if common graph exploration steps exist across subqueries.

In this paper, we propose an optimization technique for general MSMD path queries that generalizes an efficient algebraic approach for solving a variety of single-source path problems. The generalization enables *holistic* evaluation of MSMD path queries without the need for query decomposition. We present a conceptual framework for sharing computation in the algebraic framework that is based on "suffix equivalence". Suffix equivalence amongst subqueries captures the fact that multiple subqueries with different prefixes can share a suffix and as such share the computation of shared suffixes, which allows prefix path computations to share common suffix path computations. This approach offers orders of magnitude better performance than current existing techniques as demonstrated by a comprehensive experimental evaluation over real and synthetic datasets.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing

Keywords

RDF, MSMD Path Query, Suffix Equivalence, Work Sharing

1. INTRODUCTION

The growing availability of Semantic Web data is strengthening its position as an indispensable knowledge base. Many investigative and exploratory applications that rely on the Semantic Web have requirements beyond matching query-described subgraph patterns in data. Sometimes the goal is to "connect-the-dots" by uncovering the structure of relationships between a set of entities in complex social, biological, and other types of networks represented on the Semantic Web. The need to support such tasks has led to an increased interest in navigational query models e.g. reachability and path queries [9, 6, 22, 23, 20, 21, 13] and motivated recent extensions to the SPARQL query language to include a path expression query construct. Similarly motivated proposals were made earlier [6, 10, 4, 19, 5, 16]. The theoretical implications of evaluating SPARQL path expression queries are discussed in [8, 7, 14]. In particular, [7] demonstrates the limitation of the current semantics for such constructs that results in inefficient and impractical implementations in current systems, and warns about the negative impact on its adoption for real-world applications. Their results speak eloquently to the strong need for developing efficient query evaluation techniques for navigational queries.

Structure discovery queries require the ability to *extract* undescribed substructures e.g. paths or subgraphs based on a given set of entities. As examples, (i) the process of drug discovery may need to explore *all* the ways a chemical interacts with potential drug target in order to select a mechanism of action (pathway) with the least amount of side effects and also to avoid not detecting potentially dangerous drug mechanisms, (ii) in security intelligence applications, investigators may be interested in understanding the entire scope of a criminal enterprise which includes all types of direct and indirect relationships. For many structure discovery applications, it is not always the case that interesting structures can be characterized simply in terms of their structures e.g. *shortest* paths. Rather, importance is often determined by the *nature of the structure* i.e. the types of properties/edges on paths. This perspective is particularly crucial for heterogeneous networks like Semantic Web networks but maybe less so for homogeneous networks where all relationships/edges have a uniform semantics. In addition, some applications such as drug target require a view into the *entire* scope of possible relationships, not just the most immediate ones. Unfortunately, most existing techniques primarily support shortest [18, 15] or bounded length paths/subgraphs [21] or pattern-based subgraphs [12] rather than generalized path querying.

For most applications interested in finding paths or connections, the focus is often on *sets* of sources and destina-

WWW 2013, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2035-1/13/05.

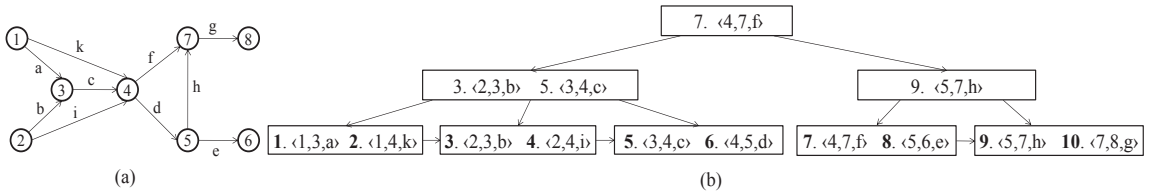


Figure 1: (a) A graph G (b) G's path sequence indexed using B+ tree

tions rather than a single source or destination i.e., ***Multi-Source Multi-Destination (MSMD) Path Queries***. Most existing techniques [18, 15, 21] require either the source or destination to be a single node. Therefore, MSMD path queries will need to be decomposed into multiple single-source path subqueries, one for each source or destination node. Unfortunately, this approach is very inefficient particularly if the database is disk-resident. For example, we ran a ***single source*** node path query on a real-world RDF dataset, BioCyc [1], containing 100K nodes using the best performing system based on the evaluation discussed in [7], and it took ***820 seconds!*** In contrast, a more *holistic*, yet naive, MSMD path query evaluation technique with ***120 query source nodes*** on the same (BioCyc) dataset took 700 secs! This is despite the fact that for the latter case the database was on disk (indexed) while the former had the graph in memory. Enabling more efficient generalized path and subgraph querying is the motivation of this paper. Related earlier efforts include [21, 13, 4, 18, 12].

In this paper, we propose an optimization technique for efficiently evaluating MSMD path queries. The optimization generalizes an algebraic approach [24, 25] for efficiently solving single-source path problems to efficiently solve ***multi-source*** path problems. The algebraic approach serves as a good foundation for efficient path query evaluation because (i) you can interpret different kinds of path problems in the same framework as discussed in [24, 25] allowing support for different kinds of path queries to be possible, (ii) it has two distinct phases that can be split into a preprocessing phase and a query processing phase with the cost of the former dominating, but can be amortized over all future queries, (iii) it is more amenable to holistic query evaluation and disk based graphs so that we can avoid the inefficiency of decomposing into single-source subqueries and very poor I/O access patterns of graph navigational strategies. Specifically, we make the following contributions:

- An efficient holistic evaluation strategy for MSMD path queries for disk resident RDF databases, which avoids query decomposition into single-source path queries.
- A conceptual framework for work sharing across subqueries based on a notion, "suffix equivalence", and algorithms for integrating work sharing into algebraic path problem solving techniques.
- A comprehensive evaluation of the approach over synthetic and real datasets. The results show a significant improvement in performance of our approach.

2. PRELIMINARIES

An RDF graph is a directed labeled graph $G = (V_G, E_G, \lambda)$, where V_G is a finite set of nodes, $E_G \subseteq V_G \times V_G$ is a set of directed edges, $\lambda : E_G \rightarrow \Gamma$ is a labeling function which assigns each edge $e \in E_G$ to a label $\lambda(e) \in \Gamma$ from a set Γ of labels. We denote a labeled edge $e = (v_1, v_2)$ with label $\lambda(e) = l_e$ as (v_1, l_e, v_2). A path in an RDF graph is defined as an alternating sequence of nodes and labeled edges. A set of paths connecting two nodes can be represented concisely as a *path expression* (PE). A path expression of type (s, d), $PE(s, d)$, is a triple $\langle s, d, R\rangle$, where R is a regular expression over the set of labeled edges (Γ,E_G) defined using the standard operators union($\cup$), concatenation($\bullet$) and closure($*$) such that the language $L(R)$ of R represents paths from s to d where $s, d \in V_G$. ε and $\emptyset$ are two atomic regular expressions denoting empty string and empty set resp. For example, given the graph in Figure 1(a), the path expression of type $(1, 4)$, $PE(1, 4) = \langle 1, 4, (1, a, 3) \bullet (3, c, 4) \cup (1, k, 4)\rangle$. For brevity, we omit nodes in a regular expression (unless required), and simply describe path expressions in terms of regular expressions over edge labels. For example, our earlier path expression can be rewritten as $PE(1, 4) = \langle 1, 4, a \bullet c \cup k\rangle$.

If a graph is ordered using any numbering scheme, information about paths in the graph can be represented using a particular sequence of path expressions called a *path sequence* (PS) [24]. For simplicity, assume that we use a node and its assigned number interchangeably, then a path sequence is the sequence of path expressions $PS = \langle s_1, d_1, R_1\rangle$, $\ldots, \langle s_l, d_l, R_l\rangle$:

- beginning with $s_i \leq d_i$ in ascending order on s_i followed by
- $\langle s_i, d_i, R_i\rangle$ with $s_i > d_i$ in descending order on s_j.

An important property of PS is that for any non-empty path p in G, there is a unique sequence of indices of PS $1 \leq i_1 < i_2 < \cdots < i_k \leq l$ and $p = p_1, p_2, \ldots, p_k$ such that $p_j \in L(R_{i_j})$. Formally, for any path p, we can find a unique partition of p into non-empty subpaths, and a unique sequences of indices I of PS, such that the ith subpath of p is represented by the path expression at the ith index in I. Given a path sequence, many path problems are solved using a simple propagation SOLVE algorithm [24] that assembles path information as it scans the path sequence from left to right. The time complexity of SOLVE is a function of the length of a PS which is at most $O(|V_G|^2)$. However, by selecting good numbering schemes as mentioned in [24] or using heuristic techniques presented in [6], we can keep the length of path sequence much closer to $O(|E_G|)$ for G. [24] also proposes optimizations based on graph decomposition techniques that enable complexity of the single-source path expression problem to be reduced to $O(|E_G|\alpha(|E_G|, |V_G|))$ where α is the functional inverse of Ackermann's function.

To support disk-resident graphs, path sequence can be indexed using a B+ tree, $psIndex$, and SOLVE algorithm can be implemented as a generalization of an indexed scan of a path sequence [6]. Indexed scan uses a *getNext*() method to advance through the path sequence and returns the next path expression in the current retrieved leaf of the B+ tree. Once the last path expression in the leaf is processed, it advances to the next leaf of the B+ tree.

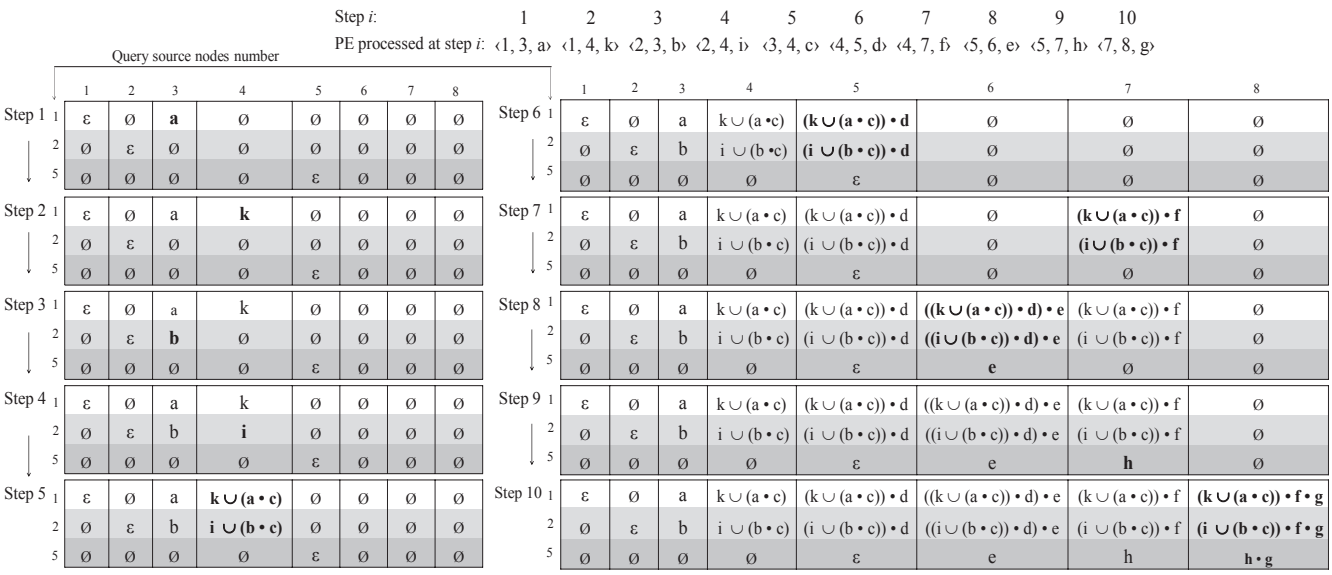


Figure 2: Stages of M-Solve for Sources $1, 2,$ and 5: (i) rows 1,2, and 3 for sources 1,2, and 5 resp and order is top to bottom then left to right; (ii) all row 1s simulate stages of S-Solve for source node 1

Figure 1 shows a graph G and its indexed path sequence as leaves of B+ tree. The bold numbers indicate the position of the path expression in PS, e.g., $PE(2,4)$ is the $4th$ path expression in PS.

Generalizing SOLVE algorithm to incorporate the index scan of a path sequence leads to an algorithm we refer to here as *S-Solve*. Given a path sequence and a source node s, *S-Solve* computes a path expression of type (s,d) for all $d \in V_G$. *S-Solve* uses an array SA, here called solve array, of size $|V_G|$ and begins by initializing $SA[s]$ with $\langle s, s, \varepsilon\rangle$.

S-Solve $(psIndex, s, SA)$
Initialization: $SA[s] \leftarrow \varepsilon$ and $SA[v] \leftarrow \emptyset$ for $v \neq s$;
1. $i = 1$
2. **while** $(PE_i(v_i, w_i) \leftarrow psIndex.getNext()) \neq NULL$
3. $\quad SA[w_i] \leftarrow SA[w_i] \cup (SA[v_i] \bullet PE_i(v_i, w_i))$
4. $\quad i = i + 1$
5. **end while**

Step i (i.e., iteration i) retrieves the ith path expression $PE_i(v_i, w_i)$ in PS with source v_i and destination w_i (line 2). Line 3 extends the path expression currently in $SA[v_i]$ i.e., $PE(s, v_i)$, by concatenating it with $PE_i(v_i, w_i)$. The resulting path expression $PE(s, w_i)$ represents paths from s to w_i via v_i. It then is used to extend the path expression currently in $SA[w_i]$ using a union operation. We refer to each step i.e., each iteration as a *s-SolveStep* - single SolveStep. If during a s-SolveStep line 3 "extends" $PE(s, v_i)$ with $PE_i(v_i, w_i)$ (to create $PE(s, w_i)$), then we refer to the action in line 3 as a (v_i, w_i) *extension*. Extensions will not be produced if $SA\,[v_i]$ is empty. This implies that so far no paths have been found from source s to v_i, thus there is no path from s to w_i via v_i. Consequently, we view the action of line 3 in this situation as a $\emptyset$ extension. At the end of *S-Solve*, $SA[d]$ contains $PE(s, d)$ representing all paths from s to d. The running time of *S-Solve* is $O(l)$, where l is the length of the path sequence.

Example 1. To illustrate the behavior of *S-Solve* on a single node 1, we consider the first rows of each of steps in Figure 2. In each step, the row represents the states of the solve array after that step. Specifically, each location d in the array contains the current state of the path expression $PE(1, d)$. Before step 1, *S-Solve* initializes $SA[1]$ with $\langle 1, 1, \varepsilon\rangle$ and $SA[v]$ with $\langle v, v, \emptyset\rangle$ for $v \in V_G \setminus \{1\}$. Step 1 processes the first element in PS, $\langle 1, 3, a\rangle$, and produces the path expression $\langle 1, 3, a\rangle$ since the regular expression in current location is ε and $\varepsilon \bullet a = a$ which is then unioned with the current path expression in $SA[3]$ to produce a since $a \cup \emptyset = a$. The result of this iteration is a $(1, 3)$ extension. In a similar manner, step 2 computes $\langle 1, 4, k\rangle$ and stores it in $SA[4]$, resulting in a $(1, 4)$ extension. Step 3 produces $\emptyset$ extension since the current path expression in $SA[3]$ is $\emptyset$ and $\emptyset \bullet b = \emptyset$. The process continues similarly as it steps through the elements of the path sequence.

2.1 Evaluating MSMD Path Queries Using Path Sequences

An MSMD path query is defined as follows:

Definition 1. ***(MSMD Path Query)*** *Given a graph $G = (V_G, E_G, \lambda)$, an MSMD path query is a 2-tuple (S, D) where $S, D \subseteq V_G$ are sets of sources and destinations resp. The result of an MSMD path query is the set $\{PE(s_i, d_j) |$ $s_i \in S \wedge d_j \in D\}$, and $PE(s_i, d_j)$ represents all paths from s_i to d_j in G.*

[24] suggests a straightforward adoption of *S-Solve* algorithm to an MSMD problem: execute *S-Solve* once for each source. We call this algorithm *Iterative MultiSolve* and its complexity is $O(|S|l)$ where l is the length of path sequence. However, when the path sequence resides on disk, this approach incurs high I/O costs from the repeated path sequence scans. In [5], we propose the *M-Solve* algorithm as an improvement over *Iterative MultiSolve* that reduces I/O costs by restructuring the for loops. During SolveStep i in *M-Solve* which processes path expression $PE_i(v_i, w_i)$ from the path sequence, we produce (v_i, w_i) extensions for a subset of sources. Specifically, (v_i, w_i) extensions are produced for those sources that have paths to w_i via v_i. We refer to each SolveStep as an *m-SolveStep*, since each step consists

Table 1: Symbol Table

Symbol	Description
R	Regular expression over edge labels (e.g., $a \bullet c \cup k$)
$L(R)$	Language of the regular expression R
(v_i, w_i, R_i)	The ith PE processed by SolveStep i that has $src = v_i$ and $dest = w_i$, and regular exp. R_i
$PE_i(v_i, w_i)$	Shorthand for the PE (v_i, w_i, R_i)
Descendant SolveStep	m-SolveStep k is a *descendant* SolveStep of step i if $i < k$ and either i) for $PE_i(v_i, w_i)$ and $PE_k(v_k, w_k)$, $w_i = v_k$ (direct descendant) or ii) $\exists$ m-SolveStep j s.t. $i < j < k$ and SolveStep j is a direct descendant of step i and SolveStep k is a descendant of step j.
x *ancestor* y	For SolveStep i whose source node is x, $\exists$ SolveStep j whose source node is y s.t. step j is a descendant SolveStep of step i, then x is called an ancestor of y.
S_i^+	A subset of a set S of query source nodes containing elements that produce (v_i, w_i) extension
$SES_{(n,i)}$	Suffix equivalent set w.r.t. node n at m-SolveStep i
$MQ = (S, D)$	MSMD path query with source set S and destination set D

of multiple s-SolveSteps that each produces an extension for one of the sources.

Example 2. Figure 2 shows the 10-step *M-Solve* process for MSMD query $(\{1, 2, 5\}, V_G)$. Each step is represented by three rows that show the intermediate path expressions computed for the 3 different sources. The "multi-source" processing begins from step 3, which initiates the solve process for source 2 (happens with the first incoming path expression whose source is 2 i.e. $\langle 2, 3, b\rangle$). For subsequent m-SolveSteps, the subset of query sources i.e., sources 1 and 2 produce $(3, 4)$ extension i.e. rows 1 and 2 are filled. Step 8 initiates the solve process for source 5, which is reflected in the first non $\emptyset$ entry i.e., entry 6 in the third row. This represents that so far source 5 only has path to 6. Typically, the solve array maintains for each location a *list* of path expressions (one for each source). However, for ease of exposition, we separate the computations for different sources into different rows.

Discussion. The multiprocessing strategy of *M-Solve* avoids repeated scanning of a path sequence, however, there is some repetition within an m-SolveStep, this occurs when paths from two different sources overlap. E.g., for sources 1 and 2 and destination 5, the m-SolveSteps beginning from 5 will produce **the same extensions for both** 1 **and** 2. Specifically, at step 5, both sources 1 and 2 compute the suffix expression "$\bullet c$" as the extension $(3, 4)$. Similarly, the suffix expression "$\bullet d$" is computed as the extension $(4, 5)$ at step 6 and all subsequent steps also compute "equivalent" extensions. We consider such SolveSteps "redundant" since the same computation is essentially repeated for all sources. Factorizing such redundant computations will enable an approach in which we can compute the shared subpath expression once for a set of "equivalent" sources. Although, both strategies have overall time complexity bounds of $O(|S|l)$, there are salient cost advantages to the approach with factorized computation,

1. it allows the average number of extensions per m-SolveStep to remain close to 1 unlike $|S|$ in the regular *M-Solve* algorithm.

2. it reduces the overhead due to "wasted redundant SolveSteps": wasted SolveSteps do not eventually lead to a destination in the query. Predicting which SolveSteps are wasteful during processing is impractical (requires reachability checks for source-destination pairs at every SolveStep). However, by enabling a *work sharing* approach across the solve processes for the different sources, we can ensure that wasted SolveSteps are not computed repeatedly.

3. HOLISTIC MSMD PATH QUERY EVALUATION

3.1 Foundations

Each step in *M-Solve* can be viewed as a multi-SolveStep (*m-SolveStep*) that encapsulates multiple s-SolveSteps, where each s-SolveStep is associated with some s in S. We now formalize m-SolveSteps in a way that allows explicit capturing of equivalent or redundant extensions in an m-SolveStep using a notion called *suffix equivalence*. Table 1 summarizes terms and notations used in the paper.

DEFINITION 2. ***(m-SolveStep)*** *Given a graph G with path sequence $PS = PE_1(v_1, w_1), \ldots, PE_l(v_l, w_l)$ and an MSMD path query $MQ = (S, D)$ where $S, D \subseteq V_G$, m-SolveStep i processes $PE_i(v_i, w_i)$ and produces a 2-tuple $\langle S_i^+, PES_i^+\rangle$ where*

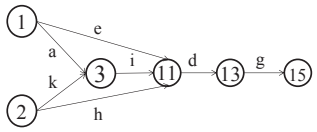
1. *$S_i^+ \subseteq S$ such that $s \in S_i^+ \implies PE(s, w_i) = PE(s, v_i) \bullet PE_i(v_i, w_i) \neq \emptyset$;*
2. *PES_i^+ is the set of path expressions $PE(s_j, w_i)$ such that $s_j \in S_i^+$.*

m-SolveStep i generates a subsets S_i^+ where elements in S_i^+ produce the same (v_i, w_i) extension indicating the reachability of w_i via an intermediate vertex v_i by sources in S_i^+. In example 2, m-SolveStep 3 is represented as a 2-tuple $\langle\{S_4^+ = \{2\}\}, \{PE(2, 4)\}\rangle$. On the other hand, m-SolveStep 5 is represented as a 2-tuple $\langle\{S_5^+ = \{1, 2\}\}, \{PE(1, 4), PE(2, 4)\}\rangle$ because sources 1 and 2 produce the same extension $(3, 4)$.

Since an (v_i, w_i) extension amounts to extending paths with suffix subpaths (those represented in $PE_i(v_i, w_i)$), the sources in S_i^+ can be thought of as being "equivalent" in terms of their suffix paths. We make this notion more precise in the following definition.

DEFINITION 3. ***(Suffix Equivalence $\equiv$)*** *Source node s_p is suffix equivalent to source node s_q w.r.t. extension (v_i, w_i), denoted as $s_p \equiv_{(v_i,w_i)} s_q$, if $s_p, s_q \in S_i^+$. Further, $s_p \equiv_{(v_i,w_i)} s_q \implies s_p \equiv_{(v_j,w_j)} s_q$ for m-SolveStep j which is a descendant m-SolveStep of i. (The implication that the suffix equivalence of sources in S_i^+ is preserved over all descendant m-SolveSteps of i holds due to the transitivity of path connectivity.) In our example, 1 and 2 are suffix equivalent w.r.t. $(3, 4)$ and therefore are suffix equivalent w.r.t. all extensions in descendant m-SolveSteps of step 5, which are $(4, 5), (4, 7), (5, 6), (5, 7)$, and $(7, 8)$. 1 and 2 share multiple common suffix path expressions $PE(3, 8), PE(4, 8), PE(5, 8)$, and $PE(7, 8)$.*

PS = { ‹ 1, 3, a ›, ‹ 1, 11, e ›, ‹ 2, 3, k ›, ‹ 2, 11, h ›, ‹3, 11, i ›, ‹ 11, 13, d ›, ‹ 13, 15, g › }



(a)

Step i & $PE_i(v_i, w_i)$		$SES_{(v_i,h)}$	$SES_{(w_i,k)}$	$SES_{(w_i,i)} = SES_{(v_i,h)} \oplus_\theta SES_{(w_i,k)}$
1	$PE_1(1, 3)$	$SES_{(1,0)} = \{1\}_1$	$SES_{(3,0)} = \emptyset$	$SES_{(3,1)} = \{1\}_1$
2	$PE_2(1, 11)$	$SES_{(1,0)} = \{1\}_1$	$SES_{(11,0)} = \emptyset$	$SES_{(11,2)} = \{1\}_1$
3	$PE_3(2, 3)$	$SES_{(2,0)} = \{2\}_2$	$SES_{(3,1)} = \{1\}_1$	$SES_{(3,3)} = \{\{1\}_1, \{2\}_2\}_3$
4	$PE_4(2, 11)$	$SES_{(2,0)} = \{2\}_2$	$SES_{(11,2)} = \{1\}_1$	$SES_{(11,4)} = \{\{1\}_1, \{2\}_2\}_{11}$
5	$PE_5(3, 11)$	$SES_{(3,3)} = \{\{1\}_1, \{2\}_2\}_3$	$SES_{(11,4)} = \{\{1\}_1, \{2\}_2\}_{11}$	$SES_{(11,5)} = \{\{1\}_1, \{2\}_2, \{1\}_1, \{2\}_2\}_3\}_{11}$
6	$PE_6(11, 13)$	$SES_{(11,5)} = \{\{1\}_1, \{2\}_2, \{\{1\}_1, \{2\}_2\}_3\}_{11}$	$SES_{(13,0)} = \emptyset$	$SES_{(13,6)} = \{\{1\}_1, \{2\}_2, \{\{1\}_1, \{2\}_2\}_3\}_{11}$
7	$PE_7(13, 15)$	$SES_{(13,6)} = \{\{1\}_1, \{2\}_2, \{\{1\}_1, \{2\}_2\}_3\}_{11}$	$SES_{(15,0)} = \emptyset$	$SES_{(15,7)} = \{\{1\}_1, \{2\}_2, \{\{1\}_1, \{2\}_2\}_3\}_{11}$

(b)

Figure 3: (a) A graph G and its PS (b) An example of SES computation for $mq = (\{1, 2\}, \{11, 15\})$ for G

Exploiting suffix equivalence can enable reduction in cost of MSMD path query evaluation. Specifically, if the computation of suffix extensions can be shared by "equivalent" sources, then recomputations of the same extension for all such sources can be avoided. More precisely, given an MSMD path query $MQ = (S, D)$ and l_1 number of shared extension steps (i.e., the number of SolveSteps) shared by $S' \subseteq S$, then the suffix extension computations can be shared to reduce the cost of processing by a factor of $(l_1 \times (|S'| - 1) - |S'|)/(|S'| \times l_1)$, from $O(|S'| \times C_{bpe} \times l_1)$ to $O(C_{bpe} \times (|S'| + l_1))$ where C_{bpe} is the cost of a single extension.

3.2 Sharing SolveSteps Across "Suffix Equivalent Sets" in M-Solve

To maximize the amount of shared computation amongst suffix equivalent nodes, it is important to be able to identify the earliest possible m-SolveStep for which the elements of an S_i^+ become suffix equivalent i.e. the smallest i' such that m-SolveStep i is a descendant of m-SolveStep i' and $S_i^+ = S_{i'}^+$. We will refer to m-SolveStep i' as the *anchor m-SolveStep* for S_i^+ since its source node (*minimal anchor node*) is the origin of the longest shared suffix computations in S_i^+. For example, m-SolveStep 5 in Figure 3 is the anchor m-SolveStep for S_6^+ since SolveStep 5 is the lowest step such that $S_5^+ = S_6^+ = \{1, 2\}$ and step 6 is the descendant step of 5. Also, node 3 is the anchor node for S_6^+. However, the problem of identifying and managing shared suffixes is not that simple. This is because the nodes in S_i^+ may have multiple "interacting" suffix equivalent relationships when some of the prefix paths originating from nodes in S_i^+ do not share the longest possible suffix paths. In this case, there is a shorter suffix that is shared by all prefixes in S_i^+ but a longer suffix rooted at anchor node that is shared by a subset of prefixes i.e. there is a "conflict" in their suffix equivalence relationship. For example, in Figure 3, a shorter suffix $\langle 11, 15, d \bullet g \rangle$ is shared by 4 prefixes: $\langle 1, 11, a \bullet i \rangle$, $\langle 1, 11, e \rangle$, $\langle 2, 11, k \bullet i \rangle$, and $\langle 2, 11, h \rangle$. However, a longer suffix $\langle 3, 15, i \bullet d \bullet g \rangle$ is shared by 2 prefixes $\langle 1, 3, a \rangle$ and $\langle 2, 3, k \rangle$. We call the origin node of the shorter suffix a *conflict anchor node*. In the above example, node 11 is a conflict anchor node for $\{1, 2\}$.

To develop an efficient representation of nodes that are suffix equivalent, we can conflate information about all pairs of sets S_i^+ and S_j^+, $1 \leq i, j \leq l$ (length of path sequence), $i \neq j$ and $w_i = w_j$. For example, in Figure 3, $S_1^+ = \{1\}$ and $S_3^+ = \{2\}$ both have w_i as node 3, thus can be conflated. This will switch the perspective of suffix equivalence to be in terms of a node n in the graph, rather than in terms of m-SolveSteps. For example, conflated S_1^+ and S_2^+ i.e., $\{1, 2\}$ can be seen as the suffix equivalent set w.r.t. node 3. In effect, conflating will summarize suffix equivalence relationships between nodes that reach n, allowing us to have $|V_G|$ suffix equivalent node sets or *Suffix equivalent Sets* rather than l sets. However, this representation must also capture information about conflict and anchor nodes as well as their precedence relationships. For example, the information that the suffix equivalent set $\{1, 2\}$ has node 3 as an anchor node also needs to be captured by the desired representation.

DEFINITION 4. ***(Suffix Equivalent Set (SES))*** *Given a graph G and a set S of query sources, a Suffix Equivalent Set $SES_{(w_i,i)}$ w.r.t. node $w_i \in V_G$ at m-SolveStep i is a labeled set $\{X\}_y$ where $y \in V_G$ is referred as a label of the suffix equivalent set representing the anchor node of $SES_{(w_i,i)}$, and X is either a subset of S or a set of suffix equivalent sets. For example, $SES_{(3,3)} = \{\{1\}_1, \{2\}_2\}_3$. This represents the fact that sources 1 and 2 are suffix equivalent w.r.t. node 3 at step 3.*

Since labels associated with each nested set represent an anchor node, a desirable property of an SES is that its nesting structure preserves the structure of path prefixes. This ensures that reassembling the subpaths from SES structure information produces valid path structures.

DEFINITION 5. ***(Prefix-Preserving Property of SESes)*** *The prefix-preserving property of an SES implies that if we generate a set $AL = \{al_1, \ldots, al_m\}$ where al_i is a sequences of anchor nodes (i.e., labels) ordered from the innermost to the outermost level, and 1) every anchor node is represented in some al_i; 2) $\forall al_i = (n_1, \ldots, n_t)$ is ordered topologically; 3) $\forall al_i, \exists al_j \in AL$ s.t. $\mathsf{suffix}(al_i) = \mathsf{suffix}(al_j)$.*

The inclusion relationships between suffix equivalent sets that naturally arise due to the transitivity of suffix equivalence enable us to reuse previously computed suffix equivalent sets. This results in an incremental computation strategy that aligns naturally with the execution of *M-Solve*. This leads to an inductive definition of suffix equivalent sets w.r.t. a node n in a way that updates their states over m-SolveSteps i in which n is the w_i as follows:

i) $i = 0$: $SES_{(s,0)} = \{s\}_s, \forall s \in S$

ii) $i > 0$: $SES_{(w_i,i)} = SES_{(w_i,k)} \oplus_\theta SES_{(v_i,h)}$, where $1 \leq k, h \leq i$ and steps k and h are the most recent m-SolveSteps

that update $SES_{(w_i,k)}$ and $SES_{(v_i,h)}$ resp. and $\oplus_\theta$ is a combination operator which we define shortly.

In case i), $SES_{(s,0)} = \{s\}_s$ because at m-SolveStep 0, node s is *only* suffix equivalent to itself and there is no anchor node besides itself since no paths have been computed.

To define the $\oplus_\theta$ operator in case ii), we assume the existence of the following functions defined on suffix equivalent sets. For $SES_{(n,i)} = \{X\}_y$ and $SES_{(m,j)} = \{U\}_t$,

- $\mathsf{add}(SES_{(n,i)}, SES_{(m,j)})$ returns $\{X, SES_{(m,j)}\}_y$;
- $\mathsf{label}(SES_{(n,i)})$ returns the label y;
- $\mathsf{isLabel}(SES_{(n,i)}, r)$ returns $True$ if $r = y$; otherwise, returns $False$;
- $\mathsf{isEmpty}(SES_{(n,i)})$ returns $True$ if $SES_{(n,i)} = \emptyset$; otherwise, returns $False$.

DEFINITION 6. *($\oplus_\theta$ **Operation**) Assume $SES_{(v_i,h)}$ is suffix equivalent set w.r.t. v_i which is last updated at SolveStep h and $SES_{(w_i,k)}$ is suffix equivalent set w.r.t. w_i which is last updated at SolveStep k, then,*
$SES_{(v_i,h)} \oplus_\theta SES_{(w_i,k)} =$

$$\begin{cases} SES_{(v_i,h)} & \mathit{isEmpty}(SES_{(w_i,k)}) \\ \{SES_{(w_i,k)}, SES_{(v_i,h)}\}_{w_i} & \neg \mathit{isLabel}(SES_{(w_i,k)}, w_i) \\ \mathsf{add}(SES_{(w_i,k)}, SES_{(v_i,h)}) & \end{cases}$$

If the $\oplus_\theta$ operator can enforce the prefix-preserving property at each m-SolveStep, then integrating the operator at every m-SolveStep can guarantee the prefix-preservation of all SESes at the end of final m-SolveStep.

LEMMA 1. *Assume $\exists SES_{(w_i,k)}$ and $SES_{(v_i,h)}$, where $1 \leq k, h \leq i$ and steps k and h are the most recent m-SolveSteps that update $SES_{(w_i,k)}$ and $SES_{(v_i,h)}$ resp. At m-SolveStep i, $SES_{(w_i,i)} = SES_{(v_i,h)} \oplus_\theta SES_{(w_i,k)}$ is prefix-preserving.*

PROOF. To prove that $SES_{(v_i,w_i)}$ is prefix-preserving, there are three cases that need to be considered.

Condition (1), $SES_{(w_i,k)} = \emptyset \implies \nexists$ m-SolveStep i' s.t. $i' < i$ and $w_i = w_{i'}$. This means that so far node w_i has not been reached by any other node except v_i. Therefore, all nodes including those that are suffix equivalent w.r.t v_i reach w_i via v_i due to the transitivity of path connectedness. Since there is no new branch to w_i, w_i is not an anchor node. For $SES_{(w_i,i)}$, w_i is not an anchor node and the anchor node for $SES_{(w_i,i)}$ is the same as $SES_{(v_i,h)}$.

For the second and third conditions of $\oplus_\theta$, some SES has already been associated with w_i from a previous m-SolveSteps. We will need to update the SES to include new sources and/or new anchor node sequences to reflect any new prefix paths depending on two possibilities: either w_i has already been identified as an anchor node or w_i is *just* being identified as an anchor node. In the former, the nesting level does not change and we merely need to add v_i's SES into the current outermost level that has w_i as anchor node. (This represents new set of prefixes that merge at w_i). Nesting in this way preserves the ordering in which these nodes will appear on a path. In the latter case, we need to increase the nesting depth by creating a new SES with the two SESes nested as elements and make w_i the label for the outermost level. In this case, since the anchor nodes for $SES_{(v_i,h)}$ and $SES_{(w_i,k)}$ meet at w_i, nesting them in the new SES preserves the order of these nodes.

The second and third condition address the two situations when both SESes (from left and right operands) are non-empty. First condition addresses condition with $SES_{(w_i,k)}$ as empty. We ignore the case of both operands being empty since we initialize the SESes for each node s in query source set (left operand) with $\{s\}_s$ at the beginning. Subsequently, the destination node (right operand) of the previous step serves as the source node (left operand) for the next step. □

Example 3. Figure 3(b) shows the first 5 steps of computing SESes for $mq = (\{1,2\}, \{11,15\})$ given the graph and its path sequence in Figure 3(a). Initially (i.e., $i = 0$), we have $SES_{(1,0)} = \{1\}_1$ and $SES_{(2,0)} = \{2\}_2$. *SESes* computed at m-SolveSteps 1 to 5 are listed in Figure 3(b). As shown here, m-SolveStep 1 propagates $SES_{(1,0)}$ i.e. $\{1\}_1$ to node 3 leading to $SES_{(3,1)} = \{1\}_1$. On the other hand, at m-SolveStep 3, source 2 meets source 1 at node 3, thus, we nest $SES_{(2,0)}$ and $SES_{(3,1)}$ into $SES_{(3,3)}$ and update its label to 3 resulting in $SES_{(3,3)} = \{\{1\}_1, \{2\}_2\}_3$.

3.3 Implementation

Our approach for MSMD path query evaluation with suffix computation sharing is a two-phase algorithm ***PrefixSolve*** that consists of algorithms *sesSolve* and *solveSelectedDest*. *sesSolve* (Algorithm 1) extends the original *M-Solve* algorithm by integrating the management of suffix equivalent sets as well as corresponding prefix and suffix sub path expressions and their associations. *solveSelectedDest* (Algorithm 2) reconstructs complete path information for destination nodes that are found to be reachable from some source using information about any shared suffixes that reach those destinations.

3.3.1 The *sesSolve* Algorithm

For each node v, *sesSolve* maintains two lists $PrefixList$ and $SuffixList$ in $SA[v]$ that store prefix and suffix path expressions respectively and a map that associates node v to the current state of v's SES i.e. $SES_{(v,i)}$. *sesSolve* initializes the $SES_{(s,0)}$ with $\{s\}_s$ and $PrefixList$ with $\langle s, s, \varepsilon\rangle$ for $s \in S$ and stores them in $SA[s]$ (lines 1-3). For $s' \in V_G \setminus S$, it sets $SA[s'].SES_{(s',0)}$ and $SA[s'].PrefixList$ to $\emptyset$ (lines 4-5). At m-SolveStep i (i.e., the ith iteration), *sesSolve* mainly performs two tasks: 1) computing prefix path expressions or storing suffix path expressions; and 2) updating suffix equivalent set. Task 1 is performed by considering two cases:

- Case 1 (lines 9-10): $SES_{(v_i,i)} = \{s\}_s$, which means that at step i, only s has been found to have a path to v_i. There are no redundant constituent steps in m-SolveStep i, and thus the extension (v_i, w_i) is produced for s in the same way as the *S-Solve* resulting in prefix path expression $PE(s, w_i)$ stored in $SA[w_i]$.
- Case 2 (lines 11-12): $SES_{(v_i,i)}$ is not a singleton, which means that a set of suffix equivalent sources all reach v_i and share $PE_i(v_i, w_i)$ as suffix path expression. Thus, the algorithm inserts $PE_i(v_i, w_i)$ into $SA[w_i].SuffixList$.

Then, *sesSolve* updates $SES_{(w_i,i)}$ (lines 13-19) according to the definitions 4 and 5.

Example 4. Given $mq = (\{1,2\}, \{11,15\})$ in Figure 3, after initialization we have $SES_{(1,0)} = \{1\}_1$ and $SES_{(2,0)} = \{2\}_2$ stored in $SA[1]$ and $SA[2]$ resp. At step 1, since $SES_{(1,0)}$ is singleton, *sesSolve* produces $(1,3)$ extension resulting in prefix path expression $PE(1,3) = \langle 1, 3, a\rangle$. We update $SES_{(3,1)} = \{1\}_1$. SESes at steps 2 to 4 are computed similarly and shown in Figure 3(b). In addition, we compute prefix path

Algorithm 1: sesSolve

Input: $psIndex, S$;
Output: An array SA to store SESes, $PrefixList$, and $SuffixList$ for node d at location d ;

1 **foreach** $s \in S$ **do**
2 $SA[s].SES_{(s,0)} \leftarrow \{s\}_s$;
3 $SA[s].PrefixList.add((s, s, \varepsilon))$;
4 **foreach** $s' \in V(G) \setminus S$ **do**
5 $SA[s'].SES_{(s',0)} \leftarrow \emptyset$;
6 $i = 1$;
7 **while** $PE_i(v_i, w_i) \leftarrow psIndex.getNext()$ *not NULL* **do**
8 $src \leftarrow$ label$(SA[v_i].SES_{(v_i,i)})$;
9 **if** $|SA[v_i].SES_{(v_i,i)}| = 1$ **then**
10 $SA[w_i].PrefixList.add(SA[v_i].PrefixList.get(src) \bullet PE_i(v_i, w_i) \cup PE(src, w_i))$;
11 **else if** $|SA[v_i].SES_{(v_i,i)}| > 1$ **then**
12 $SA[w_i].SuffixList.add(PE_i(v_i, w_i))$;
//Handle condition isEmpty$(SES_{(w_i,k)})$
13 **if** $SA[w_i].SES_{(w_i,i)} = \emptyset$ **then**
14 $SES_{(v_i,h)}$;
//Handle condition isLabel$(SES_{(w_i,k,w_i)})$
15 **else**
16 **if** *label*$(SA[v_i].SES_{(v_i,i)}) = w_i$ **then**
17 add$(SES_{(w_i,k)}, SES_{(v_i,h)})$;
//Handle condition ¬isLabel$(SES_{(w_i,k,w_i)})$
18 **else**
19 $\{SES_{(w_i,k)}, SES_{(v_i,h)}\}_{w_i}$;
20 $i = i + 1$;
21 **return** SA ;

Algorithm 2: solveSelectedDest

Input: $SA[d]$, where $d \in D$;
Output: $SA[d]$ consisting of complete path expressions for all sources to d ;

1 $PQ \leftarrow \emptyset$;
2 $SA[d].bPE \leftarrow (d, d, \varepsilon)$;
3 $PQ.insert(SA[d])$;
4 **while** $PQ.size > 0$ **do**
5 $pq \leftarrow PQ.dequeue()$;
6 $cID \leftarrow PQ.getID()$;
7 $y \leftarrow label(pq.SES)$;
8 **if** $y \neq cID$ **then**
9 **while** $y \neq sID$ **do**
10 **foreach** *PE with source sID in* $pq.SuffixList$ **do**
11 $SA[sID].bPE \leftarrow pq.SuffixList.get(sID) \bullet pq.bPE \cup SA[d].PrefixList.get(sID)$;
12 $PQ.insert(SA[sID])$;
13 **else**
14 **foreach** *nested ses with label z in* $pq.SES$ **do**
15 **if** $ses = \{s\}_s$ **then**
16 $PE(s, d) \leftarrow SA[y].PrefixList.get(s) \bullet SA[y].bPE \cup SA[d].PrefixList.get(s)$;
17 **else**
18 $SA[z].bPE \leftarrow SA[y].SuffixList.get(z) \bullet SA[y].bPE \cup SA[d].PrefixList.get(z)$;
19 $PQ.insert(SA[z])$;
20 **return** SA ;

expressions $PE(1, 11) = \langle 1, 11, e\rangle$, $PE(2, 3) = \langle 2, 3, k\rangle$, and $PE(2, 11) = \langle 2, 11, h\rangle$. Then, at step 5, since $SES_{(3,3)}$ is not singleton, which satisfies the condition on line 11, we only save $PE(3, 11) = \langle 3, 11, i\rangle$ in $SA[11]$ instead of producing $(3, 11)$ extension for 1 and 2.

At the end of *sesSolve*, each non-empty element of SA maintains a suffix equivalent set and a set of prefix and suffix sub path expressions. *solveSelectedDest* uses suffix equivalent sets to determine associations between suffix and prefix path expressions and concatenates them to produce complete path expressions for source-destination combinations.

3.3.2 *The* solveSelectedDest *Algorithm*

To eliminate the wasted computations, we only run *solveSelectedDest* for query destinations. Further, given some profitable ordering scheme used for graphs e.g., topological ordering of its strong components, we can select the order of destinations to run thoughtfully, so that repeated computations do not arise across computations for different destinations. For the query mq in Example 3, running node 11 before node 15, leads to repetition for some of the computation that has been done for 11 because the path expression from source to 15 contains the one from source to 11 as a subexpression. We can avoid this by proceeding in reverse topological order from destination nodes. In implementation, we keep nodes to be processed in order using a priority queue. The algorithm uses BFS strategy to explore the nodes that reach d in a backward direction and processes them in a specific order i.e., the element with higher topological order has higher priority. For each node cID being processed, the algorithm does two things:

- assembling suffix path expressions; specifically, it computes $SA[sID].bPE$ representing paths from sID to d by concatenating $PE(sID, cID)$ (currently retrieved suffix sub path expression) with $SA[cID].bPE$ which is the already assembled suffix path expression for cID; Note that, initially, we have $SA[d].bPE = \langle d, d, \varepsilon\rangle$.
- concatenating the prefix path expressions in $SA[cID].PrefixList$ with $SAcID.bPE$ if there exist any prefixes for cID to compute path expressions for d.

During each step, there are two cases need to be considered: (1)(lines 9-12) if current node does not have prefixes associated with it, then the algorithm only assembles the suffix path expressions for each retrieved node until it encounters the node whose prefix list is not empty; (2)(lines 13-19) otherwise, the algorithm first assembles the suffix path expressions using retrieved suffix sub path expression and then computes path expressions for d. At the end of the algorithm, $SA[d].bPE$ stores all complete path expressions for d from sources that reach d.

Example 5. Considering the destination 11, at initialization, we have $SA[11].bPE = \langle 11, 11, \varepsilon\rangle$. The algorithm starts by processing $SA[11]$. From example 3, we have that SES of 11 has label 11 satisfying the condition $y = cID$. The algorithm then retrieves and processes its nested SESes. For singleton SESes, e.g., $\{1\}_1$ (i.e., source 1 reaches 11 via a prefix sub path expression $PE(1, 11)$), it computes $PE(1, 11)$ by concatenating $PE(1, 11)$ with $SA[11].bPE$ resulting in $PE(1, 11) = \langle 1, 11, e\rangle$. Similarly, we compute $PE(2, 11) = \langle 2, 11, h\rangle$. For non-singleton SES i.e., $SES_{(3,3)}$, the algorithm computes $SA[3].bPE$ by concatenating the suffix path expression i.e., $PE(3, 11)$ with $SA[11].bPE$ resulting in $SA[3].bPE = \langle 3, 11, i\rangle$. It then processes $SA[3]$ by retrieving the nested SESes in $SES_{(3,3)}$ and processing them

Table 2: Properties of the Datasets

	BioCyc	SP2B	BSBM
# of Nodes	1,160,709	1,571,149	1,154,367
# of Edges	5,278,504	9,053,235	7,757,574
# of PEs	4,192,419	5,798,502	4,867,370
# of SCCs	105,014	1,569,530	1,154,332

individually resulting in $PE(1,11) = \langle 1,3,a\rangle \bullet SA[3].bPE = \langle 1,11,a \bullet i \cup e\rangle$ and $PE(2,11) = \langle 2,11,k \bullet i \cup h\rangle$.

THEOREM 1. *At the end of sesSolve, all the shortest complete prefixes to one anchor node have been computed and all path expressions that comprise suffixes have been recorded in the topological order. solveSelectedDest assembles prefixes with associated suffixes correctly. (Proof omitted for brevity)*

To further improve performance, we integrated a **prefetching strategy** to decrease the latency of accessing a path expression during *M-Solve*. This is achieved by prefetching a set of path expressions into a cache using a separate thread, which ensures that the path expressions are in-memory when needed. Retrieving a subsequence of path expressions also reduces latency by reducing the overall seek time.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the scalability and performance of the different approaches for MSMD path queries. Among the algebraic approaches, we compared 3 variants with different optimization strategies: *Iterative Multisolve* (*ITRMS*), *M-Solve* (*MSOLVE*), and *PrefixSolve* (*PRFS*). Additionally, we compared an extended version of *PRFS* that enables a prefetching strategy (*PRFSPF*). We also implemented 1 representative navigational evaluation technique *DFSS* that is based on depth-first-search that executes on disk-resident graphs. To minimize the random I/O access patterns typically generated by navigational style algorithms, we used a storage model that clusters nodes on disk based on disconnected subgraphs of a graph. The subgraphs were indexed using BerkeleyDB and for each pair of query source-destination nodes, the associated subgraphs were loaded into memory, and the *DFSS* algorithm was executed. Since join-based techniques [18, 15] address shortest or bounded length paths and are not optimized for multiple sources and destinations, we do not compare against them in this evaluation. We also do not include algorithms requiring memory-based graphs [11, 26].

Setup. All experiments were conducted on a machine with 2.33GHz Intel Xeon running on Linux, with a 2.6.18 kernel with maximal runtime memory of JVM was set to 15G. All algorithms were implemented using Java 1.6 and Berkeley DB Java Edition was used for storage and B+ tree indexing with a cache size of 8G. Each query was executed three times without dropping the caches and the average execution time was measured across three runs.

Datasets. Two synthetic datasets generated using RDF benchmark generators BSBM [2] and SP2B [3] were used for scalability evaluation. Additionally, a subset of the real-world data collection Biocyc [1] that consists of 1763 databases describing the genome and metabolic pathways of a single organism was used. The path sequences for datasets were precomputed and loaded into BerkeleyDB prior to query processing. Table 2 summarizes the properties of the datasets.

Table 3: PEs and Query Time on BioCyc Dataset

	MSOLVE		PRFS	
# of Nodes	P-Exps	Time	P-Exps	Time
100K	112,219	116	293	6
200K	227,122	269	472	11
400K	454,387	662	472	31
600K	–	–	7,193	82
850K	–	–	7,193	85
1100K	–	–	15,991	86

4.1 Scalability Evaluation

We conducted a total of 6 scalability experiments by 1) varying sizes of MSMD path queries (from 20 to 200) with a fixed size of data graphs; 2) varying sizes of the data graphs (from $100K$ to $1100K$) with a fixed size of MSMD queries (120). Note that the size of a MSMD path query mq'$= (S, D)$ is defined as $|S| + |D|$. For both cases, a subset of nodes from the three datasets were selected as sample queries. The set of nodes were verified to be connected.

Figure 4(a)-(c) show the results of 1) for BioCyc, SP2B, and BSBM datasets, resp, while Figure 5(a)-(c) show the results of 2). *Note that missing bars in the charts represent cases where the corresponding algorithms did not finish query evaluation within two hours or ran out of memory before obtaining the results.* The experimental results show overall superiority of the algebraic-based *PRFS* algorithm whose performance gains over other approaches increases with query and data sizes, indicating better scalability. The reasons are as follows.

The query decomposition strategy of *DFSS* results in large I/O costs and repeated exploration of the same search space across multiple subqueries making the total cost prohibitive, particularly for queries with a large number of query nodes. *ITRMS* suffers similar limitations. However, for BSBM dataset, *DFSS* outperforms *ITRMS* for both scalability experiments (see Figure 4(c) and Figure 5(a)). We note that BSBM datasets consist of a large number of star substructures with depth of 1 and the schema graph is small with 10 nodes and 8 edges resulting in low connectivity. This results in a smaller search space for *DFSS* during query processing. On the other hand, the algebraic techniques scan the entire path sequence representing the entire graph. We also notice that for Biocyc dataset, *DFSS* outperforms *ITRMS* when the number of query nodes is no greater than 100. A close observation of the results reveals that some sources and destinations (randomly selected) are not connected. In this case, *DFSS* exploits the indexing strategy to detect the disconnection immediately without exploring the graph (different components have different range of keys). *MSOLVE* shows improved performance by avoiding a potential $|Q|$ (i.e., the number of query nodes) separate disk I/Os for each path expression. The results for *PRFS* shows the benefits of avoiding redundant and wasted computations.

Table 3 shows the total number of path expressions (i.e. path expression computation as a unit of work) computed and execution time of the solve process to highlight the performance characteristics of both algorithms on six data graphs ranging from $100K$ to $1100K$ nodes (BioCyc dataset shown in Figure 5(b)). Missing entries indicate failure to compute results after 2 hours. *PRFS* demonstrates clear benefits by doing less work due to the sharing strategy reflected in the fewer number of path expressions computed.

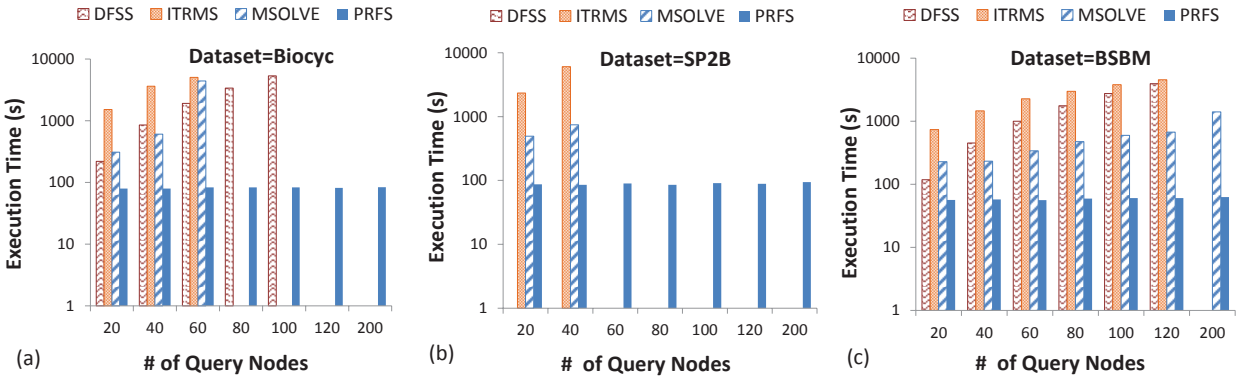


Figure 4: Scalability study with increasing number of query nodes

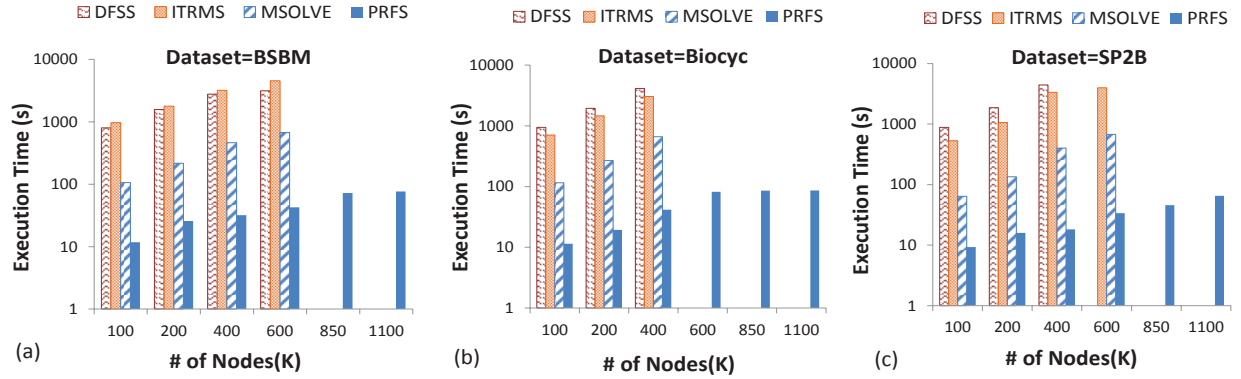


Figure 5: Scalability study with increasing size of graphs

4.2 Performance Evaluation

In this sub-section, we take a closer look at the *PRFS* algorithm. We first study the impact of paths originating from different query nodes but with overlapping suffixes, on the performance of algorithms. Then, we present a study on the benefits of optimization using the prefetching strategy.

4.2.1 Impact of DPO on the Performance

We introduce a metric $DPO(S, D)$ to capture the degree of path overlap between a set of sources S and a set of destinations D defined as follows:

$$DPO(S, D) = \frac{\#\text{ of nodes in the overlap paths between } S \text{ and } D}{\text{total } \#\text{ of nodes in all paths between } S \text{ and } D}.$$

Intuitively, a smaller DPO implies a lower degree of overlap. For this experiment, DPO varied from 0 to 0.67, and the graph size varied from $100K$ to $1100K$ nodes. Test cases were generated by running *M-Solve* with 100 randomly selected query nodes to obtain path information for the nodes. We then select source and destination node sets manually based on connectivity and compute DPO for each case. The total number of query nodes used for this experiment is less than 40 since the algorithms that do not use a sharing strategy have prohibitively long execution times with large query sizes. The results are shown in Figure 6. When $DPO = 0$ (no path overlap in query node paths), *PRFS* offers no advantage and is equivalent to *M-Solve*. On the other hand, for $DPO = 0.26$, 0.5, and 0.67, *PRFS*'s performance advantage is clear for all data sizes.

In addition, we observe that *DFSS* outperforms *PRFS* and other approaches for $DPO = 0$ and 0.26. The reasons are as follows. During the query processing, *DFSS* first checks if the pair of nodes is in the same component, if yes, it loads the component into memory and finds paths using DFS; otherwise, no path exists between them. In this test, since all query nodes reside in the same component, *DFSS* only loads the component into the memory once similar to an in-memory algorithm. However, for $DPO = 0$, *PRFS* will not benefit from saving redundant path expressions computations since there are no common suffix path expressions. For $DPO = 0.26$, there are only 12 pairs of nodes to be computed, so *DFSS* performs well in this case. On the other hand, for the last two cases with high existence of common suffix path expressions, *PRFS* outperforms the other approaches.

4.2.2 Optimization Using Prefetching Strategy

Here, we study the performance of *PRFS* with prefetching optimization, denoted by *PRFSPF*. In this experiment, we introduce a parameter called *prefetching capacity* (PC) to capture the proportion of path expressions prefetched to memory. Figure 7(a) and (b) compare the time cost by *PRFS* and *PRFSPF* on BioCyc and SP2B datasets for $PC = 10\%$ and 50%, resp. For all six data sizes over two datasets, we see an overall performance gain of up to 22% for $PC = 10\%$ and 54% for $PC = 50\%$. Prefetching a good number of path expressions into cache allows m-Solvesteps to retrieve path expressions from memory instead of from disk without requiring the entire path sequence to be maintained in memory. This decouples the latency associated with the *getNext*() call on the B+ tree index. However, in some cases, *PRFS* may not benefit from prefetching. For example, for the BioCyc dataset with $100K$ nodes, when PC is set to 10%, execution time of *PRFSPF* is similar to that of *PRFS*. A close observation reveals that: 1) All

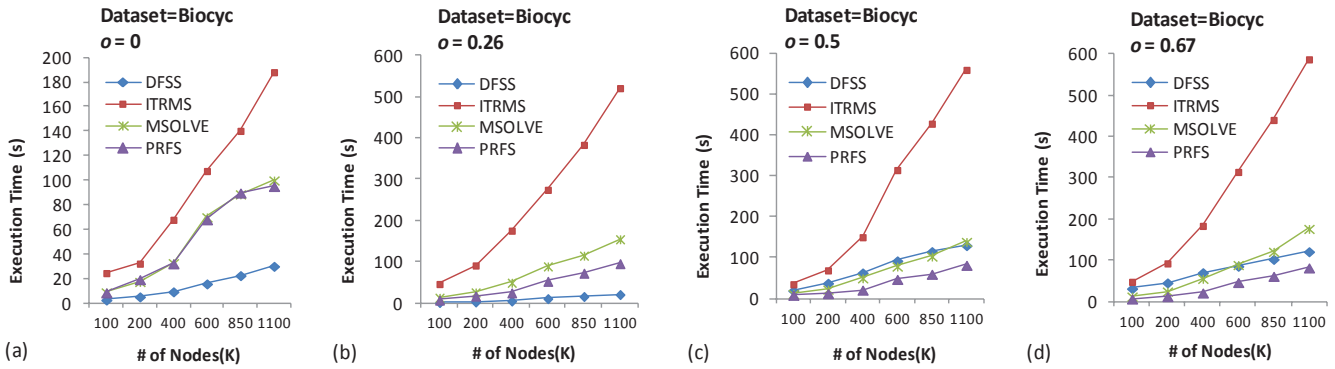


Figure 6: Performance study with varying DPO

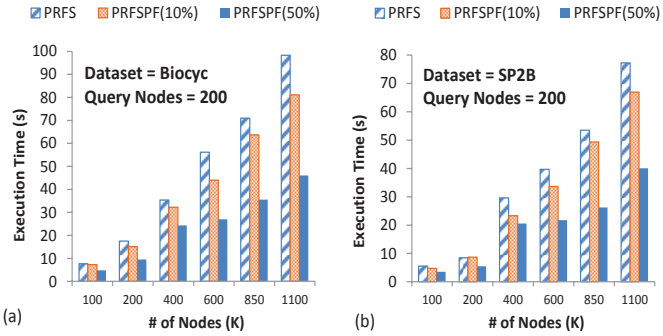


Figure 7: Varying prefetching capacities

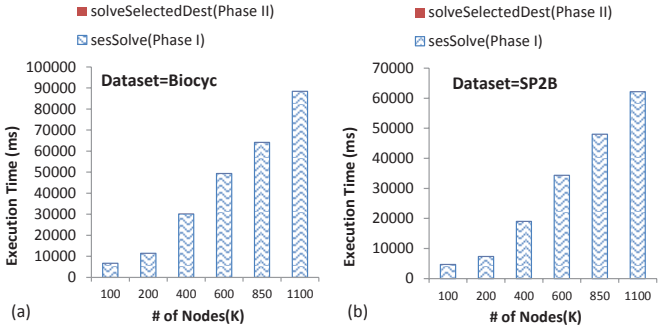


Figure 8: Impact of *PRFS* phases on performance

path expressions in the path sequence are relatively small containing only single edges. This means that a disk page containing B+ tree contains a large number of path expressions reducing overall disk I/O. 2) For all these path expressions, the algorithm produces $\emptyset$ extensions. In other words, the algorithm did not build new path expressions using the prefetched path expressions. Therefore, in this case, prefetching does not improve performance. Assuming a fixed-size cache, prefetching strategy may not always decrease the query execution time based on the above observations. Therefore, a more optimal prefetching strategy is needed in the future.

4.2.3 *Impact of Two Phases of PRFS on the Performance*

Figure 8 shows the impact of different phases of *PRFS* algorithm on the performance. The execution time for *sesSolve* dominates since the reassembly of prefixes and suffixes handled by *solveSelectedDest* is performed on path expressions in the solve array in main memory and there is less work because complete suffix path expressions have been assembled only once and attached to associated prefixes.

5. RELATED WORK

In recent years, there have been several research efforts in providing navigational functionalities for query languages for RDF data. These languages can be classified into two categories: path pattern matching queries [22, 23, 20, 10, 17] that find node pairs connected by paths matching a path pattern; and path "extraction" queries [21, 13, 4, 18, 12] that return paths. We will focus our related work discussion on path extraction queries. For evaluating path extraction queries, some existing systems [13, 12] leverage navigational style approach which is not suitable for large disk-resident data, while some other systems [21, 4, 18] provide solutions from a database perspective by using join-based approach. Specifically, [18], which is the most closely related to our work, provides a full-fledge database solution by proposing a join-based graph operation and cardinality estimation for path triples during the query processing. However, all of the above approaches are different from our work in the following aspects: (1) All these approaches only supported single-source multi-destination (or vice verse) shortest path queries. To use them for MSMD path queries, we need to repeat the process for multiple sources (or destinations), which will lead to inefficiency, especially for disk-resident graphs. (2) Since most of the algorithms are designed for mainly supporting fixed length or shortest path query, it is unclear how to generalize them to solve all path queries.

6. CONCLUSIONS AND FUTURE WORK

This paper extends an algebraic framework to enable support for generalized MSMD path query evaluation. It presents framework for work sharing across subqueries that avoids redundant wasted computations. Future work will address the issue of efficient representation of path expressions during the execution time for further performance improvement.

7. ACKNOWLEDGMENTS

The work presented in this paper is partially funded by NSF grant IIS-0915865.

8. REFERENCES

[1] Biocyc http://download.bio2rdf.org/data/biocyc/.

[2] Bsbm http://www4.wiwiss.fu-berlin.de/bizer/berlinsparqlbenchmark/.

[3] Sp2b http://dbis.informatik.uni-freiburg.de/forschung/projekte/sp2b/.

[4] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending sparql with regular expression patterns (for querying rdf). *J. Web Sem.*, 7(2):57–73, 2009.

[5] K. Anyanwu, M. P. Kumar, and A. Maduko. Structure discovery queries in disk-based semantic web databases. In *SKG*, pages 336–342, 2008.

[6] K. Anyanwu, A. Maduko, and A. P. Sheth. Sparq2l: towards support for subgraph extraction queries in rdf databases. In *WWW*, pages 797–806, 2007.

[7] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In *WWW*, pages 629–638, 2012.

[8] P. Barceló, C. A. Hurtado, L. Libkin, and P. T. Wood. Expressive languages for path queries over graph-structured data. In *PODS*, pages 3–14, 2010.

[9] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[10] L. T. Detwiler, D. Suciu, and J. F. Brinkley. Regular Paths in SparQL: Querying the NCI Thesaurus. *AMIA Annual Symposium proceedings / AMIA Symposium. AMIA Symposium*, pages 161–165, 2008.

[11] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *KDD*, pages 118–127, 2004.

[12] V. Fionda, C. Gutierrez, and G. Pirrò. Extracting relevant subgraphs from graph navigation. In *International Semantic Web Conference (Posters & Demos)*, 2012.

[13] V. Fionda, C. Gutierrez, and G. Pirrò. Semantic navigation on the web of data: specification of routes, web fragments and actions. In *WWW*, pages 281–290, 2012.

[14] G. H. L. Fletcher, M. Gyssens, D. Leinders, J. V. den Bussche, D. V. Gucht, S. Vansummeren, and Y. Wu. Relative expressive power of navigational querying on graphs. In *ICDT*, pages 197–207, 2011.

[15] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang. Relational approach for shortest path discovery over large graphs. *PVLDB*, 5(4):358–369, 2011.

[16] S. Gao, H. Fu, and K. Anyanwu. An agglomerative query model for discovery in linked data: Semantics and approach. In *WebDB*, 2010.

[17] S. H. Garlik, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 query language. http://www.w3.org/TR/sparql11-query/.

[18] A. Gubichev and T. Neumann. Path query processing on very large rdf graphs. In *WebDB*, 2011.

[19] K. Kochut and M. Janik. Sparqler: Extended sparql for semantic association discovery. In *ESWC*, pages 145–159, 2007.

[20] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. A path-based relational rdf database. In *ADC*, pages 95–103, 2005.

[21] M. Przyjaciel-Zablocki, A. Schätzle, T. Hornung, and G. Lausen. Rdfpath: Path query processing on large rdf graphs with mapreduce. In *ESWC Workshops*, pages 50–64, 2011.

[22] A. Seaborne. Rdql - a query language for rdf (member submission). Technical report, W3C, January 2004.

[23] A. Souzis. Rxpath specification proposal.

[24] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.

[25] R. E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, 1981.

[26] H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD*, pages 404–413, 2006.