Efficient processing of containment queries on nested sets

Ahmed Ibrahim and George Fletcher

Eindhoven University of Technology
The Netherlands

EDBT 2013
Genoa, Italy

Nested sets

Nested structure in data is natural in a wide variety of applications, for example

- scientific workflows
- XML and complex object data management
- business process management
- JSON data management
  - NoSQL key-value stores
  - web analytics

Nested sets

Nested structure in data is natural in a wide variety of applications, for example

- scientific workflows
- XML and complex object data management
- business process management
- JSON data management
  - NoSQL key-value stores
  - web analytics

Nested sets are an abstraction of the basic hierarchical structure of the data occurring in these many applications

Nested sets

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

Example nested set collection S: where people live, where they hold driving privileges, and, for each locale, the set of licenses and vehicle types for which they are authorized

Set containment queries

A basic problem. Given a set q and a collection of sets S, find all sets of S containing q:

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

Set containment queries

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

Example on our collection S:

$$q = \{\texttt{USA}, \{\texttt{UK}, \{\texttt{A}, \texttt{motorbike}\}\}\}$$

i.e., "people that live in the USA who have license type A valid for a motorbike in the UK"

Set containment queries

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

Example on our collection S:

$$q = \{\texttt{USA}, \{\texttt{UK}, \{\texttt{A}, \texttt{motorbike}\}\}\}$$

i.e., "people that live in the USA who have license type A valid for a motorbike in the UK"
Then,

$$q \bowtie_{\subseteq} S.value \quad = \quad \{(q, Tim)\}$$

where Tim is the record having key $\texttt{Tim}$.

Set containment queries

Set containment is a fundamental query pattern, which finds a variety of applications, e.g., in data mining solutions.

Consequently, efficient processing of set containment queries has been heavily investigated since the 90's.

However, to our knowledge, all known solutions are engineered for **flat** sets, that is, sets of atomic objects without nesting.

Set containment queries

Set containment is a fundamental query pattern, which finds a variety of applications, e.g., in data mining solutions.

Consequently, efficient processing of set containment queries has been heavily investigated since the 90's.

However, to our knowledge, all known solutions are engineered for **flat** sets, that is, sets of atomic objects without nesting.

Also closely related are the heavily studied tree-pattern queries for XML and nested relational join queries.

- However, solutions for these problems are engineered for restricted **special cases** of nested sets.

Our contributions in this paper

(1) We highlight the general problem of **nested set containment** and propose two practical solutions:

- a **top down** algorithm, which starts processing at the outer-most nesting level of the query, and working inwards; and,
- a **bottom up** algorithm, which works depth-first, 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.

Our contributions in this paper

(2) We develop extensions to both algorithms, to handle

- related query types (such as superset and set-overlap); and,
- natural variations of the semantics of containment.

Our contributions in this paper

(2) We develop extensions to both algorithms, to handle

- related query types (such as superset and set-overlap); and,
- natural variations of the semantics of containment.

(3) We show that our solutions are efficient and scalable, on a variety of synthetic and real data sets.

Our contributions in this paper

(2) We develop extensions to both algorithms, to handle

- related query types (such as superset and set-overlap); and,
- natural variations of the semantics of containment.

(3) 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.

Preliminaries

Data model

We consider data objects in the form of finite sets built over some universe of atomic objects (e.g., strings or integers).

We assume no particular internal structure or nesting depth.

Preliminaries

Data model

We consider data objects in the form of finite sets built over some universe of atomic objects (e.g., strings or integers).

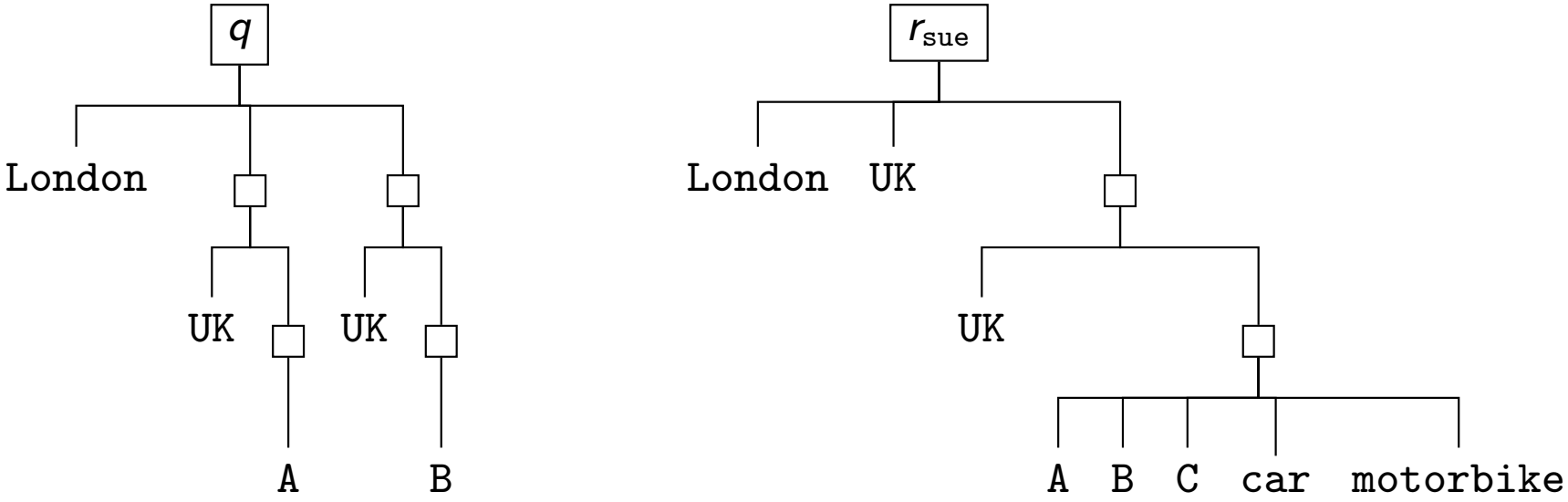
We assume no particular internal structure or nesting depth.

Containment

We interpret "$q \subseteq s$" as **homomorphic containment**.

That is, viewing nested sets as a type of unordered node-labeled rooted trees, we consider q to be contained in s if there is a homomorphism embedding q in s.

Preliminaries

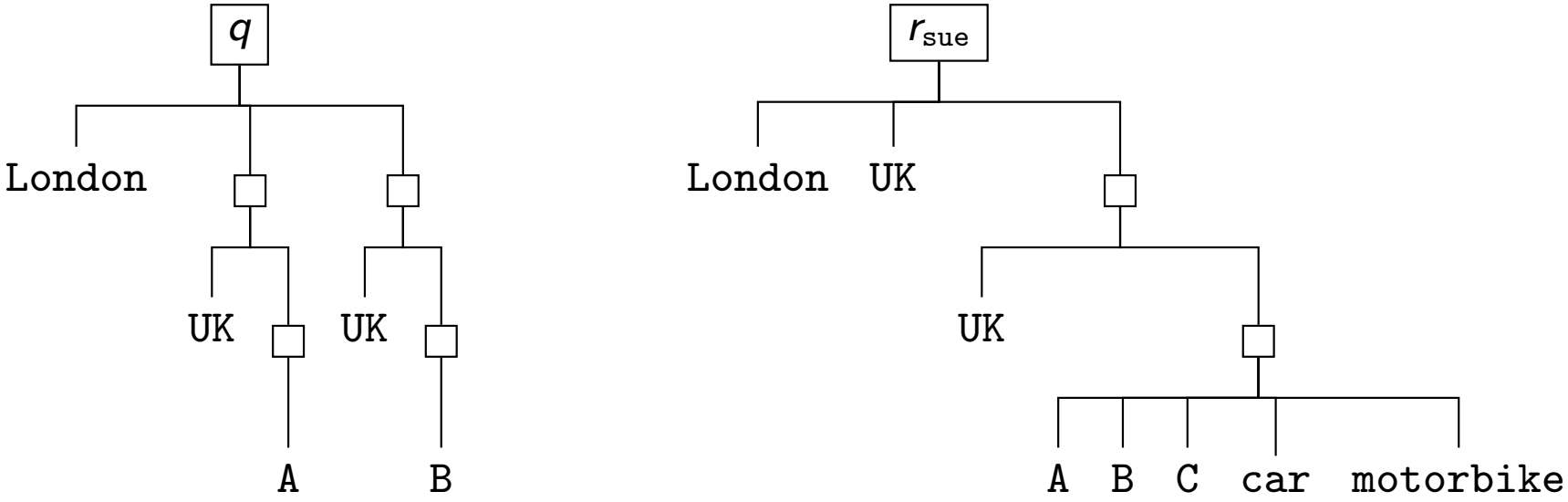


Example. Consider the sets

$$
\begin{aligned}
q &= \{\texttt{London}, \{\texttt{UK}, \{\texttt{A}\}\}, \{\texttt{UK}, \{\texttt{B}\}\}\} \\
r_{\texttt{sue}} &= \{\texttt{London}, \texttt{UK}, \{\texttt{UK}, \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{car}, \texttt{motorbike}\}\}\}
\end{aligned}
$$

visualized above on the left and right, resp.

Preliminaries

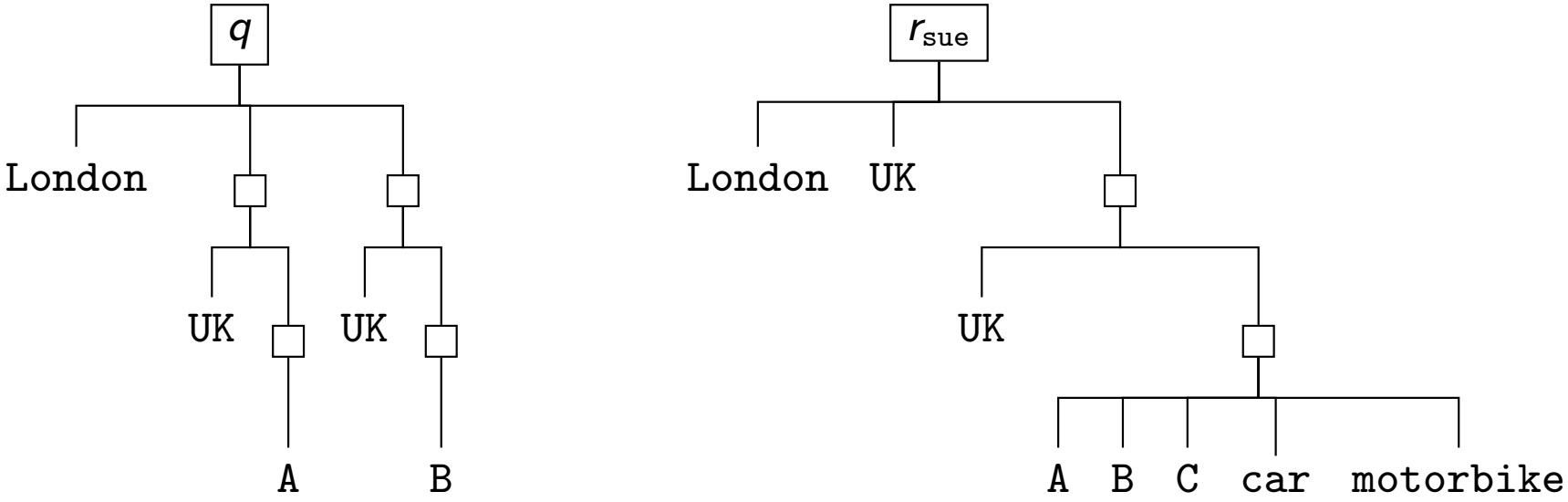


Example. Consider the sets

$$
\begin{aligned}
q &= \{\texttt{London}, \{\texttt{UK}, \{\texttt{A}\}\}, \{\texttt{UK}, \{\texttt{B}\}\}\} \\
r_{\texttt{sue}} &= \{\texttt{London}, \texttt{UK}, \{\texttt{UK}, \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{car}, \texttt{motorbike}\}\}\}
\end{aligned}
$$

visualized above on the left and right, resp.
Then q is homomorphically contained in $r_{\texttt{sue}}$.

Preliminaries



Example, cont. Note, however, that there is no injective embedding possible.

- That is, the left set is not "isomorphically" contained in the right set.

Preliminaries

Storage

We adopt the popular and widely available **inverted file data structure** for physical representation of collections of nested sets.

The inverted file maps each atom a in a collection S to an ordered list of all occurrences of a in S. Namely,

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

sorted on p_i values, where

- the p_i's are the (arbitrarily assigned) identifiers of all "locations" (i.e., sets) containing a in S; and,
- each C_i is the sorted listing of identifiers of all non-atomic elements of p_i.

Preliminaries

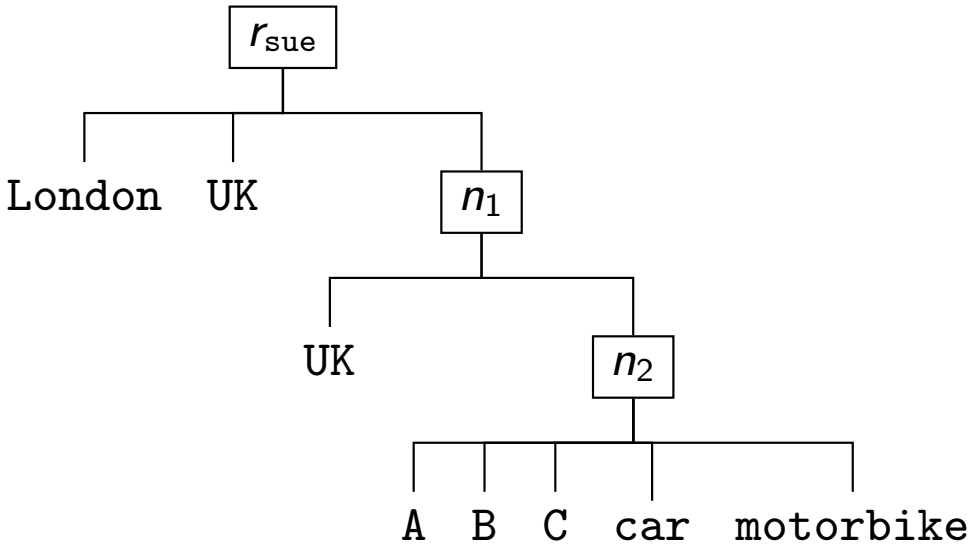
key	value
Sue	$\{\texttt{London}, \texttt{UK}, \{\texttt{UK}, \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{car}, \texttt{motorbike}\}\}\}$
Tim	$\{\texttt{Boston}, \texttt{USA}, \{\texttt{USA}, \texttt{VA}, \{\texttt{A}, \texttt{B}, \texttt{car}\}\}, \{\texttt{UK}, \{\texttt{A}, \texttt{motorbike}\}\}\}$

Example. Consider again our collection S ...

Preliminaries

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

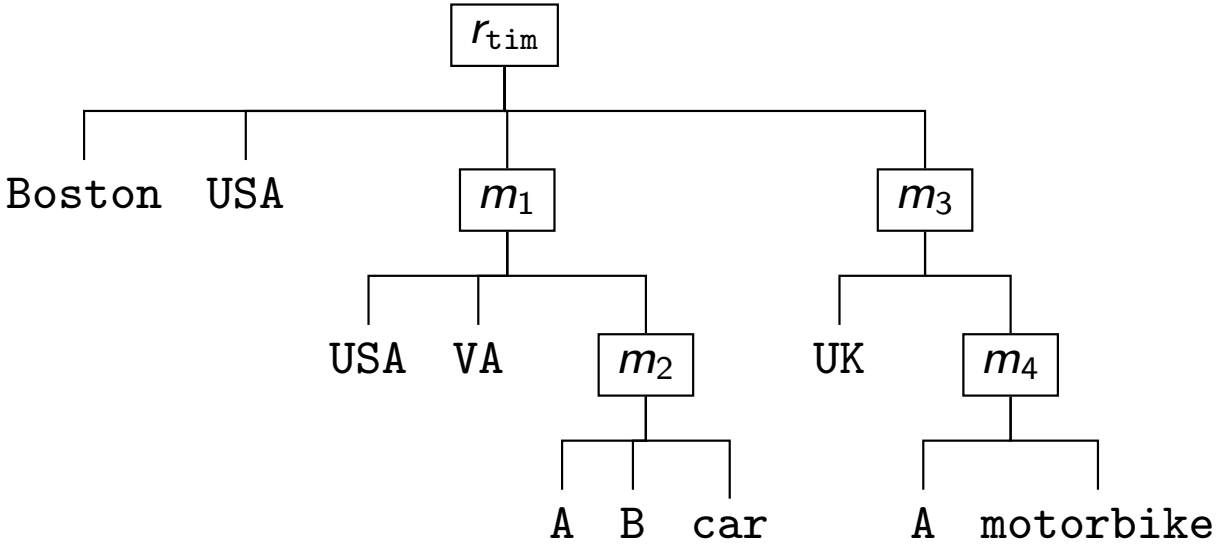
Example, cont. ... with the following assignment of identifiers for Sue ...



Preliminaries

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

Example, cont. ... and with the following for `Tim`.



Preliminaries

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

Example, cont. Then we have the following inverted file S_{IF} for S.

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	$\langle (m_1, \langle m_2 \rangle) \rangle$

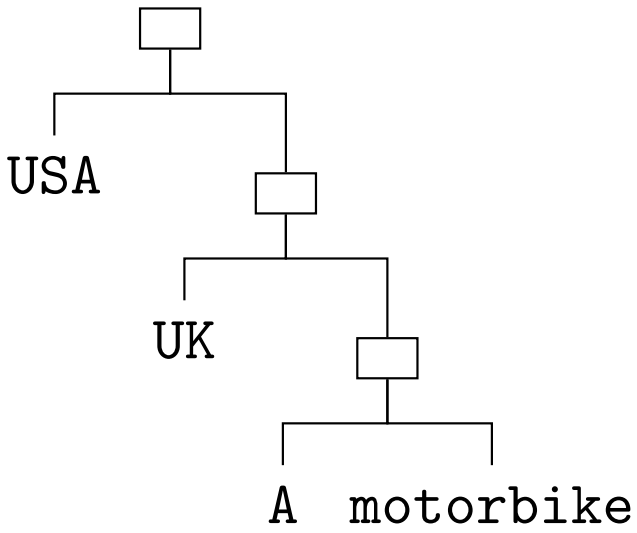
Efficiently processing containment queries

In the paper, we present two different solutions for computing $q \bowtie_{\subseteq} S$ under the homomorphic semantics for containment, using this inverted file data structure.

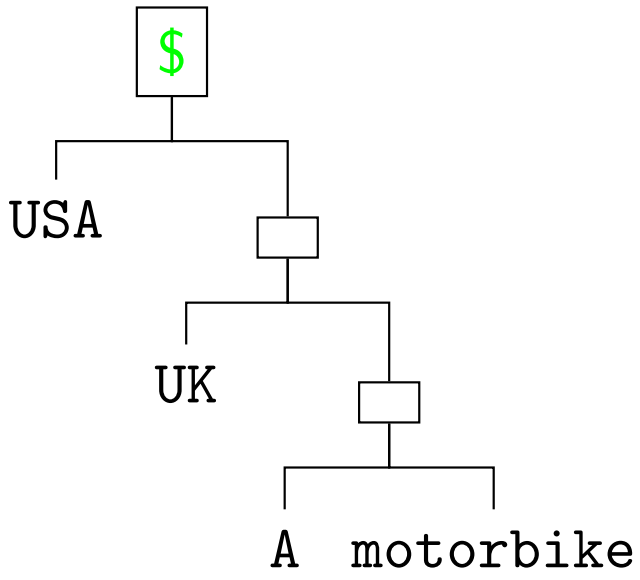
Efficiently processing containment queries

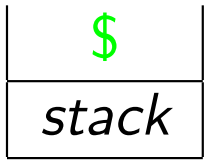
In the paper, we present two different solutions for computing $q \bowtie_{\subseteq} S$ under the homomorphic semantics for containment, using this inverted file data structure.

Let's just consider the **bottom-up** approach, starting with our example query q:



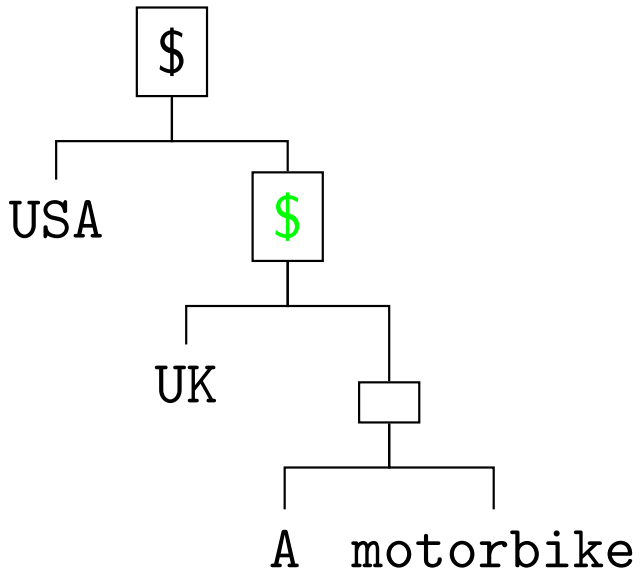
Example run of the bottom-up algorithm

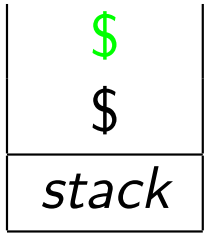




(a) We start at the root set, pushing a marker $ onto an empty stack.

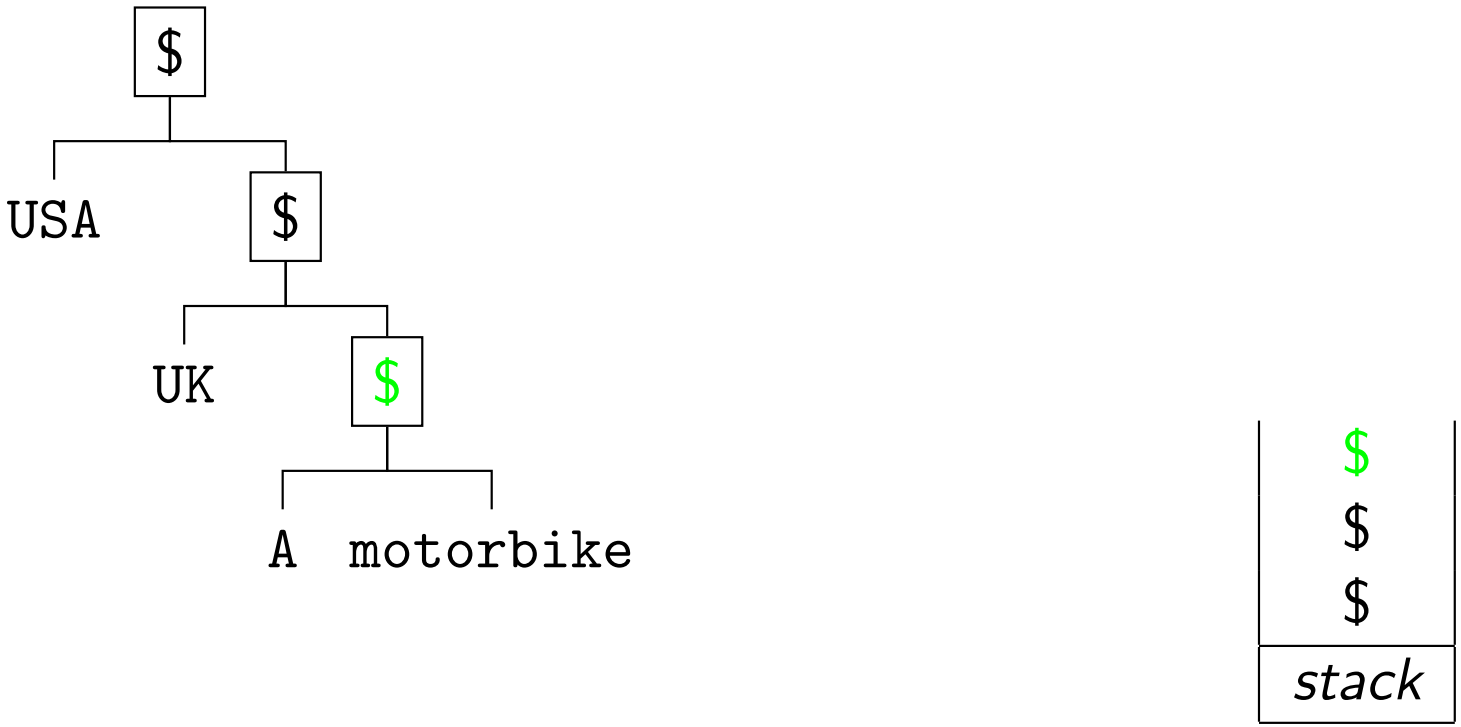
Example run of the bottom-up algorithm





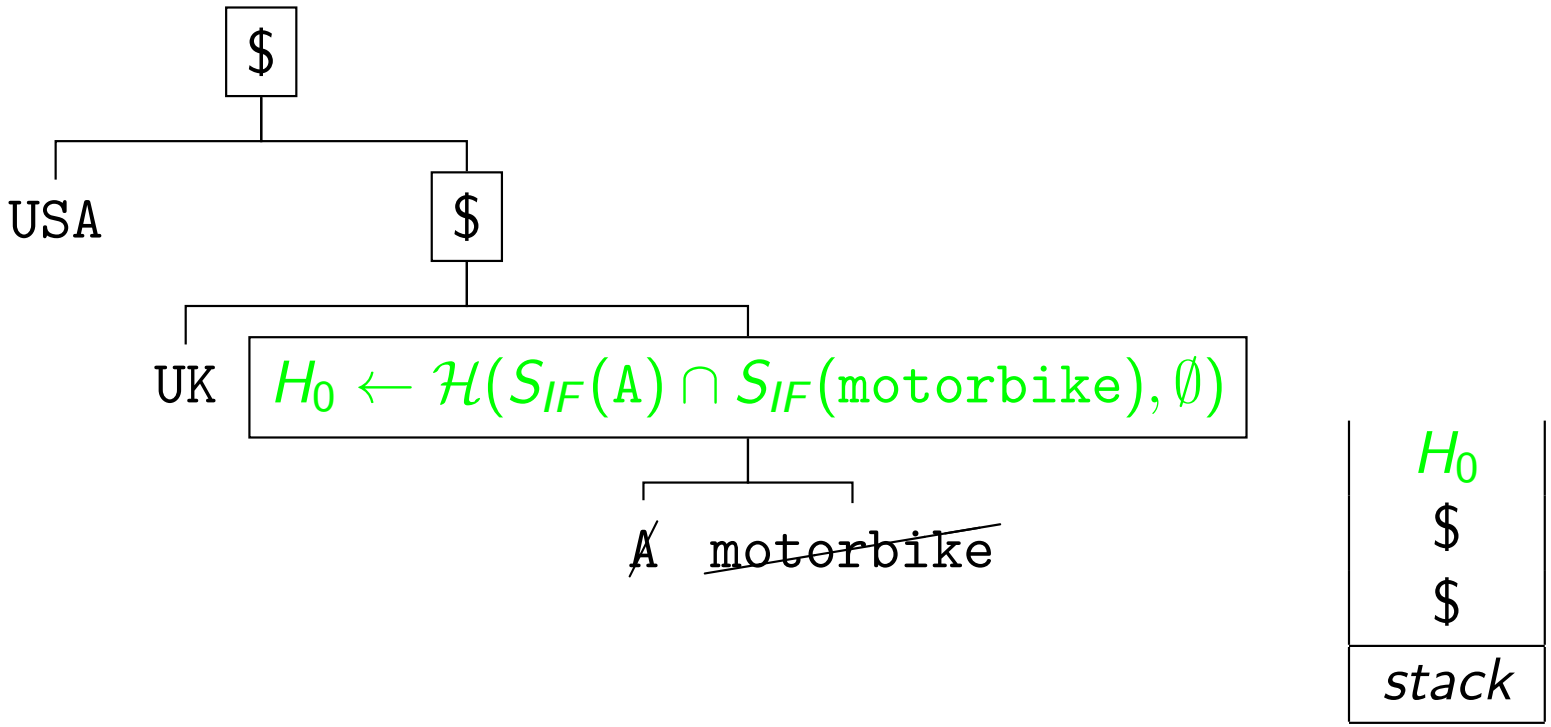
(b) We continue with the non-atomic child of the root, pushing a second marker onto the stack.

Example run of the bottom-up algorithm



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

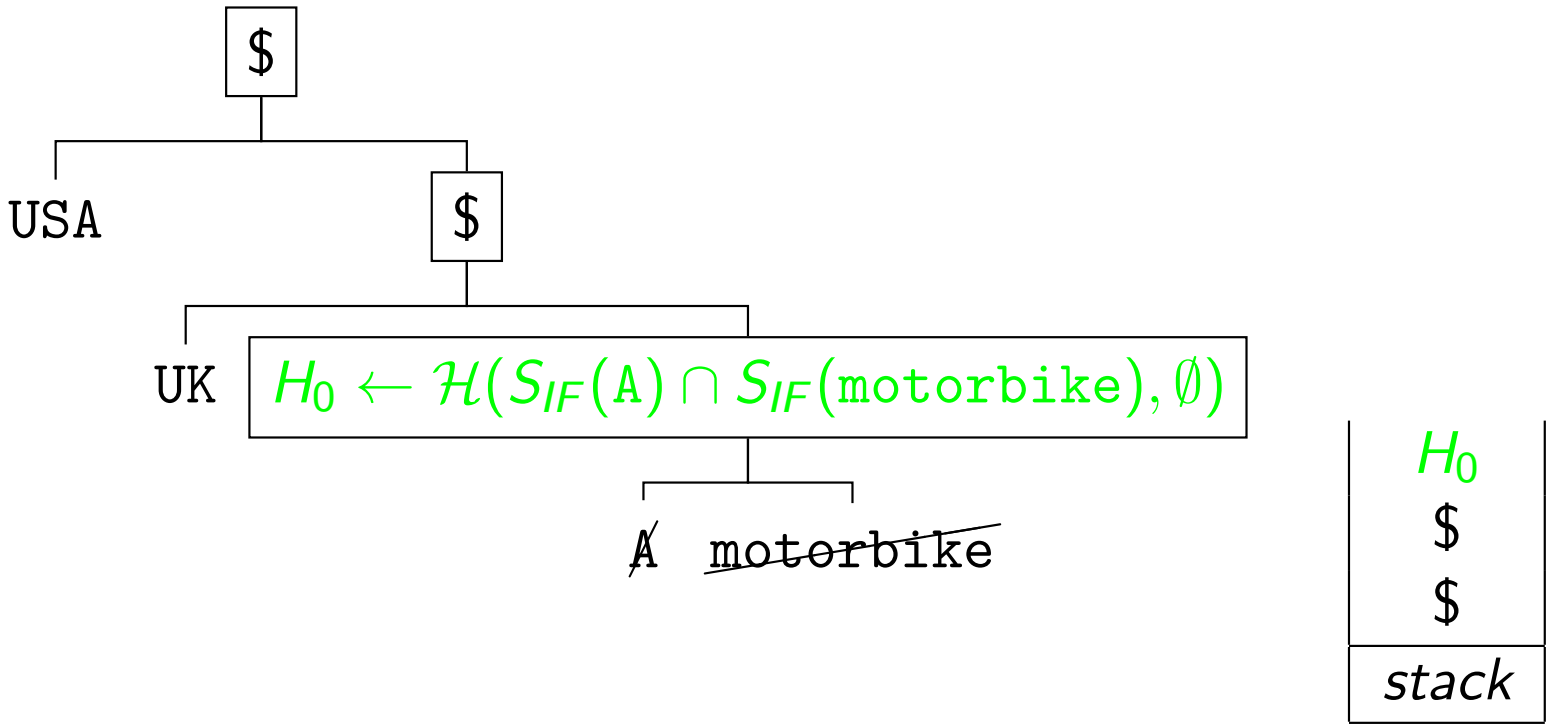
Example run of the bottom-up algorithm



$\mathcal{H}(Candidates, Lists)$ returns the "heads" of all inverted lists in *Candidates* having children in each of the elements of *Lists*.

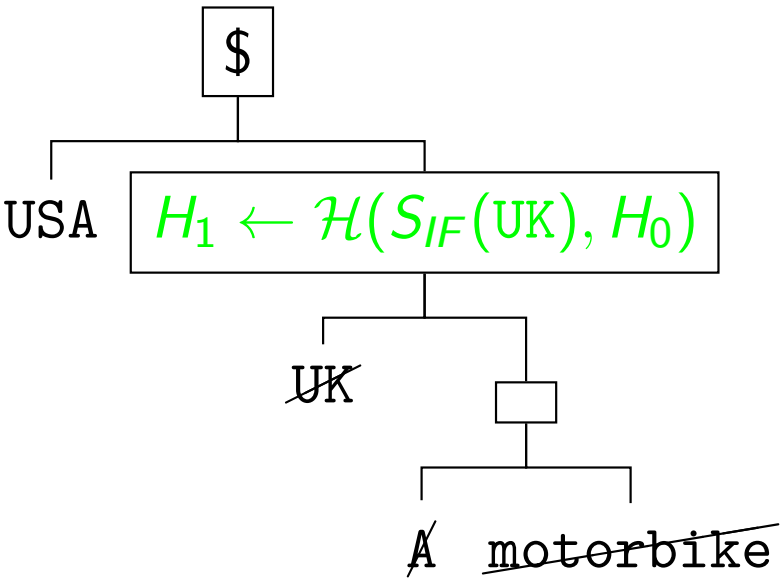
Intuitively, these "head" nodes successfully cover the full subquery rooted at the current node.

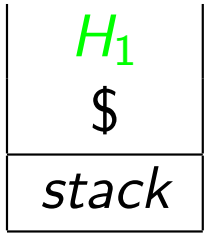
Example run of the bottom-up algorithm



(d) We evaluate the inner-most set (i.e., construct $Candidates = S_{IF}(\mathtt{A}) \cap S_{IF}(\mathtt{motorbike})$), pop the stack (i.e., construct $Lists = \emptyset$), and put the local result H_0, consisting of all the heads of *Candidates*, onto the stack.

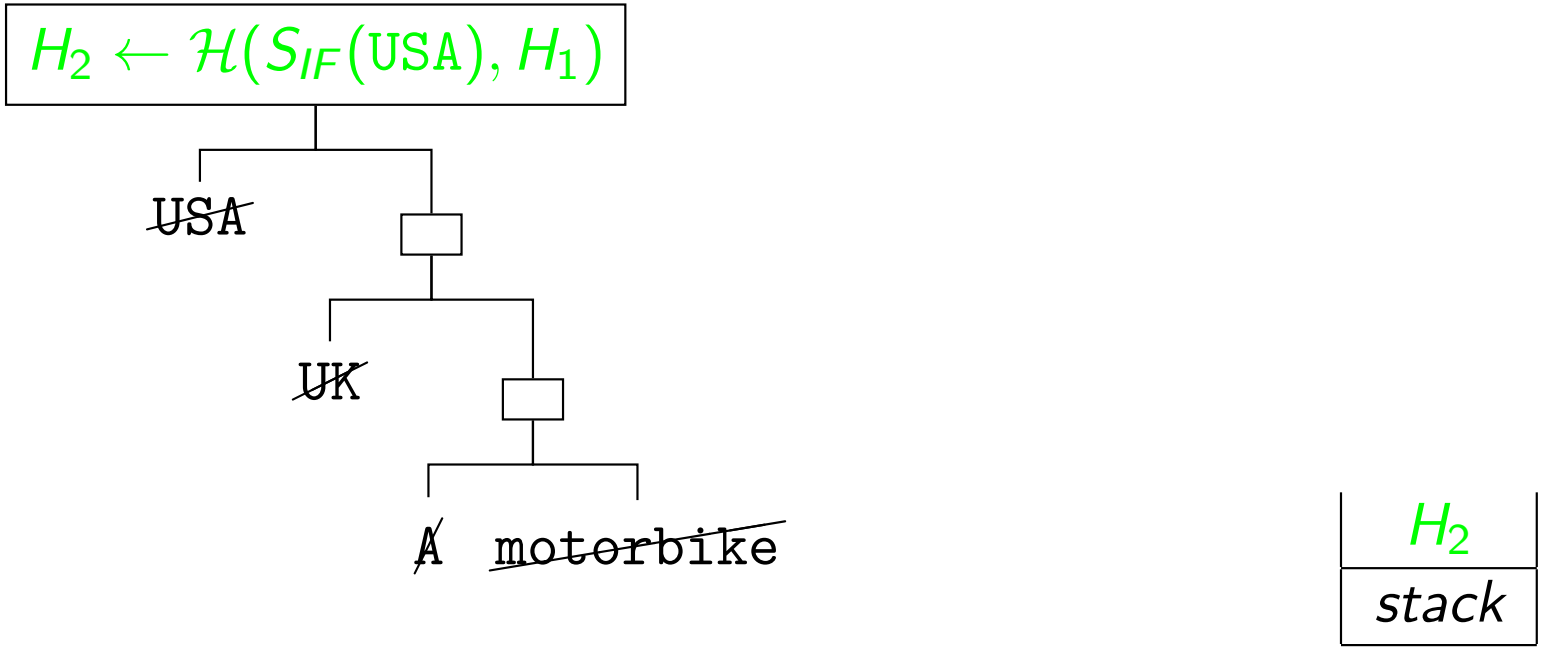
Example run of the bottom-up algorithm





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

Example run of the bottom-up algorithm



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

The bottom-up algorithm

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

The bottom-up algorithm

Bottom-up-containment

input: query q, inverted file S_{IF}

output: $q \bowtie_{\subseteq} S$

1: $s \leftarrow$ a new empty stack
2: Bottom-up-interior$(root(q), s, S_{IF})$
3: **return** pop(s)

The bottom-up algorithm

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

1: $s \leftarrow$ a new empty stack
2: Bottom-up-interior($root(q), s, S_{IF}$)
3: **return** pop(s)

In the next slide, let $nodes(n)$ and $\ell(n)$ denote the set of non-atomic and atomic elements, resp., of set n.

The bottom-up algorithm: bottom-up-interior

input: query node n, stack s, inverted file S_{IF}

The bottom-up algorithm: bottom-up-interior

input: query node n, stack s, inverted file S_{IF}

```
push($, s)
for all c ∈ nodes(n) do
    Bottom-up-interior(c, s, S_IF)
end for
```

The bottom-up algorithm: bottom-up-interior

input: query node n, stack s, inverted file S_{IF}

```
push($, s)
for all c ∈ nodes(n) do
    Bottom-up-interior(c, s, S_IF)
end for
Lists ← ∅
while peek(s) ≠ $ do
    Lists ← Lists ∪ {pop(s)}
end while
pop(s)
```

The bottom-up algorithm: bottom-up-interior

input: query node n, stack s, inverted file S_{IF}

```
push($, s)
for all c ∈ nodes(n) do
    Bottom-up-interior(c, s, S_IF)
end for
Lists ← ∅
while peek(s) ≠ $ do
    Lists ← Lists ∪ {pop(s)}
end while
pop(s)
if ∀L ∈ Lists : L ≠ ∅ then
    Candidates ← ⋂_{ℓ∈ℓ(n)} S_IF(ℓ)
    Heads ← {h | ∃C : [(h, C) ∈ Candidates
                                    ∧∀L ∈ Lists : C ∩ L ≠ ∅]}
    push(Heads, s)
```

The bottom-up algorithm: bottom-up-interior

input: query node n, stack s, inverted file S_{IF}

```
push($, s)
for all c ∈ nodes(n) do
    Bottom-up-interior(c, s, S_IF)
end for
Lists ← ∅
while peek(s) ≠ $ do
    Lists ← Lists ∪ {pop(s)}
end while
pop(s)
if ∀L ∈ Lists : L ≠ ∅ then
    Candidates ← ⋂_{ℓ∈ℓ(n)} S_IF(ℓ)
    Heads ← {h | ∃C : [(h, C) ∈ Candidates
                                    ∧∀L ∈ Lists : C ∩ L ≠ ∅]}
    push(Heads, s)
else
    push(∅, s)
end if
```

The bottom-up algorithm

It is easy to see that the run-time of the bottom-up algorithm is $\mathcal{O}(|q| * |S|)$.

- The top-down approach has the same complexity

The bottom-up algorithm

It is easy to see that the run-time of the bottom-up algorithm is $\mathcal{O}(|q| * |S|)$.

- The top-down approach has the same complexity

Also in the paper

- optimizations:
  - caching of popular atoms in the database
  - Bloom-filters for early termination
- extensions:
  - adaptations (to both algorithms) for superset joins, set equality joins, and set-overlap joins
  - adaptations (to both algorithms) for "isomorphic" and "homeomorphic" containment

Empirical study: set up

Environment

- Machine: Fedora 12 / 64-bit Linux, with 8 x 2 Ghz processors, 144 GB RAM, and a 2.8 TB disk
- Inverted file: external memory hash table using Tokyo Cabinet (version 1.24) with main memory caching disabled
- Implementation: single-threaded Java (version J2SE-1.5) process

Empirical study: set up

Data

- Synthetic: atomic values drawn from 10,000,000 labels
  - uniformly: wide and deep sets
  - skewed Zipfian: wide and deep sets, for various skew factors
- Real
  - Twitter tweets
  - DBLP articles

Empirical study: set up

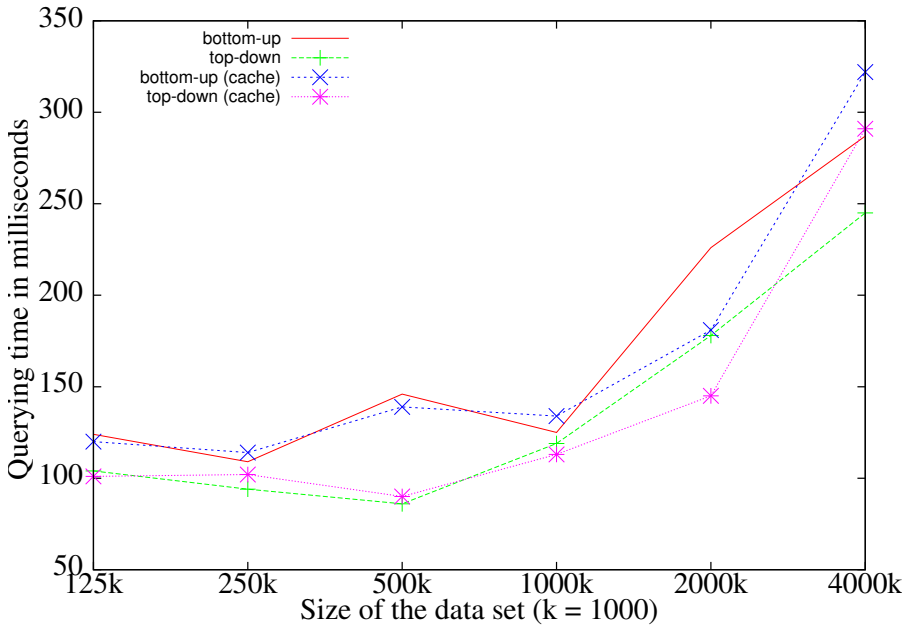
Queries

From each data set, we selected 100 arbitrary objects as queries.

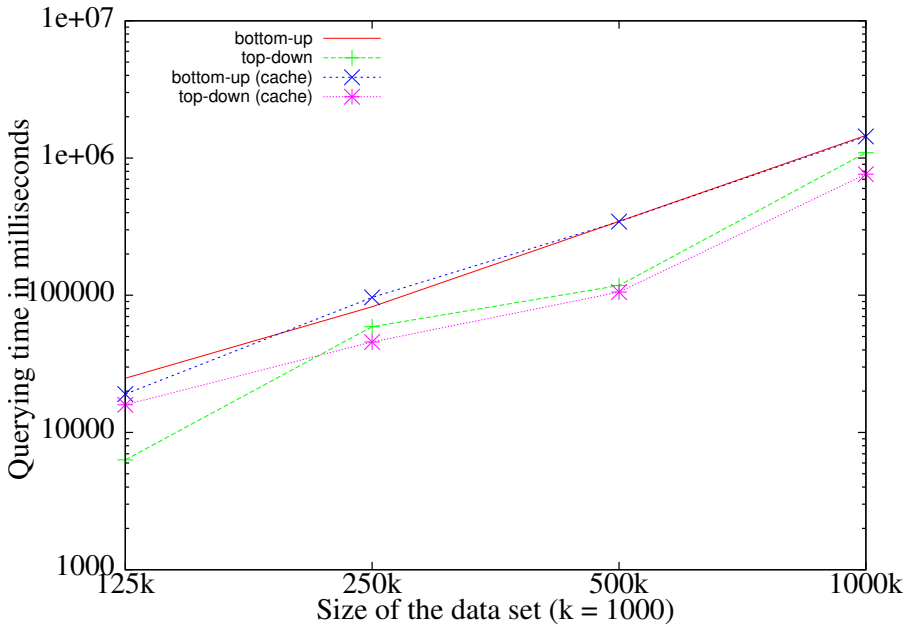
We distorted half of the selected queries such that they are not contained in the data collection (i.e., we have 50 positive queries and 50 negative queries).

Measured elapsed time of sequentially executing the queries. For each measurement, we repeat this ten times, exclude the minimum and maximum, and report the average.

Empirical study: results



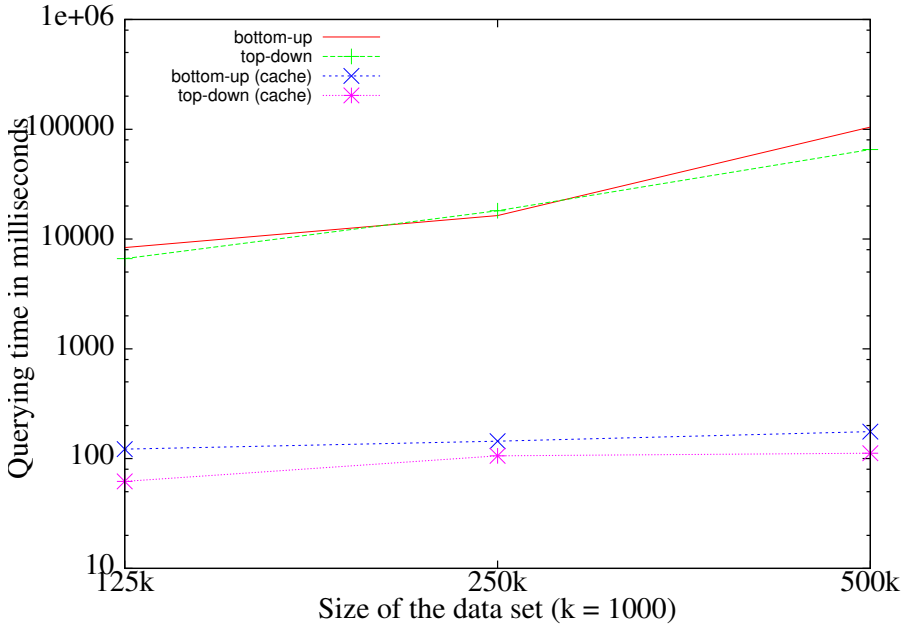
Uniform wide sets



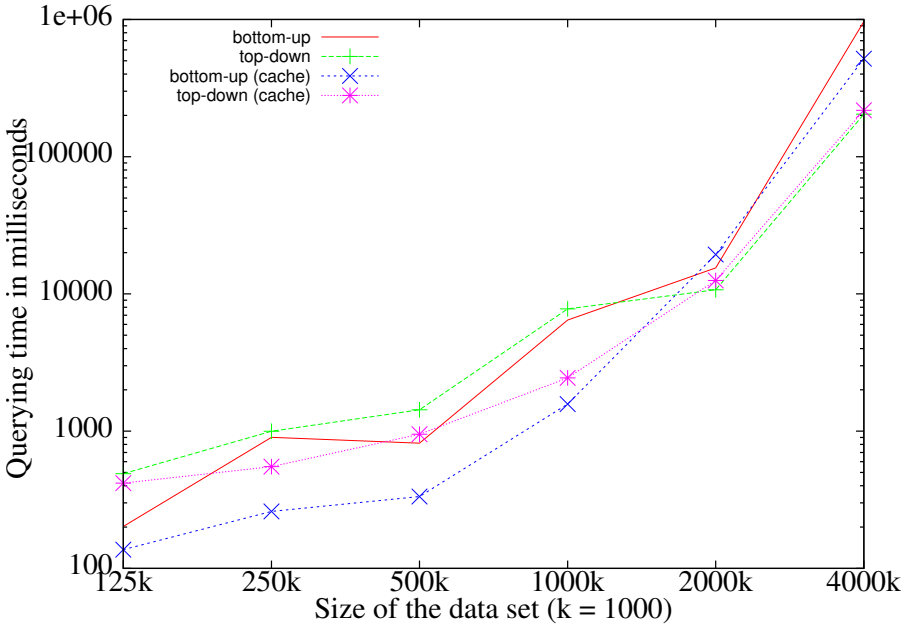
Skewed deep sets

A sampling of results on synthetic data

Empirical study: results



Twitter tweets



DBLP articles

A sampling of results on real data

Summary

Highlighted the problem of nested set containment, presented two practical solutions, with optimizations and extensions, and demonstrated their effectiveness on various data sets

Summary

Highlighted the problem of nested set containment, presented two practical solutions, with optimizations and extensions, and demonstrated their effectiveness on various data sets

Solutions are conceptually simple, using standard readily-available industrial-strength data structure

- Easy to apply for practical use
- Good foundation for further study

Future work

Many interesting research directions remain open.

- Skewed data is still challenging for our algorithms. Study applications of recent results on efficiently dealing with list intersections and data skew.
- Study natural variations of the data model (e.g., multi-set and list types; noisy and uncertain data).
- More sophisticated pruning mechanisms for nested data
- Investigate alternative caching mechanisms, e.g., caching with respect to an evolving query workload.

Future work

Many interesting research directions remain open.

- Skewed data is still challenging for our algorithms. Study applications of recent results on efficiently dealing with list intersections and data skew.
- Study natural variations of the data model (e.g., multi-set and list types; noisy and uncertain data).
- More sophisticated pruning mechanisms for nested data
- Investigate alternative caching mechanisms, e.g., caching with respect to an evolving query workload.

Thank you! Questions?