Efficient processing of containment queries on nested sets

Ahmed Ibrahim
Eindhoven University of Technology
The Netherlands
a.ibrahim@student.tue.nl

George H. L. Fletcher
Eindhoven University of Technology
The Netherlands
g.h.l.fletcher@tue.nl

ABSTRACT

We study the problem of computing containment queries on sets which can have both atomic and set-valued objects as elements, i.e., nested sets. Containment is a fundamental query pattern with many basic applications. Our study of nested set containment is motivated by the ubiquity of nested data in practice, e.g., in XML and JSON data management, in business and scientific workflow management, and in web analytics. Furthermore, there are to our knowledge no known efficient solutions to computing containment queries on massive collections of nested sets. Our specific contributions in this paper are: (1) we introduce two novel algorithms for efficient evaluation of containment queries on massive collections of nested sets; (2) we study caching and filtering mechanisms to accelerate query processing in the algorithms; (3) we develop extensions to the algorithms to a) compute several related query types and b) accommodate natural variations of the semantics of containment; and, (4) we present analytic and empirical analyses which demonstrate that both algorithms are efficient and scalable.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

General Terms

Algorithms, Experimentation

Keywords

nested sets, containment join, inverted file, homomorphism

1. INTRODUCTION

Data sets with nested structure are encountered in a wide variety of practical domains and applications. For example, nested structures occur in: scientific workflows, XML and JSON data management, business process management, web mash-ups, data integration systems, NoSQL document and key-value stores, and complex object data management. Industrial-strength tools for web-scale data storage and analytics, such as Google's Dremel and Yahoo's Pig, are also designed around a nested data model [26, 27].[1]

As a simple illustration of nested structure, Table 1 gives a snippet of a collection S of nested set data. This data set concerns where people live (at the top level of nesting), the set of locales in which they hold driving privileges (at the second nesting level), and, at the deepest nesting level, the set of licenses and vehicle types they are authorized to handle in a particular locale. For example, Tim lives in Boston, and can operate a car in Virginia (USA), where he holds class A and B licenses. This simple set-based abstraction, which we adopt in this paper, captures the basic hierarchical structure of the nested data which occurs in the many concrete applications discussed above.

Table 1: An example of a nested set collection S.

key	*value*
Sue	{London, UK, {UK, {A, B, C, car, motorbike}}}
Tim	{Boston, USA, {USA, VA, {A, B, car}}, {UK,{A, motorbike}}}

A fundamental query pattern for sets is the containment join, i.e., given collection Q with set-valued attribute A (for example, the *value* column of Table 1), retrieve, for each object $q \in Q$, all objects s in a collection S where $q.A \subseteq s.B$, for set valued attribute B. In other words, the join of Q in S on $A \subseteq B$ is defined as:

$$Q \bowtie_{A \subseteq B} S \quad = \quad \{(q, s) \mid q \in Q \wedge s \in S \wedge q.A \subseteq s.B\}. \quad (1)$$

As an example, consider the query: "retrieve all people that live in the USA who have license type A valid for a motorbike in the UK", which we formulate as the nested set $q = \langle query : \{\texttt{USA}, \{\texttt{UK}, \{\texttt{A}, \texttt{motorbike}\}\}\}\rangle$. Evaluating $\{q\} \bowtie_{query \subseteq value} S$ on the database of Table 1 would give us $\{(q, Tim)\}$, where Tim is the record having key Tim.

As an essential set-based query type [3, 22], containment joins also find basic application in diverse areas such as query processing [4] and data mining [29]. Consequently, efficient processing of set containment queries has been heavily investigated. Basic approaches include the use of inverted files, signature files, and hashing to facilitate efficient join

[1]The Dremel team has remarked that "A nested data model underlies most of structured data processing at Google [...] and reportedly at other major web companies" [26].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy
Copyright 2013 ACM 978-1-4503-1597-5/13/03 ...$15.00.

processing (e.g., [15, 24, 25, 28, 35, 34]). To the best of our knowledge, however, all known solutions for processing containment queries are restricted to the special case of *flat* sets, i.e., sets without nesting, whose members are all atomic objects. Clearly, with the widespread use of *nested* data and the fundamental nature of the containment query pattern, efficient solutions for computing *nested* containment are called for. As they are designed specifically for flat set processing, state of the art containment solutions are not immediately applicable for nested sets, especially for those sets having arbitrarily deep nesting structure.

Containment queries are also closely related to the heavily studied tree pattern queries (TPQ) [14] and nested relational join queries [9]. Indeed, nested set containment queries are essentially a hybrid of flat set queries and these query types. While a nested set can be conceptualized as a type of tree (as we do below), all natural tree encodings of nested sets exhibit structure which violate the design of existing TPQ evaluation solutions (e.g., [8, 11, 19, 38]). In particular, these solutions crucially rely on internal tree nodes having exactly one label, whereas tree encodings of nested sets exhibit potentially unlabeled or multi-labeled internal nodes. Although we build in our work on the same mature inverted file data structure used by TPQ solutions for representing XML trees (e.g., the classic position-based node encoding used in [8]), the richer structure of nested sets necessarily requires a fresh re-think of query processing solutions. Indeed, it has already been observed that state-of-the-art TPQ solutions are inadequate even in the special case of flat set containment joins [35]. Finally, solutions for nested relational join processing are designed and optimized for the restricted special case of nested sets having fixed pre-defined input schemas, and hence are not suitable for processing collections of heterogeneous nested sets [9].

Motivated by these observations, in this paper we present the first known solutions for scalable, efficient computation of containment queries on collections of nested sets. In particular, we make the following contributions.

- We highlight the problem of nested set containment and propose two practical solutions. The first algorithm works top down, starting at the outer-most nesting level of the query, and working inwards. The second algorithm works bottom up, starting at the deepest nesting level of the query, and working outwards. We also study caching and filtering mechanisms to accelerate query processing in the algorithms.
- We develop extensions to both algorithms, to handle (1) related query types (such as superset and set-overlap); and, (2) natural variations of the semantics of containment.
- We show that our solutions are efficient and scalable, on a variety of synthetic and real data sets.

Additional features of our algorithms include conceptual simplicity and their use of the widely adopted inverted file data structure. These practical aspects further help to give our solutions strong potential for practical use and impact.

We proceed in the rest of the paper as follows. In the next section, we give our basic definitions. In Section 3, we then present and theoretically analyze our two solutions. This is followed in Section 4 with a discussion of extensions to our algorithms. In Section 5, we then discuss an empirical study of our two algorithms, demonstrating their practical behavior. Finally, we give closing remarks in Section 6.

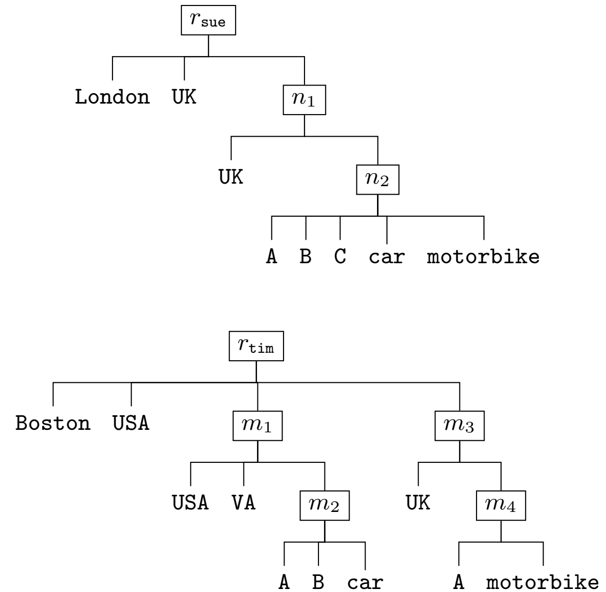


Figure 1: Tree representations of the nested set values associated with keys `Sue` and `Tim`, resp., in collection S of Table 1.

2. PRELIMINARIES

In this section we give basic definitions, and describe the main data structure used in our algorithms.

Data model. In this paper, we consider data objects in the form of finite sets built over some universe of atomic objects (e.g., strings or integers). We impose no restrictions on the cardinality or nesting depth of data objects. As sets, we also assume no ordering in the internal structure of data objects.

The problem. We study how to efficiently compute Equation 1 of Section 1 when Q and S are finite sets of data objects. We assume that both Q and S are too large to fit in internal memory. In particular, we will treat Q as a set of *queries* over which we iterate, and, simplifying Equation 1, focus specifically on the problem of computing

$$q \bowtie_{\subseteq} S \quad = \quad \{(q, s) \mid s \in S \text{ and } q \subseteq s\} \qquad (2)$$

for a given query (i.e., nested set value) $q \in Q$. Note that we will assume that the set-valued attributes of interest in Q and S are fixed, and hence did not explicitly mention attribute names in Equation 2 and will continue to not do so in the sequel.

Notions of containment. We next clarify how we interpret the condition "$q \subseteq s$" of Equation 2, which, as we will see, has several distinct natural semantics. Towards this, we first introduce a handy alternative perspective on nested sets.

A nested set can be viewed as a type of unordered node-labeled rooted tree, where internal (i.e., non-leaf) nodes denote sets and leaf nodes denote atomic values. Figure 1

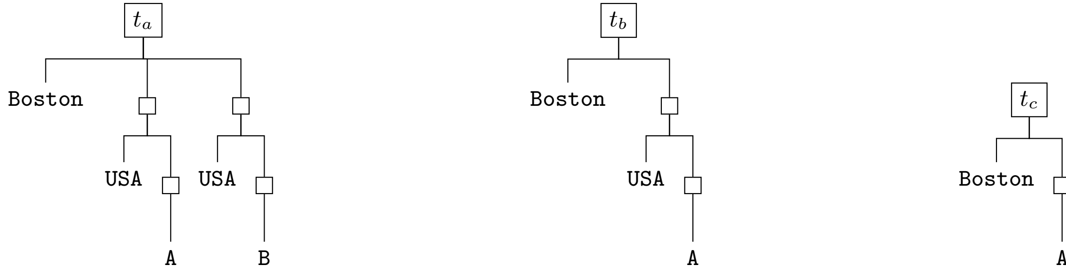


Figure 2: The set rooted at t_a is not $\subseteq_{\text{iso}}$-contained, but is $\subseteq_{\text{hom}}$- (and hence, also $\subseteq_{\text{homeo}}$-) contained, in the set of Figure 1 rooted at $r_{\texttt{tim}}$. The set rooted at t_b is $\subseteq_{\text{iso}}$- (and hence also $\subseteq_{\text{hom}}$- and $\subseteq_{\text{homeo}}$-) contained in $r_{\texttt{tim}}$. The set rooted at t_c is $\subseteq_{\text{homeo}}$- (but not $\subseteq_{\text{hom}}$-, and hence also not $\subseteq_{\text{iso}}$-) contained in $r_{\texttt{tim}}$.

gives tree representations of the set values associated with keys `Sue` and `Tim` from collection S of Table 1. In this representation, arbitrary unique identifiers label distinct internal nodes (we use integer IDs in the implementation of our data structures; see below). For internal node n of tree T, we use the notation $nodes(n)$ to denote the set of non-leaf children of n in T and $\ell(n)$ to denote the set of (labels of) leaf node children of n in T. For example, $nodes(r_{\texttt{tim}}) = \{m_1, m_3\}$ and $\ell(m_1) = \{\texttt{USA}, \texttt{VA}\}$. Clearly, this interpretation of nested sets is well-defined; we omit a formal definition here, due to space limitations. In the sequel, we will move freely between speaking of sets as trees and vice versa.

For flat sets, there is a default natural notion of containment, namely standard subset semantics. Here, to evaluate $q \subseteq s$, we are simply interested in finding an assignment of the atoms of flat set q to the atoms of the flat set s. In nested sets, however, we must not only map the leaf nodes of q to the leaf nodes of s, but we must also map internal nodes of q to internal nodes of s, while ensuring that set nesting is preserved (e.g., the root of q is mapped to the root of s). In other words, for "$q \subseteq s$" to hold, we must be able to map q in a root preserving manner to a subtree of s.

We can identify three basic variants of the notion of subtree containment which arise naturally in practice:

- *subtree isomorphism* ($\subseteq_{\text{iso}}$) — the mapping of internal nodes of q to internal nodes of s is injective and parent-child edges in q must be mapped to parent-child edges in s.[2] In the literature, this is also known as a (root-preserving) unordered path embedding [20] or a top-down, unordered subtree isomorphism [32, 36].

- *subtree homomorphism* ($\subseteq_{\text{hom}}$) — the mapping from the internal nodes of q to the internal nodes of s is not necessarily injective, but parent-child edges in q must be mapped to parent-child edges in s. This is also referred to as an unordered pseudo-tree matching or a tree homomorphism matching [10, 18].

- *subtree homeomorphism* ($\subseteq_{\text{homeo}}$) [10, 18] — the mapping from internal nodes of q to internal nodes of s is not necessarily injective, and parent-child edges in q can be mapped to ancestor-descendant edges in s.[3]

Note that all isomorphic embeddings are homomorphic embeddings, and all homomorphic embeddings are homeomorphic embeddings. Furthermore, both of these inclusions are strict. Figure 2 gives illustrations of these three notions.

We adopt $\subseteq_{\text{hom}}$ as our primary semantics for $\subseteq$. We make this choice because (1) homomorphism is very natural (e.g., corresponds to the semantics of conjunctive query languages such as those at the heart of SQL, search languages, XPath, and SPARQL [7, 11, 13, 14]); and, (2) solutions for homomorphic containment can easily be relaxed to solutions for checking homeomorphic containment as well as extended to the stricter notion of isomorphic containment [10, 18]. We discuss how to adapt our algorithms to compute containment under the alternate iso- and homeo-morphism semantics in Section 4.

Inverted files for nested sets. In the literature on processing containment queries on flat sets, solutions using inverted files as the physical representation of the database have demonstrated robust efficient performance [15, 24, 35].[4] Furthermore, a variety of industrial-strength open-source solutions for building inverted files are available off the shelf, and are widely adopted and used by practitioners.

For these reasons, we have also adopted the inverted file as the basic data structure in our solutions. The key-space of our inverted file is the set of all atomic values occurring in the collection S of Equation 2. The inverted lists associated with these values store information about the location of the values in the collection. In our physical representation of a nested set, all internal nodes are assigned an arbitrary unique integer identifier. The "location" of a leaf node is given by the identifier of its parent, i.e., the ID of the set containing the leaf.

Unlike in a traditional inverted file, it is not sufficient for us to store in the inverted lists just the locations of atomic values. We must also store structural information of the given key value, in order to facilitate navigation in the set hierarchy. Hence, we extend the inverted lists with additional information regarding related internal nodes. In par-

[2]i.e., for all nodes m of q, if m' is a child of m and m is mapped to node n in s, then m' must be mapped to a child n' of n.

[3]i.e., for all nodes m of q, if m' is an internal child of m and m is mapped to node n in s, then m' must be mapped to an internal node n' in the subtree rooted at n. In order to retain hierarchical structure, however, we require that edges from internal nodes to leaf nodes in q must still be mapped to parent-child edges in s. This restriction can easily be lifted, if need be.

[4]Recall that an inverted file is an index data structure which maps each atomic value appearing in the database instance to a sorted list of the locations of the occurrences of the value in the data; cf. [37].

Table 2: The inverted file S_{IF} for the collection S of Figure 1.

atom	*inverted list*
London	$\langle(r_{sue},\langle n_1\rangle)\rangle$
UK	$\langle(m_3,\langle m_4\rangle),(n_1,\langle n_2\rangle),(r_{sue},\langle n_1\rangle)\rangle$
A	$\langle(m_2,\langle\rangle),(m_4,\langle\rangle),(n_2,\langle\rangle)\rangle$
B	$\langle(m_2,\langle\rangle),(n_2,\langle\rangle)\rangle$
C	$\langle(n_2,\langle\rangle)\rangle$
car	$\langle(m_2,\langle\rangle),(n_2,\langle\rangle)\rangle$
motorbike	$\langle(m_4,\langle\rangle),(n_2,\langle\rangle)\rangle$
Boston	$\langle(r_{tim},\langle m_1,m_3\rangle)\rangle$
USA	$\langle(m_1,\langle m_2\rangle),(r_{tim},\langle m_1,m_3\rangle)\rangle$
VA	$\langle(m_1,\langle m_2\rangle)\rangle$

ticular, for a given atomic value a (i.e., a search key) in the collection, the associated inverted list is

$$S_{IF}(\mathtt{a}) \quad = \quad \langle(p_1,C_1),\ldots,(p_n,C_n)\rangle$$

sorted on p_i values, where the p_i's are the identifiers of all locations of leaf nodes labeled a in the collection S, and, for each $1 \le i \le n$, C_i is the sorted listing of $nodes(p_i)$. Table 2 gives the inverted file S_{IF} for the nested sets of Figure 1, representing the collection S of Table 1.

To navigate in the set hierarchy using S_{IF} requires a series of inverted list joins $\bowtie_{\mathrm{IF}}$, defined as

$$L \bowtie_{\mathrm{IF}} L' \quad = \quad \{(p,C') \mid (p,C) \in L \wedge (p',C') \in L' \wedge p' \in C\},$$

where L and L' are inverted lists. For example, to determine in our example collection S which sets are contained in a set having the atom UK, which is in turn contained in a set containing the atom London, we perform navigation one level down into the set hierarchy, from London to UK, as $S_{IF}(\mathtt{London}) \bowtie_{\mathrm{IF}} S_{IF}(\mathtt{UK})$, which we see from Table 2 evaluates to $\langle(r_{sue},\langle n_2\rangle)\rangle$. In other words, there is only one such appropriately nested set in the database, namely n_2 nested within r_{sue}.

3. TWO CONTAINMENT ALGORITHMS

In this section we present two different approaches to solving Equation 2, under the homomorphic semantics for containment, using the inverted file data structure described in the previous section. The first algorithm takes a top-down perspective which starts by evaluating the root node of the query and continues recursively with its children. The second algorithm takes a bottom-up approach which starts exploring the set at its leaf nodes, in a depth-first fashion. For both solutions, we first explain the algorithm via an example, and then give details.

Before we move to the presentation of the algorithms, we make two comments. (1) A naive solution to computing containment of q in S is to apply an off-the-shelf subtree homomorphism algorithm (e.g., [18]) to each pairing (q,s), for $s \in S$. Intuitively, such an approach would be substantially more expensive than processing S in bulk, as we do in our algorithms below, since this would require retrieving every single object from the database. A small empirical study of the naive solution which we undertook confirmed this intuition. (2) Our algorithms, as presented here, assume non-empty sets at each nesting level (e.g., they do not support queries such as $\{\mathtt{A},\{\{\{\mathtt{B}\}\}\}\}$). The algorithms can be directly adapted to handle empty sets, however, following the approach for implementing homeomorphic containment discussed in Section 4.2.

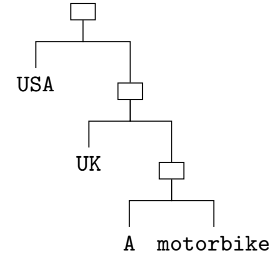


Figure 3: The tree representation of the example query q of Section 1.

3.1 Top-down algorithm

Example run. We illustrate the top-down approach with our query q from Section 1; Figure 3 shows the tree representation of q. Let S_{IF} be the inverted file of Table 2. We proceed from the root of the query as follows.

1. We first retrieve the inverted list of USA and place it in a temporary list R_0, i.e.,

   $$R_0 \quad \leftarrow \quad \langle(m_1,\langle m_2\rangle),(r_{tim},\langle m_1,m_3\rangle)\rangle.$$

   Figure 4a illustrates this step.

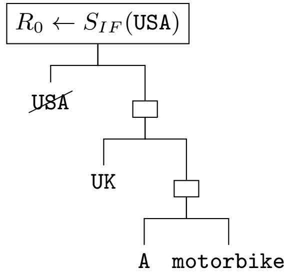
2. We next navigate downwards, by retrieving the inverted list of UK and performing the $\bowtie_{\mathrm{IF}}$-join with the values stored in R_0 to determine which roots have children in this set. We store these values in $R_1 \leftarrow R_0 \bowtie_{\mathrm{IF}} S_{IF}(\mathtt{UK})$, i.e., $R_1 \leftarrow \langle(r_{tim},\langle m_4\rangle)\rangle$. Figure 4b illustrates this step.

3. We continue by retrieving the inverted lists that satisfy the intersection of A and motorbike and navigate from the paths stored in R_1 into this set. We store these values in $R_2 \leftarrow R_1 \bowtie_{\mathrm{IF}}(S_{IF}(\mathtt{A}) \cap S_{IF}(\mathtt{motorbike}))$, i.e., $R_2 \leftarrow \langle(r_{tim},\langle\rangle)\rangle$. Figure 4c illustrates this processing.

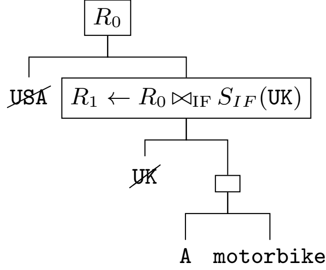
4. We finish processing by propagating all results back up towards the root, evaluating along the way the intersection $\pi_1(R_2) \cap \pi_1(R_1) \cap \pi_1(R_0)$ of the sets of path "heads" (i.e., the first element of each pair) in each of the temporary lists of paths, i.e., $\{(r_{tim})\} \cap \{(r_{tim})\} \cap \{(r_{tim})\} = \{(r_{tim})\}$. The last intersection at the root finishes the computation. Recall from our discussion in Section 1, that q is indeed contained in the record associated with r_{tim}.

The algorithm. We give pseudo code of the top down approach in Algorithms 1 and 2. The algorithm starts with a call to Top-down-containment (Algorithm 1) with a query q and an inverted file encoded database S_{IF}. We compute the set of candidates for the embedding of the root node of the query $root(q)$, and pass them along to Algorithm 2.

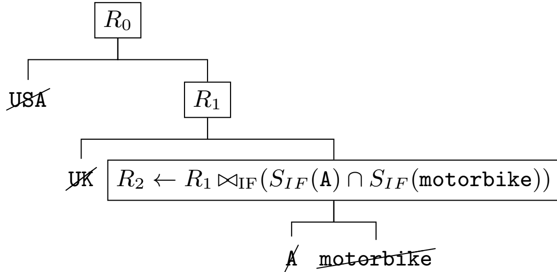
The children of the root (and so on, recursively) are processed in Top-down-interior (Algorithm 2). Input N is the current set of sibling nodes of the query to be processed. If N is empty, we have hit the bottom of a set nesting, and we



(a) Initially, we evaluate (the leaves of) the root of q and store the resulting list in R_0. We cancel out the leaf node USA to indicate that it has now been processed.



(b) Next, we evaluate the internal child of the root, placing the resulting list in R_1.



(c) Finally, we evaluate the inner-most set, and then propagate up the local result R_2. The final result is the intersection of the sets of heads of paths in R_0, R_1 and R_2.

Figure 4: Illustration of the top-down algorithm on the example query of Figure 3.

Algorithm 1 Top-down-containment
input: query q, inverted file S_{IF}
output: $q \bowtie_{\subseteq} S$

1: $P \leftarrow \bigcap_{\ell \in \ell(root(q))} S_{IF}(\ell)$
2: **return** Top-down-interior($nodes(root(q))$, P, S_{IF})

Algorithm 2 Top-down-interior
input: set N of query siblings; set P of current successful paths from roots in S to nodes in N; inverted file S_{IF}
output: roots in P which successfully contain the sets N in this location

1: **if** $N = \emptyset$ **then**
2:    **return** $\{root \mid \exists c : (root, c) \in P\}$
3: **else if** $P = \emptyset$ **then**
4:    **return** $\emptyset$
5: **else**
6:    $Roots \leftarrow \{root \mid \exists c : (root, c) \in P\}$
7:    **for all** nodes $n \in N$ **do**
8:       $Candidates \leftarrow \bigcap_{\ell \in \ell(n)} S_{IF}(\ell)$
9:       $P' \leftarrow P \bowtie_{\text{IF}} Candidates$
10:       $Roots' \leftarrow$ Top-down-interior($nodes(n), P', S_{IF}$)
11:       $Roots \leftarrow Roots \cap Roots'$
12:    **end for**
13:    **return** $Roots$
14: **end if**

return all roots which have successfully covered this subtree of the query (lines 1-2). Input P contains the set of paths in S which have successfully navigated to this subtree. If P is empty, there are no such paths, and we return $\emptyset$ to indicate this (lines 3-4). Otherwise, there are successful paths to this location, and we have more work to do (lines 5-14). On line 6, we compute the set of roots in S which have successfully navigated to this part of the query. Then, in the for-loop at lines 7-12, we process each internal query node $n \in N$. At line 8, we evaluate n. At line 9, we extend the current paths from the root into the retrieved lists, storing these in P'. At line 10, we recur on the children of n. In line 11, we filter out roots which were not able to successfully cover the subquery rooted at n. Finally, we return the subset of roots which were able to successfully cover all of N (line 13).

Analysis. Correctness of the top-down algorithm is straightforward. The worst case running time for evaluating a leaf of q is $\mathcal{O}(|S|)$, i.e., the number of (internal) nodes in the database. Letting $\mathcal{L}_q$ denote the number of leaf nodes of q, the worst case running time of Top-down-containment (Algorithm 1) is then bounded by $\mathcal{O}(\mathcal{L}_q * |S|)$. Letting $\mathcal{N}_q$ denote number of internal nodes of q, the cost breakdown of Top-down-interior (Algorithm 2) is as follows.

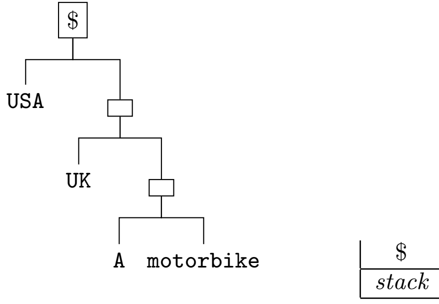
- lines 1-6: P is at most as large as the database, hence, over the course of the full run of the algorithm, cost is bounded by $\mathcal{O}(\mathcal{N}_q * |S|)$.
- line 8: Over the course of the full run of the algorithm, we evaluate this intersection once for each leaf of the query, with worst case cost as before, giving us an upper bound of $\mathcal{O}(\mathcal{L}_q * |S|)$.
- lines 9 and 11: Since both lists are sorted, each of these operations, over the course of the full run of the algorithm, is bounded by $\mathcal{O}(\mathcal{N}_q * |S|)$.

We conclude that the worst-case running time of the top-down algorithm is $\mathcal{O}(|q| * |S|)$.

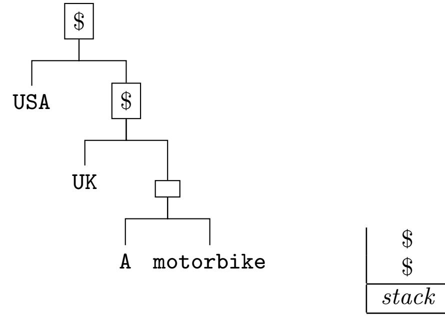
3.2 Bottom-up algorithm

Example run. The second algorithm we introduce evaluates containment in a depth-first fashion, using a stack to store and process intermediate results. The basic idea is to process the subtree under a given query node before processing the node itself. We use the query q of Figure 3 and inverted file S_{IF} of Table 2 to illustrate the bottom-up algorithm, which proceeds as follows.

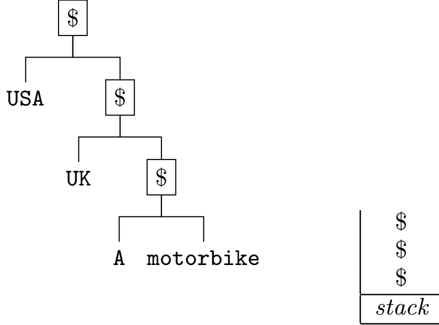
1. We start by initializing an empty stack at the root of the query. Figure 5a shows that a special marker



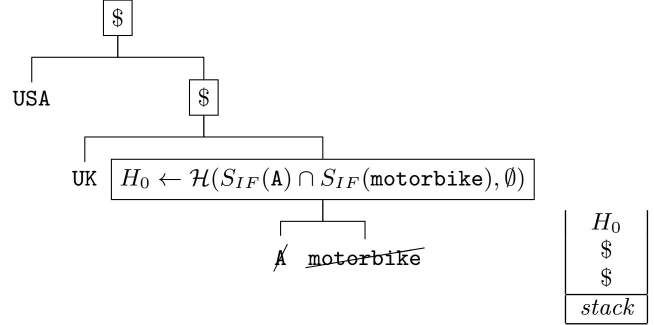
(a) We start at the root, pushing the first marker onto the stack.



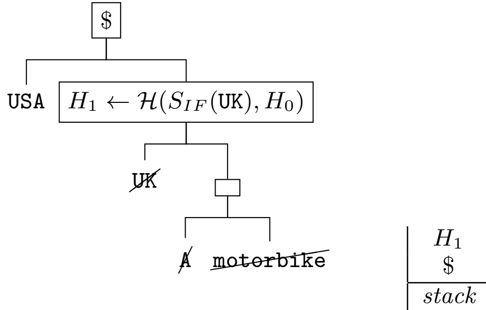
(b) We continue with the child of the root, pushing a second marker onto the stack.



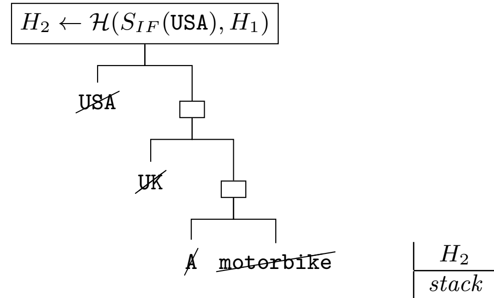
(c) We continue descending, pushing a third marker onto the stack.



(d) We evaluate the inner-most node, pop the stack, and put the local result H_0 onto the stack.



(e) We evaluate and, popping the stack, process the second node, and put the local result H_1 onto the stack.



(f) We process the root and put the final results H_2 on top of the stack.

Figure 5: Illustration of the bottom-up algorithm on the example query of Figure 3.

'$' is pushed onto the stack, as we recur on the root's single internal child. Figure 5b and 5c show that a second and then third marker are pushed onto the stack, as depth-first processing continues. Hitting the innermost nesting level, we (1) evaluate the current node, (2) pop the stack until a '$' marker is reached, collecting all retrieved sets in a set *Lists*, and (3) push onto the stack the set of the heads of all inverted list elements found in the first step which have children in each of the lists in *Lists*. Intuitively, these heads identify nodes in the database which successfully cover the subquery rooted at the current node. Here, $Lists = \emptyset$, as the current node has no internal children which were evaluated earlier in the computation. We place all heads of the evaluation $H_0 \leftarrow \mathcal{H}(S_{IF}(\mathtt{A}) \cap S_{IF}(\mathtt{motorbike}), \emptyset)$ on the stack,[5] i.e.,

$$\begin{aligned} H_0 \leftarrow\ & \text{the set of the heads of all lists in} \\ & S_{IF}(\mathtt{A}) \cap S_{IF}(\mathtt{motorbike}) \text{ which have} \\ & \text{children in each of the sets in } Lists = \emptyset \\ \leftarrow\ & \text{the set of the heads of all lists in} \\ & S_{IF}(\mathtt{A}) \cap S_{IF}(\mathtt{motorbike}) \\ \leftarrow\ & \{m_4, n_2\}. \end{aligned}$$

This step is illustrated in Figure 5d. Note that at this point, r_{tim} and r_{sue} (the root ancestors of m_4 and n_2,

[5] A formal definition of $\mathcal{H}()$ is given below (line 12 of Algorithm 4).

resp.) are both candidates for the full query.

2. Moving up one nesting level, we next process the child of the root. This involves constructing the local set of candidates for embedding this node in the database, namely $S_{IF}(\texttt{UK})$, and popping the stack until the next marker is reached, to construct $Lists = \{H_0\}$. Then, we place onto the stack the set $H_1 \leftarrow \mathcal{H}(S_{IF}(\texttt{UK}), H_0)$ of all "head" nodes in the local list of candidates which have paths into H_0, i.e.,

$$H_1 \quad \leftarrow \quad \{(m_3, n_1)\}.$$

This step is illustrated in Figure 5e. Note that r_{tim} and r_{sue} are still both candidates for q.

3. Moving up to the root, we repeat the same process, constructing local candidates $S_{IF}(\texttt{USA})$, popping the stack to build $Lists = \{H_1\}$, and pushing

$$\begin{aligned} H_2 \quad &\leftarrow \quad \mathcal{H}(S_{IF}(\texttt{USA}), \{H_1\}) \\ &\leftarrow \quad \{r_{tim}\} \end{aligned}$$

onto the stack. This step is illustrated in Figure 5f. The final step is to pop the final result set H_2 off the top of the stack. As expected, Tim satisfies the query; Sue is not in the final result set as she lives in the UK.

The algorithm. We give the pseudo code of the bottom-up approach in Algorithms 3 and 4. The algorithm starts with a call to Bottom-up-containment (Algorithm 3) with a query q and an inverted file encoded database S_{IF}.

Algorithm 3 Bottom-up-containment
input: query q, inverted file S_{IF}
output: $q \bowtie_{\subseteq} S$

```
1: s ← a new empty stack
2: Bottom-up-interior(root(q), s, S_IF)
3: return pop(s)
```

After setting up the (global) stack, processing proceeds with a call to Bottom-up-interior (Algorithm 4) on the root node of the query, $root(q)$. Lines 1-4 in Algorithm 4 correspond to Figures 5a-5c, namely, we recur on children until a flat set is encountered. Lines 5-9 collect from the top of the stack the set $Lists$ which must be covered by candidates for the current query node n. If each element of $Lists$ is non-empty (Line 10), then it is potentially possible to cover the subtree rooted at n in the database; otherwise, it is not possible and we push an empty set onto the stack to signal this to n's parent (Lines 14-15). In the case that it is possible, we compute the local set of candidates to cover n (Line 11), collect those candidate nodes which actually do cover the subtree rooted at n (Line 12), and push the set of these "head" nodes onto the stack for processing by the parent of n (Line 13). The process of constructing this set is denoted by $\mathcal{H}()$ in Figures 5d-5f. The bottom-up algorithm terminates when we have worked our way back to the root node, leaving the final result set on top of the stack, which is finally retrieved in Algorithm 3, Line 3.

Analysis. Correctness of the bottom-up algorithm is straightforward. To calculate the running time, we use the same notation as in our analysis of the top-down algorithm in Section

Algorithm 4 Bottom-up-interior
input: query node n, stack s, inverted file S_{IF}

```
1: push($, s)
2: for all c ∈ nodes(n) do
3:    Bottom-up-interior(c, s, S_IF)
4: end for
5: Lists ← ∅
6: while peek(s) ≠ $ do
7:    Lists ← Lists ∪ {pop(s)}
8: end while
9: pop(s)
10: if ∀L ∈ Lists : L ≠ ∅ then
11:    Candidates ← ⋂_{ℓ∈ℓ(n)} S_IF(ℓ)
12:    Heads ← {h | ∃C : [(h, C) ∈ Candidates
                 ∧∀L ∈ Lists : C ∩ L ≠ ∅]}
13:    push(Heads, s)
14: else
15:    push(∅, s)
16: end if
```

3.1. The work of Bottom-up-containment (Algorithm 3) takes constant time. The cost breakdown of Bottom-up-interior (Algorithm 4) is as follows.

- lines 1-10, 13-16: over the course of the whole computation, each internal node is processed once on each of these lines, all of constant cost, and hence is $\mathcal{O}(\mathcal{N}_q)$.
- line 11: Over the course of the full run of the algorithm, we evaluate this intersection once for each leaf of the query, with worst case cost of $\mathcal{O}(\mathcal{L}_q * |S|)$.
- line 12: Since all lists are sorted, this operation, over the course of the full computation, is $\mathcal{O}(\mathcal{N}_q * |S|)$.

We conclude that the worst-case running time of the bottom-up algorithm is $\mathcal{O}(|q| * |S|)$, i.e., the same as for the top-down algorithm.

3.3 Optimizations

There are many opportunities for optimizations to accelerate processing in our algorithms. In this section we discuss two simple optimizations which we will empirically study in Section 5. The first optimization is to cache a frequent subset of the inverted file payloads and the second optimization is to prune results in an early stage of join evaluation.

Caching. The inverted file S_{IF} is accessed continuously in the top-down and the bottom-up approach, i.e., for every occurrence of every leaf value ℓ in the query, we retrieve $S_{IF}(\ell)$. Retrieving the inverted list for every leaf value is fine for small queries, but for larger queries or a batch of queries, accessing S_{IF} can incur many repetitive accesses. We can reduce access cost by caching the inverted lists of the most frequent values of S in a main memory data structure, subject to an available memory budget; we study such caching below in our empirical study. We could also consider caching with respect to frequent values in a query workload; we leave this option open for future study.

Bloom filters. Bloom filters are compact data structures for probabilistic representation of a set that supports membership queries ('is element x in set Y?')[2]. Bloom filters

are applicable for flat sets, however they are not originally designed to represent hierarchies [21]. In [21], two data structures are presented, Breadth and Depth Bloom filters, which are multi-level structures that assist efficient processing of queries on XML trees. Hierarchical Bloom filters are well suited for assisting in the evaluation of set containment queries on nested sets, i.e., we can build a Bloom filter on a subset of the leaf values in a tree (say, the least frequent), place the filter at the root of the tree and do a bitwise comparison between the filters of two trees before descending into their internal structure. If the comparison fails, we know that a containment is not possible. We refer the reader to the the full version of this paper for further discussion [17].

4. EXTENSIONS

In this section we show how our top-down and bottom-up algorithms can be adapted to (1) process other set-based join types and (2) accommodate alternate embedding semantics.

4.1 Extension 1: other set-based joins

The top-down and bottom-up algorithms presented in the previous sections can be directly adapted to process several other natural set-based join types. In evaluating $q \bowtie_{\subseteq} S$, we ensured in both algorithms during evaluation of the intersection operation (i.e., line 8 of Algorithm 2 and line 11 of Algorithm 4), for each internal node n of q and for each potential match $s \in S$ for n, that s has at least as many leaf children as n does, i.e., $|\ell(n)| \leq |\ell(s)|$. Our adjustments will be to this condition and/or the construction of the set of candidate matches of n in S.

Set equality join. For the set equality join, defined as

$$q \bowtie_{=} S \quad = \quad \{(q, s) \mid s \in S \text{ and } q = s\},$$

the leaf count condition is strengthened in both algorithms, such that, when computing for internal query node n,

$$Candidates \leftarrow \bigcap_{\ell \in \ell(n)} S_{IF}(\ell),$$

we remove from $Candidates$ those sets s where $|\ell(n)| \neq |\ell(s)|$. For the implementation, we can just extend the inverted lists to also carry the leaf count of each internal node.

Superset join. For the superset join, defined as

$$q \bowtie_{\supseteq} S \quad = \quad \{(q, s) \mid s \in S \text{ and } q \supseteq s\},$$

a more relaxed nature of processing is required. In particular, we compute the set of candidates for internal query node n as

$$Candidates \quad \leftarrow \quad \biguplus_{\ell \in \ell(n)} S_{IF}(\ell),$$

where $\biguplus$ is the multi-set union, and remove from $Candidates$ all copies of those sets s where $|\ell(s)|$ does not equal the number of occurrences of s in $Candidates$. This ensures that s does not have leaf values outside of those of n. Note that multi-set semantics is used to permit the duplicates necessary for checking this condition.

ε-overlap join. The subset join retrieves all sets that have, at all nesting levels, at least as many leaves as the query. Often, we are interested in the weaker condition of having just some non-empty overlap with the query. This is captured by the ε-overlap join [24], for some $\varepsilon \geq 1$, defined as

$$q \bowtie_{\varepsilon} S = \Big\{(q, s) \;\Big|\; s \in S \wedge \exists \varphi_s^q \bigwedge_{(n,m) \in \varphi_s^q} |\ell(n) \cap \ell(m)| \geq \varepsilon\Big\},$$

where φ_s^q is an embedding of q in s (i.e., a set of pairs of all the corresponding internal nodes of q and s). We compute the set of candidates for internal query node n as

$$Candidates \quad \leftarrow \quad \biguplus_{\ell \in \ell(n)} S_{IF}(\ell),$$

and remove from $Candidates$ all copies of those sets s which do not appear at least ε times in $Candidates$.

4.2 Extension 2: other embeddings

The algorithms presented in Section 3 can also be adapted to evaluate the iso- and homeomorphic containment semantics discussed in Section 2. We illustrate this on the top-down approach (Section 3.1).

Isomorphic containment. To accommodate the special restricted case of computing $\subseteq_{\text{iso}}$ embeddings of the query q in collection S, an extra condition is added in the top-down algorithm, namely, at every level of processing q, the matches of internal nodes of q to internal nodes of s must be injective. We then must do some bookkeeping at the beginning of each recursion, to ensure that the nodes of the query and the nodes of the data set do indeed match in a one-to-one manner. First, we have to mark retrieved lists of the inverted file to ensure that they are not reused. Then, we have to potentially backtrack, updating marked lists, to check additional subtrees in the event of a non-match (i.e., returning unsuccessfully from a recursive call) to ensure that all possibilities for a correct match are investigated. While this bookkeeping is a straightforward addition to the algorithm, backtracking necessarily incurs a higher runtime complexity.

Homeomorphic containment. The top-down algorithm is adapted to check for relaxed $\subseteq_{\text{homeo}}$ embeddings in the collection S by simply tagging all nodes in the inverted lists with a scheme that encodes ancestor-descendant relationships (e.g., pre-post ordering [11]). Next, the join condition in the algorithm is updated to check ancestor-descendant containment (i.e., line 9 of Algorithm 2). This adjustment does not introduce any additional complexity since we can determine the ancestor-descendant relationship between any two nodes in constant time by using only two comparison operations [11].

5. EMPIRICAL ANALYSIS

In this section we report on a series of experiments which we performed to determine the general empirical behavior of our proposed algorithms. Our approach here was to measure the efficiency and scalability of the top-down and bottom-up algorithms on a variety of data sets and queries.

5.1 Experimental set-up

Environment. The experiments were conducted on a machine running Fedora 12 / 64-bit Linux, with 8 x 2 Ghz processors, 144 GB RAM, and a 2.8 TB disk. The data structures and algorithms were implemented in Java (version

J2SE-1.5), as a single threaded process. The inverted files were implemented using Tokyo Cabinet[6] (version 1.24) as the storage engine. Tokyo Cabinet is an industrial-strength open-source library of routines for managing key-value pair indexes on disk, as B+trees or hash tables. We used the external memory hash table in our implementation, with no main memory buffering (i.e., we explicitly disabled Tokyo Cabinet's caching mechanisms).

Data sets. For database instances S (recall Equation 2), we used four different synthetic data sets and two different real-world data sets in our experiments.

- For all synthetic data sets, we drew leaf values from a fixed domain of 10,000,000 labels. We generated two different classes of synthetic data sets. The first class was generated such that data objects had their leaf values drawn at random, uniformly distributed across the set of labels. The second class was generated such that data objects exhibited a skewed Zipfian distribution of leaf values, across all sets in the database [12]. Skew is determined by a factor $0 < \theta < 1$, where the closer θ is to 1, the greater the skew. The skewness values we used are $\theta \in \{0.5, 0.7, 0.9\}$ (we only report on a representative subset of the results for these data sets here; see [17] for all results). Within both the uniform and skewed classes, we generated synthetic databases with either "wide" or "deep" nested sets. The process to generate a nested set was as follows: starting at the root, (1) randomly choose a number of leaf nodes for the current node; (2) after assigning labels to the leaf children of the current node, stop extending this node with some probability; (3) if we do not stop, then randomly choose some number of internal children, and recur on each of them, starting at step (1). The values we used for the stopping probability and the upper bounds on the number of leaf and internal children for "wide" and "deep" data sets are given in Table 3.

  To summarize, we generated uniform-wide, uniform-deep, skewed-wide, and skewed-deep synthetic data collections.

- The first real data set is a collection of tweets from Twitter, in nested JSON format (which we directly mapped into our data model), and contains nested information, such as messages, dates, user ids and urls, about tweets on a pop idol ('Justin Bieber'). This data set, collected via the Twitter Search API,[7] is also used and described further in [30].

- The second real data set is a collection of articles retrieved from the DBLP Computer Science Bibliography,[8] as an XML database. This database contains records regarding computer science articles, which we mapped directly into nested sets in our model.

Queries. For benchmark queries, we arbitrarily selected 100 nested sets from each data collection S. We distorted half of the selected queries such that they are not contained in the data collection (i.e., we have 50 positive queries and 50 negative queries for each S); this was done by adding a new leaf value to each set which does not appear anywhere else in the database.

[6]`http://fallabs.com/tokyocabinet/`
[7]`http://dev.twitter.com/docs/api/1/get/search`
[8]`http://www.informatik.uni-trier.de/~ley/db/`

Table 3: Parameters used for generating different types of synthetic nested sets.

parameter	**wide sets**	**deep sets**
max # of leaves per node	12	2
max # of non-leaves per node	6	3
stopping probability	0.8	0.2

Inverted list caching. Each benchmark was run both with and without the caching optimization described in Section 3.3. The caching parameter, i.e., the number of lists that are buffered in main memory, was set to 250 in all cases.

Other assumptions. We made two simplifying, but easily remediable, assumptions in our empirical study. (1) The payloads for any given internal query node, i.e., the retrieved inverted lists, fit in main memory. As these lists are flat, the I/O-efficient blocked approach of Mamoulis for flat sets [24] could be easily used, if necessary, to lift this assumption. (2) The stack used in the bottom-up algorithm fits in main memory. I/O-efficient solutions for stacks, e.g., as available in the open-source STXXL library [6], can be used off-the-shelf if necessary to remove this assumption.

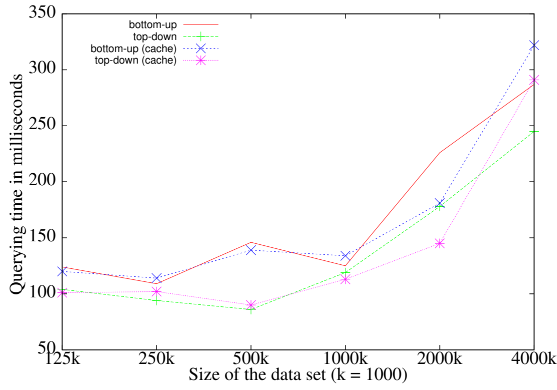
5.2 Experiments

We next discuss the series of experiments we performed. Unless stated otherwise, the unit of performance measurement in our experiments is the elapsed time of sequentially executing all 100 benchmark queries. For each measurement, we repeat this ten times, exclude the minimum and maximum timings, and report the average of the middle eight executions.

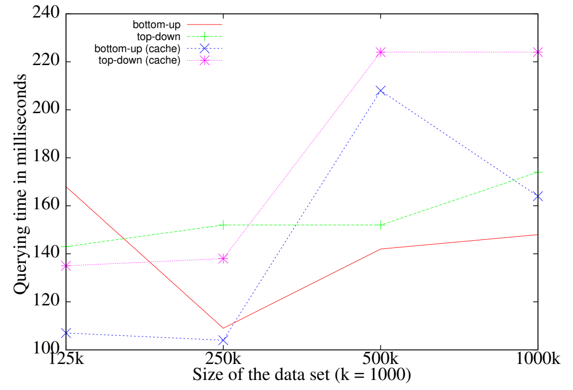
Experiment 1. We measured the cost of query execution on the uniform synthetic databases, while increasing the database size. Figures 6a and 6b show the results of this experiment. The figures show that an increase in size does not always imply an increase in querying time. Also, caching shows no real effect on the uniform data sets, as is to be expected. Overall, we see that individual queries can be processed in milliseconds, even on databases having 4 million nested sets.

Experiment 2. We measured the cost of query execution on the skewed synthetic databases, while increasing the database size. Figures 6c and 6d show the results of this experiment. These figures show that there is a modest improvement in query costs when using caching, on both the wide and deep databases. Compared to uniform data, we see that skewness has a considerable impact on evaluation costs. Overall, we see that individual queries can be processed in milliseconds to tens of milliseconds, even on databases having 4 million nested sets.

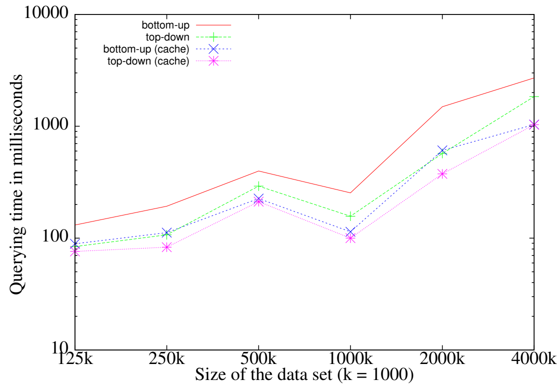
A technical observation we can make at this point is that Figures 6a-6d show noise/artifacts, i.e., small dips in querying time. The dips only occur in those situations where individual querying times are on the order of milliseconds.



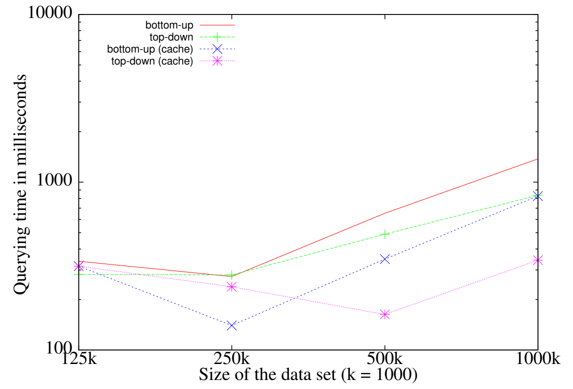
(a) Average query times on uniform wide synthetic databases ranging in size from 125K to 4M nested sets (Experiment 1).



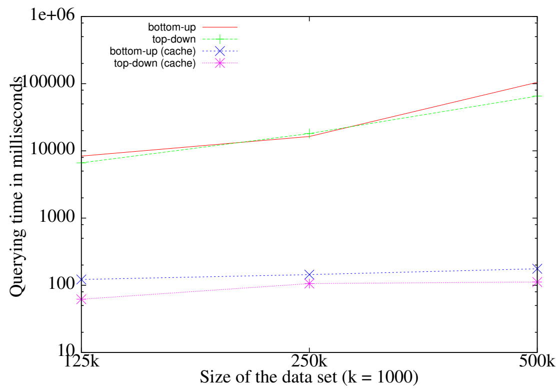
(b) Average query times on uniform deep synthetic databases ranging in size from 125K to 1M nested sets (Experiment 1).



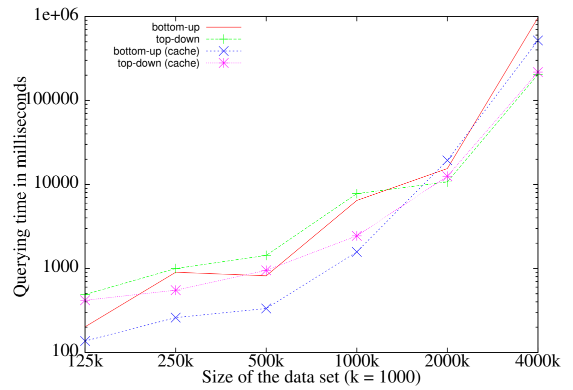
(c) Average query times on skewed ($\theta = 0.7$) wide synthetic databases ranging in size from 125K to 4M nested sets (Experiment 2).



(d) Average query times on skewed ($\theta = 0.7$) deep synthetic databases ranging in size from 125K to 1M nested sets (Experiment 2).



(e) Average query time on the Twitter data collection, ranging in size from 125K to 500K nested sets. (Experiment 3).



(f) Average query time on the DBLP data collection, ranging in size from 125K to 4M nested sets (Experiment 3).

Figure 6: Average cost (in milliseconds) for processing 100 queries on the (a-b) uniform synthetic, (c-d) skewed synthetic, and (e-f) real data sets (Experiments 1, 2, and 3).

These small fluctuations are due to the selection of queries. In particular, when looking at the queries selected, we saw that the small databases had some larger queries selected than those selected for subsequently larger databases.

Experiment 3. We measured the cost of query execution on the real databases, while increasing the database size. Upon inspection, we found that the distributions of values in both data sets were skewed. For example, popular users dominate the Twitter discussion of the pop idol.

Figure 6e shows the results of the Twitter data. The figure shows an improvement by a factor of 100 with caching.

Table 4: Average cost, in milliseconds, for individual negative ($-$) and positive ($+$) queries on databases of 500K nested sets (Experiment 4). The table shows a comparison between the top-down (TD) and bottom-up (BU) algorithms, with inverted list caching disabled and enabled. Also given are "speed-up", the ratio of no caching to caching, and $\frac{\text{TD}}{\text{BU}}$, the ratio of top-down to bottom-up.

		Uniform data						**Skewed data ($\theta = 0.5$)**						**Real data**					
		wide sets			*deep sets*			*wide sets*			*deep sets*			*Twitter*			*DBLP*		
		TD	BU	$\frac{\text{TD}}{\text{BU}}$	TD	BU	$\frac{\text{TD}}{\text{BU}}$	TD	BU	$\frac{\text{TD}}{\text{BU}}$	TD	BU	$\frac{\text{TD}}{\text{BU}}$	TD	BU	$\frac{\text{TD}}{\text{BU}}$	TD	BU	$\frac{\text{TD}}{\text{BU}}$
no caching	+	1.18	1.02	1.2	1.39	0.83	1.7	1.9	1.8	1.1	5.58	4.04	1.4	64.1	74.11	0.9	4.42	2.51	1.8
	−	0.4	1.15	0.4	0.94	1.53	0.6	0.1	4.18	0.02	1.18	7.71	0.2	9.76	14.64	0.7	0.62	0.79	0.8
caching	+	1.46	0.75	1.95	1.76	0.92	1.9	0.53	0.84	0.6	2.64	2.28	1.2	0.79	0.69	1.1	1.94	0.98	2.0
	−	0.56	0.83	0.7	1.15	1.69	0.7	0.07	3.09	0.02	0.67	4.96	0.1	0.1	0.88	0.1	0.4	0.36	1.1
speed-up	+	0.8	1.36	0.6	0.78	0.9	0.9	3.6	2.14	1.7	2.11	1.78	1.2	81.1	107.4	0.8	2.28	2.56	0.9
	−	0.71	1.39	0.5	0.82	0.91	0.9	1.4	1.35	1.04	1.76	1.55	1.1	97.6	16.64	5.9	1.55	2.19	0.7

Overall, we see that individual queries can be processed in hundreds of ms without caching, and on the order of milliseconds or less with caching. Figure 6f shows the results of the DBLP data set. Here, we see a less pronounced improvement with caching enabled, and behavior similar to those of Twitter and the other skewed data collections.

Experiment 4. In this experiment we distinguished costs for positive and negative queries. Table 4 shows the average querying time for individual positive and negative queries, for the databases having 500K nested sets. The main observation we make here is that, both with and without caching, both algorithms exhibit very similar behavior. We sometimes see a slight performance gain of the top-down approach for negative queries and the bottom-up approach for positive queries, but the differences are minor. We leave open a finer empirical analysis of top-down vs. bottom-up as an interesting topic for further research.

Experiment 5. Finally, we conducted an experiment with the Bloom filter, using the top-down algorithm, to investigate its effectiveness in early pruning of false matches of the query in the database. Here, we used uniform synthetic data with an average of seven levels of nesting. We adorn the root of every query and database object with a Bloom filter, in the form of a boolean array of 20 bits. Figure 7 shows that the Bloom filter is indeed effective in spotting false matches, leading to an order of magnitude reduction in query cost.

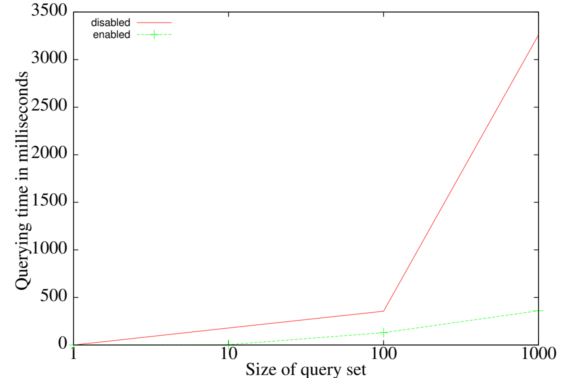


Figure 7: The average cumulative querying time when using Bloom filters with the top-down algorithm (Experiment 5). The data set contains 100K uniform synthetic deeply nested sets (average of seven levels of nesting). Every query is distorted (i.e., is a negative query).

5.3 Discussion

In general, we observed that both the top-down and bottom up algorithms exhibited efficient and scalable behavior on all databases. In particular, queries could often be processed in milliseconds or tens of milliseconds, even on collections of millions of nested sets. Both algorithms were most challenged by skewed data sets. We also demonstrated the effectiveness of caching (for non-uniform data) for both algorithms, and of filtering for early pruning of candidates in the top-down algorithm, with both optimizations exhibiting up to orders of magnitude improvement in query costs.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have developed the first known efficient scalable solutions for computing containment joins on collections of nested data sets. We gave two distinct algorithms for this basic problem, discussed optimizations to accelerate query processing with the algorithms, and presented extensions to our solutions to handle other practical set-based query patterns and alternate containment semantics. Analytic and empirical analysis demonstrated the efficiency and scalability of our solutions. Our algorithms have the additional benefits of conceptual simplicity and usage of a widely adopted data structure. Hence, our results have excellent potential to make a practical impact on data management systems and solutions.

Many interesting research challenges remain open. We close by listing a few. (1) Our empirical study showed that skewed data is challenging for our algorithms. Incorporation in our algorithms of recent results on efficiently dealing with list intersections and data skew should be investigated [5, 35]. (2) It would be interesting to study the impact on our solutions of variations to the data model (e.g., multi-set and list types) or query model (e.g., bottom-up subtree embeddings [36]). (3) Further theoretical analysis of set-based joins in the context of nested data is warranted, to place them in the broader landscape of join problems; cf. [3, 22]. (4) In practice, both queries and data are often noisy and uncertain. It would be interesting to investigate extensions to handle query relaxations such as set similarity joins and set containment in uncertain data models [1, 23, 31, 39]. (5) More sophisticated pruning mechanisms for hierarchical data could be profitably developed for our context, e.g., based on recent work such as [16, 33]. (6) A deeper study of nested set caching mechanisms should be undertaken, e.g.,

investigating alternatives such as caching with respect to an evolving query workload.

Acknowledgments. We thank Paul De Bra, Toon Calders, Alexander Serebrenik, and the reviewers for their many helpful comments.

7. REFERENCES

[1] N. Augsten, M. Bohlen, C. Dyreson, and J. Gamper. Approximate joins for data-centric XML. In *Proc. IEEE ICDE*, pages 814–823, 2008.

[2] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1:636–646, 2002.

[3] J.-Y. Cai et al. On the complexity of join predicates. In *Proc. ACM PODS*, pages 207–214, Santa Barbara, CA, USA, 2001.

[4] B. Cao and A. Badia. SQL query optimization through nested relational algebra. *ACM Trans. Database Syst.*, 32(3):18:1–18:46, 2007.

[5] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.*, 29:1–25, 2010.

[6] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Softw. Pract. Exper.*, 38(6):589–637, 2008.

[7] G. H. L. Fletcher, J. Van den Bussche, D. Van Gucht, and S. Vansummeren. Towards a theory of search queries. *ACM Trans. Database Syst.*, 35(4):28, 2010.

[8] M. Fontoura et al. Optimizing cursor movement in holistic twig joins. In *Proc. ACM CIKM*, pages 784–791, Bremen, Germany, 2005.

[9] G. Garani and R. Johnson. Joining nested relations and subrelations. *Inf. Syst.*, 25(4):287–307, 2000.

[10] M. Götz, C. Koch, and W. Martens. Efficient algorithms for descendant-only tree pattern queries. *Inf. Syst.*, 34(7):602–623, 2009.

[11] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(10):1381–1403, 2007.

[12] J. Gray et al. Quickly generating billion-record synthetic databases. In *Proc. ACM SIGMOD*, pages 243–252, Minneapolis, MN, USA, 1994.

[13] C. Gutierrez et al. Foundations of semantic web databases. *J. Comput. Syst. Sci.*, 77(3):520–541, 2011.

[14] M. Hachicha and J. Darmont. A survey of XML tree patterns. *IEEE Trans. Knowl. Data Eng.*, 25(1):29–46, 2013.

[15] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *VLDB J.*, 12(3):244–261, 2003.

[16] S. Helmer et al. Measuring structural similarity of semistructured data based on information-theoretic approaches. *VLDB J.*, 21(5):677–702, 2012.

[17] A. Ibrahim. Containment queries on nested sets. MSc Thesis, Eindhoven University of Technology, 2012.

[18] Y. Kaneta, H. Arimura, and R. Raman. Faster bit-parallel algorithms for unordered pseudo-tree matching and tree homeomorphism. *J. Discrete Algorithms*, 14:119–135, 2012.

[19] R. Kaushik et al. On the integration of structure indexes and inverted lists. In *Proc. ACM SIGMOD*, pages 779–790, Paris, 2004.

[20] P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, University of Helsinki, 1992.

[21] G. Koloniari and E. Pitoura. Filters for XML-based service discovery in pervasive computing. *Computer J.*, 47(4):461–474, 2004.

[22] D. Leinders and J. Van den Bussche. On the complexity of division and set joins in the relational algebra. *J. Comput. Syst. Sci.*, 73(4):538 – 549, 2007.

[23] X. Lian and L. Chen. Set similarity join on probabilistic data. *PVLDB*, 3(1):650–659, 2010.

[24] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proc. ACM SIGMOD*, pages 157–168, San Diego, CA, 2003.

[25] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. Database Syst.*, 28(1):56–99, 2003.

[26] S. Melnik et al. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[27] C. Olston et al. Pig latin: a not-so-foreign language for data processing. In *Proc. ACM SIGMOD*, pages 1099–1110, Vancouver, Canada, 2008.

[28] Ramasamy et al. Set containment joins: The good, the bad and the ugly. In *Proc. VLDB*, pages 351–362, Cairo, Egypt, 2000.

[29] R. Rantzau. Processing frequent itemset discovery queries by division and set containment join operators. In *Proc. ACM DMKD*, pages 20–27, San Diego, CA, USA, 2003.

[30] J. Roijackers. Bridging SQL and NoSQL. MSc Thesis, Eindhoven University of Technology, 2012.

[31] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proc. ACM SIGMOD*, pages 743–754, Paris, France, 2004.

[32] R. Shamir and D. Tsur. Faster subtree isomorphism. *J. of Algorithms*, 33:267–280, 1999.

[33] S. Tatikonda and S. Parthasarathy. Hashing tree-structured data: Methods and applications. In *Proc. IEEE ICDE*, pages 429–440, Long Beach, CA, USA, 2010.

[34] M. Terrovitis et al. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *Proc. ACM CIKM*, pages 728–737, Arlington, VA, USA, 2006.

[35] M. Terrovitis et al. Efficient answering of set containment queries for skewed item distributions. In *Proc. EDBT*, pages 225–236, Uppsala, Sweden, 2011.

[36] G. Valiente. *Algorithms on trees and graphs*. Springer, Berlin, 2002.

[37] I. H. Witten et al. *Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd ed.* Morgan Kaufmann Publishers, San Francisco, 1999.

[38] C. Zhang et al. On supporting containment queries in relational database management systems. In *Proc. ACM SIGMOD*, pages 425–436, Santa Barbara, CA, USA, 2001.

[39] X. Zhang et al. Efficient processing of probabilistic set-containment queries on uncertain set-valued data. *Inf. Sci.*, 196:97–117, 2012.