Unified Query Processing for JSON Documents and Indexes

Mihai Budiu, Microsoft Research
Gordon Plotkin*, Univ. of Edinburgh
Yuan Yu, Microsoft Research
Li Zhang, Microsoft Research

ABSTRACT

We present JPath, a JSON database query language, and its syntax, semantics, and implementation. We introduce an indexing data structure for answering JPath queries, and provide a theory unifying query execution on data and index trees using operations on matrices with lattice-valued elements.

1. INTRODUCTION

JSON [11] is becoming the lingua franca for computer data interchange, supplanting XML in many applications, both on the client-side and on the server-side. JSON is now the format of choice for representing data in many commercial products and research projects [4, 18, 22].

One important operation performed on JSON databases is the retrieval of JSON documents with a certain structure. We present JPath, a succinct query language for performing structural queries on JSON objects, closely related to Core XPath [7]. We model JSON objects as node-labeled unordered trees; JPath is composed of a small set of query operations that map naturally to the tree data model. While very succinct, JPath captures essential aspects of querying tree data.

Our aim is to enable the construction of efficient query engines for this core language that can potentially be extended by layering additional functionality on top. While XML data have been modeled as trees as well, the tree model fits JSON particularly well because of the simplicity of the JSON format. We have implemented our query language as a practical system, closely following the theoretical design.

We use *index trees* to accelerate query execution when many trees have a similar structure. An index tree summarizes the structure of an indexed collection of trees. The query process is executed in two stages: first, a query is run directly on the index tree to quickly eliminate non-matching trees; next, we run the query on each individual tree which has survived the first stage. For this process to work properly, we make sure the query execution on the index tree is *conservative*: it may provide false positives, but never false negatives. *Indexing policies* control the trade-off between index tree size, query execution time, and result accuracy.

Similar index structures have been known since the introduction of DataGuides [6]. For example in [16] they are provided for a language of regular path expressions and in [12, 21] for branching path query languages. A main theme in such works is the construction of optimal index trees for precise indexing. Further work, including [12], and surveyed in [10], traded off index tree size at the expense of precision for a smaller class of queries.

We focus on processing large document databases, and thus we optimize by trading-off precision for size, for better locality (e.g., to allow the indices to fit in main memory). To this end, we aim only at avoiding false positives. A distinguishing feature of our query algorithm is that it is executed directly on the indices, where nodes may have multiple labels for a more compact index. The algorithm is shown to be *tight*: it always produces the most accurate result for any query for a given index tree. While we do not have any theoretical results on the efficiency of eliminating false positives for an index tree of given size, we have some encouraging experimental evidence that indices can be small and effective in eliminating non-matching documents.

As a main contribution, we give a unified view of queries on trees and index trees. The results of queries are matrices which are node-indexed (for both rows and columns) over a distributive lattice, and the execution of a complex query is recast as the application of matrix operations to the matrices resulting from its sub-queries. Query semantics on trees and index trees are the same except that they operate on different underlying lattices: documents are modeled using the Boolean lattice; and index trees are modeled as elements of the powerset lattice over the ground set of indexed trees. Employing this unified view, we show that the query on the index tree is indeed conservative and tight. Being based on matrix operations, this semantics leads to efficient query execution algorithms.

Organization of the paper. In Section 2 we model JSON objects as node-labeled unordered trees, and in Section 3, we present the syntax and semantics of JPath. Our main contributions are presented in Section 4, which describes the construction and the use of index trees, and in Section 5, presenting our unified formalism for querying individual documents and index trees.

In Section 6 we show how the matrix semantics can be directly translated into an implementation. Section 7 presents experimental measurements on a large-scale data-set showing that JPath indexing overhead is low and can substantially speed-up some classes of queries. Section 8 discusses the

*Work performed while visiting Microsoft Research

relation of JPath with Core XPath, and other languages for querying JSON objects.

Several optional appendices support the presentation, including proofs of our main theorems. Appendix A presents various matters needed to build a practical system based on JPath: the cut operator, handling JSON data types, user-defined node-matchers, node label computations, and lazy lattice computations. Appendix B provides an algorithm to translate a JSON document to a tree.

2. JSON DOCUMENTS AS TREES

We assume that JSON documents conform to the official JSON grammar [11], summarized as:

$$\begin{array}{lcl} value & := & basevalue \mid object \mid array \\ object & := & \{\,[label\texttt{:}value\,[\texttt{,}label : value]^*]\} \\ array & := & \texttt{[}\,[value\,[\texttt{,}value]^*]\,\texttt{]} \end{array}$$

Figure 1 shows two JSON documents giving partial information about two hypothetical companies. We represent each as an unordered tree (i.e., the order of the children of a node is unspecified). To do this, we first eliminate arrays, representing them as objects with numeric labels. Thus:

```
[ { "city": "Moscow" }, { "city": "Athens" } ]
```

is rewritten as:

```
{ 0: { "city": "Moscow" }, 1: { "city": "Athens" } }
```

The nodes of the trees are labeled with typed values: the root is labeled with an empty label; each internal node is labeled with a JSON object label (strings or integer indices in the case of array elements); and each leaf is labeled with a JSON basevalue. Figure 1 shows the trees corresponding to the two JSON documents. The tree representation models JSON documents accurately, except for the ordering of siblings[1].

Notation: We write Σ for the set of labels, taken to be a collection of strings over some given alphabet. We use $\top$ for "true" and $\bot$ for "false". We use $\mathcal{P}(X)$ to denote the power-set of a set X, i.e., $\mathcal{P}(X) = \{x \mid x \subseteq X\}$. We write $[n]$ for the set $\{1, \ldots, n\}$.

We define a *labeled rooted JSON tree* as a quadruple:

$$t = (V_t, C_t, \texttt{label}_t, \texttt{root}_t).$$

where:

- V_t is the set of nodes of t,
- $C_t : V \to \mathcal{P}(V)$ gives the set of children $C_t(v)$ of a given node v. It describes the *child relation*, assumed to form a tree,
- $\texttt{label}_t : V \to \Sigma$ assigns a label to each node, and
- $\texttt{root}_t \in V$ is the root of t.

[1]If desired our scheme could be adapted to maintain information about this ordering by using an encoding similar to the array encoding.

We extend C_t to operate on sets of nodes: for $U \subseteq V$, define $C_t(U) =_{\text{def}} \cup_{v \in U} C_t(v)$. We drop the index t when it is clear from the context, writing just $(V, C, \texttt{label}, \texttt{root})$.

3. THE JPATH QUERY LANGUAGE

In this section we define the core of the JPath query language, the supplementary cut operator and other additions required in a practical implementation are discussed in Appendix A.

Our query language is functional: a query does not mutate the trees queried. When a query is executed on a collection of JSON trees, it is executed independently on all the trees in the collection. Running the query on a tree produces a set of nodes in the tree.

JPath Syntax: The following are special characters: `*`, `^`, `&`, `|`, `/`, (,). The complete syntax of JPath queries is given in Figure 2. There are eight different ways to construct a query: three primitive, three binary operators, all associative; and two unary operators. The primitives include *nodematchers*; they are ranged over by the metavariable **nm**, and denote functions from labels to Boolean values.

$query :=$	**nm**	nodematcher
	ε	empty query
	`/`	child operator
	$query\, query$	query sequence
	$(query \texttt{ | } query)$	logical or
	$(query \texttt{\&} query)$	logical and
	$(query\texttt{*})$	Kleene star
	$(\texttt{\^}query)$	snap operator

Figure 2: Syntax of JPath query language.

JPath Semantics: A query is always applied at a tree node and returns a set of tree nodes. For a given tree t we write $\mathcal{T}_t\,[\![q, v]\!] \subseteq V_t$ for the meaning of query q at a node $v \in V_t$. When we apply a query q to a tree we apply it to the root node of the tree, yielding $\mathcal{T}_t\,[\![q]\!] =_{\text{def}} \mathcal{T}_t\,[\![q, \texttt{root}_t]\!]$. We say that "q matches t" if it produces a non-empty set of result nodes. In this section we assume a fixed JSON tree t and drop the subscripts, writing $\mathcal{T}[\![q, v]\!]$, etc.

Figure 3 gives the semantics of JPath queries, where q^n abbreviates $\overbrace{q \ldots q}^{n}$ (i.e., q repeated n times). We assume a given semantics $\mathcal{N}[\![\textbf{nm}]\!] : \Sigma \to \{\top, \bot\}$ for nodematchers.

As queries are created by stitching together simpler queries together, e.g., using `&`, their semantics is defined recursively.

Here is an informal description of the semantics:

- Nodematchers denote given Boolean functions that are applied to the current node label. The (singleton set of the) node itself is returned if the function returns "true" $\top$; otherwise the empty set. One common kind of nodematcher maps a single label to $\top$; such nodematchers are written as the label themselves.
- Epsilon ε is the empty query; it always returns the node to which it is applied.

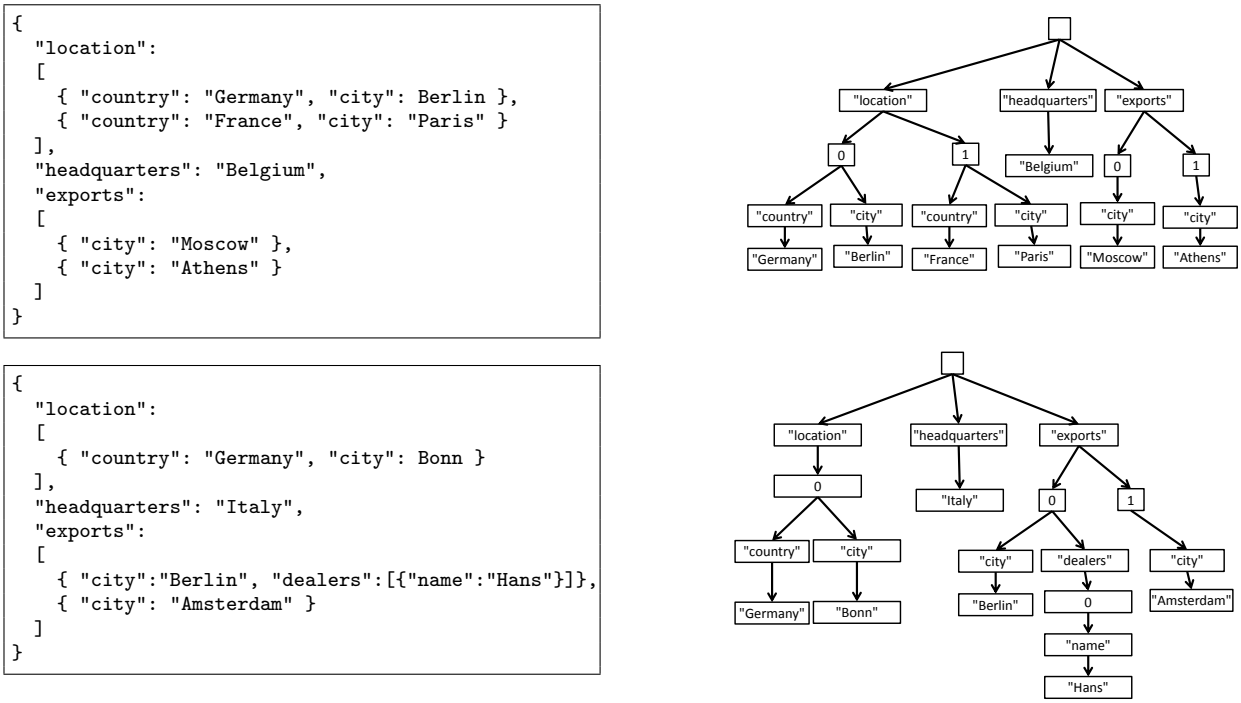


Figure 1: Two JSON documents and their tree representations.

$$
\begin{array}{lcl}
\mathcal{T}[\![\mathbf{nm}, v]\!] & = & \begin{cases} \{v\} & \text{if } \mathcal{N}[\![\mathbf{nm}]\!](\texttt{label}(v)) = \top \\ \emptyset & \text{otherwise.} \end{cases} \\
\mathcal{T}[\![\varepsilon, v]\!] & = & \{v\} \\
\mathcal{T}[\![/, v]\!] & = & C(v) \\
\mathcal{T}[\![q_1 \;\; q_2, v]\!] & = & \cup_{u \in \mathcal{T}[\![q_1, v]\!]} \mathcal{T}[\![q_2, u]\!] \\
\mathcal{T}[\![q_1 \;\texttt{|}\; q_2, v]\!] & = & \mathcal{T}[\![q_1, v]\!] \cup \mathcal{T}[\![q_2, v]\!] \\
\mathcal{T}[\![q_1 \;\texttt{\&}\; q_2, v]\!] & = & \mathcal{T}[\![q_1, v]\!] \cap \mathcal{T}[\![q_2, v]\!] \\
\mathcal{T}[\![q\texttt{*}, v]\!] & = & \cup_{n \geq 0} \mathcal{T}[\![q^n, v]\!] \\
\mathcal{T}[\![\texttt{\^{}}q, v]\!] & = & \begin{cases} \{v\} & \text{if } \mathcal{T}[\![q, v]\!] \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

Figure 3: JPath semantics.

• The *child* operator `/` returns all the children of a node.

• The *sequence* operator concatenates two queries. To compute the result of $q_1 q_2$ at a node v, one computes the results of q_1 at v and then applies q_2 to each of them. The result of the combined query is the union of these applications of q_2. For example, the query `//` matches all the grand-children of the root node. The query `/"headquarters"/"Italy"` is a sequence of 4 queries, two nodematchers, and two children, and in Figure 1 only produces a match in the second tree.

• The operators *or* `|` and *and* `&` combine two queries. They apply both queries at the current node, but "or" unions of the results while "and" intersects them.

• The *star* operator `*` is inspired by the Kleene star operator from regular expressions. The result of the star of a query q is the union of the results obtained from applying q several times. We define its *k-bounded approximation* as $\cup_{n \leq k} \mathcal{T}[\![q^n, v]\!]$. The infinite sequence of k-bounded approximations is monotonically increasing, and so converges in at most $|V|$ steps. For example, the query `(/*)/"Berlin"` produces results in all trees with a non-root node labeled `"Berlin"`.

• The *snap* operator `^` is more unusual. It checks whether a query matches a node and then discards the result of the match. It can be used to detect complex topologies, e.g., `(^/"headquarters"/"Italy")/"exports"/"city"` produces matches of nodes `"Berlin"` and `"Amsterdam"` in our second tree. The operator is called "snap" because it resumes query matching at the node where it is applied; in this example, the `"exports"` node must be a sibling of the `"headquarters"` node.

We refer to Appendix C for some more complex examples of JPath queries.

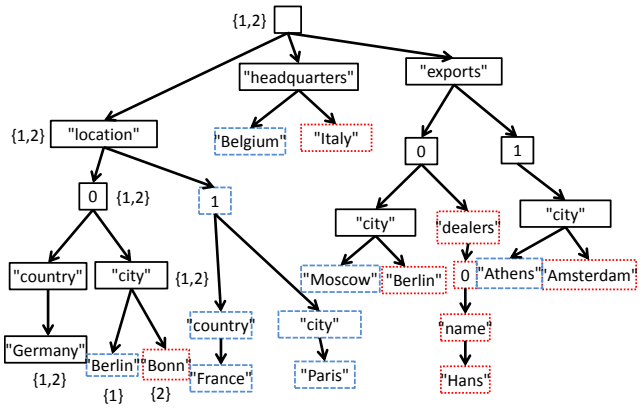
All our query operators are monotone: when applied to a tree with a larger set of nodes and the same root they will produce a larger result. This ensures both an efficient query matching algorithm and an effective indexing method. This is why we did not introduce a negation query.

4. INDEXING FOR JPATH QUERIES

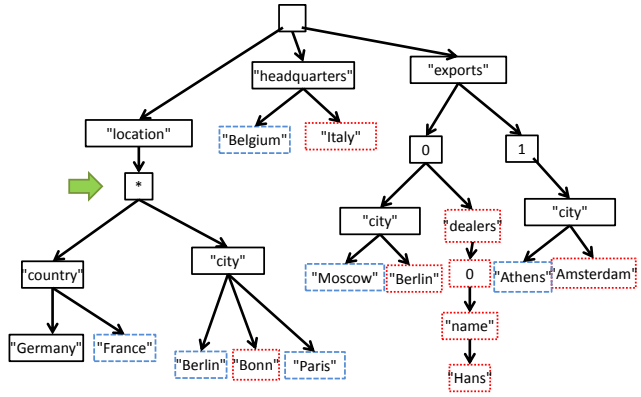
4.1 Building Index Trees

The idea of indexing is to represent an indexed set of trees $T = \{t_i \mid i \in [n]\}$ by a single *summary* node-labeled index tree, denoted by $\sigma(T)$ (or just σ when T is clear from the context). We require that the roots of all the $t_i \in T$ have the same label[2]. The index tree σ is an approximate summary of

[2]With the tree representation of JSON described in Section 2, our



(a) An index tree for the two documents in Figure 1. The black nodes with continuous lines are common to both trees, the dashed blue nodes come only from the first tree, and the dotted red nodes from the second tree. The sets $\{1\}$, $\{2\}$ and $\{1,2\}$ denote the inverted indexes associated to each node (only some are shown).



(b) Compressing the labels of the index tree from (a). The arrow points to a node obtained by merging and collapsing `0` and `1` nodes in (a). The merged node has a special label $*$, denoting "all possible labels". Note that the descendants of `0` and `1` get recursively merged into the descendants of $*$.

Figure 4: Examples of index trees.

the trees in T. Queries can be executed directly on σ, with result both a set of nodes, and a set $I \subseteq [n]$ of indices of trees that may match the query; I contains no false negatives (i.e., any t_i with $i \notin I$ does not match the query), though it may contain false positives.

Each node in the index tree corresponds to a set of nodes from the component trees via maps $r_i : V_{t_i} \to V_\sigma$, associating to each t_i node its representative in σ.

Figure 4(a) shows a possible index tree built for our example trees in Figure 1. Intuitively, we try to "overlap" the component trees and represent common structure only once, similar to trie data structures. In the figure we use color and all black, continuous line nodes exist in both component trees, while the dashed blue nodes exist only in the first tree, and the dotted red nodes only in the second tree. Each node in the index maintains an *inverted index* with a list of the corresponding tree indices (so a black simple node would have the list $\{1,2\}$, a dashed blue node the list $\{1\}$, and a dotted red node the list $\{2\}$). We associate to each node of $v \in V_\sigma$ an inverted index $\mathrm{I}(v) =_{\text{def}} \{i \mid \exists u \in V_{t_i}.\ r_i(u) = v\}$. The figure shows the inverted indexes of some of the nodes; all nodes with the same color and line style have the same inverted index.

To allow compact index trees, a node $v \in \sigma$ may have multiple labels. If the set of labels in a node grows large we can over-approximate it, e.g., using the set of all labels (depicted by a special symbol $*$). This makes indexing less precise but requires less storage space and shortens query evaluation time. In figure 4(b) one sees that the `0` and `1` nodes have been "merged" and then "collapsed" into $*$.

The index tree is most useful when the JSON trees in the database fall into a small number of groups, each of whose members have a "similar" structure, as happens frequently in practice. The common structure is represented only once in the index tree, providing a compact representation for a large number of documents.

Formalizing, an index tree σ for an indexed set of trees $T = \{t_i \mid i \in [n]\}$ is a structure:

$$(V_\sigma, C_\sigma, \texttt{label}_\sigma, \texttt{root}_\sigma, \mathrm{I}_\sigma)$$

where:

- V_σ is the set of nodes of the index tree,
- $C_\sigma : V_\sigma \to \mathcal{P}(V_\sigma)$ gives the *child relation*, assumed a tree,
- $\texttt{label}_\sigma : V_\sigma \to \mathcal{P}(\Sigma)$ is a labeling function, which associates a *set* of labels from Σ to each node,
- $\texttt{root}_\sigma \in V_\sigma$ is the root of t, and
- $\mathrm{I}_\sigma : V_\sigma \to \mathcal{P}([n])$, is an inverted index, which associates a set of tree indices to each node.

We say that maps $r_i : V_{t_i} \to V_\sigma, i \in [n]$ *faithfully represent* t_i *in* σ if they obey the following conditions:

Root and Child consistency: Roots and children in a tree map to roots and children in the index tree:

$$\forall i \in [n].\ r_i(\texttt{root}_{t_i}) = \texttt{root}_\sigma\,,$$
$$\forall i \in [n], v, w \in V_{t_i}.\ w \in C_{t_i}(v) \implies r_i(w) \in C_\sigma(r_i(v))\,.$$

Label consistency: Each index tree node label includes the labels of all represented tree nodes:

$$\forall i \in [n], t, v \in V_{t_i}.\ \texttt{label}_{t_i}(v) \in \texttt{label}_\sigma(r_i(v))\,.$$

Inverted index consistency: If v is a node of t_i, then i is in the inverted index of $r_i(v)$:

$$\forall i \in [n], v \in V_{t_i}.\ i \in \mathrm{I}(r_i(v))\,.$$

4.2 Index Tree Construction

Figure 5 gives a pseudocode implementation that builds an index tree from an indexed set of trees $T = \{t_i \mid i \in [n]\}$. The function MERGE has arguments a parent node and a subset of its children; the children are merged into a single node,

JSON tree roots always have an empty label, so this constraint is always satisfied.

removed from the tree and replaced with the merged node. The merged node label is the union of all children's labels. A policy (specified via a labelThreshold value) controls when labels are collapsed (into a "*" label).

```
function MERGE(node parent, node[ ] nodes):node
    result = new node();
    label(result) = ∪_{v∈nodes} label(v)
    if label(result).Length > labelThreshold then
        label(result) = *                    ▷ Approximate label
    end if
    if parent ≠ null then
        parent.children.Remove(nodes)
        parent.children.Append(result)
    end if
    result.children = ∪_{v∈nodes} v.children
    result.parent = parent
    I(result) = ∪_{v∈nodes} I(v)
    return result
end function

procedure BUILDINDEX(jsonTree[ ] trees)
    for all i in 1...trees.MaxIndex, v in V_{trees[i]} do
        I(v) = {i}                           ▷ Create an inverted index
    end for
    allRoots = {root_t | t ∈ trees}
    indexroot = MERGE(null, allRoots)
    while ∃v ∈ V_σ. (c = PICKCHILDREN(v)) ≠ ∅ do
        MERGE(v, c)
    end while
end procedure
```

Figure 5: Pseudocode for building an index tree.

The main procedure is BUILDINDEX. It starts by creating an inverted index for each node containing just the index of its parent tree, in fact transforming each tree into an index tree representing just the tree itself. A policy controls which nodes are merged and is expressed in terms of an unspecified function called PICKCHILDREN, which selects a subset of children from each node to merge. In practice we expect that this function chooses children of its argument node v with identical labels for merging; this works well when many trees in the collection have similar structures.

4.3 Running queries on index trees

The index tree constructed above can be viewed as a compact "union" of all the JSON trees in the database. Since the index is also a tree, we can run the query against it to emulate querying against all the trees in the database. Then we can use the inverted index to find out the "candidate trees" which might match the query. The crucial property is conservativity, that we never miss any result using this procedure. In this section, we first give a conservative algorithm which is direct extension of the previous query matching algorithm. We then present a (still conservative) refinement which gives a more precise candidate set and leads us to our unified framework for querying individual trees and index trees (in Section 5).

We slightly modify the query matching algorithm, in order to handle multiply-labeled nodes. The inverted indexes are used to retrieve the trees that may match the query from the set of nodes the algorithm returns. For example, a query that returns a dotted red node by running on the index tree from Figure 4(a) or (b) may only match the second of the documents in Figure 1.

We write $\mathcal{I}_\sigma[\![q, v]\!]$ for "the result of query q run on index tree σ when applied to a node $v \in V(\sigma)$". The definition of $\mathcal{I}$ is almost identical to that of $\mathcal{T}$ as in Figure 3 except that the nodematcher case is slightly different. This is because each node in the index tree may result from merging several nodes in the JSON trees and therefore may be associated with multiple labels. A node matches a nodematcher as long as any of the associated labels does.

$$\mathcal{I}[\![\mathbf{nm}, v]\!] = \begin{cases} \{v\} & \text{if } \exists l \in \texttt{label}(v).\ \mathcal{N}[\![\mathbf{nm}]\!](l) = \top, \\ \emptyset & \text{otherwise.} \end{cases}$$

The following theorem states that the result of a query q run on an index tree $\sigma(T)$ is a conservative approximation of the execution of the query on any tree in T; the straightforward proof is by induction on q and can be found in Appendix D.

THEOREM 4.1. *Let $\{r_i \mid i \in [n]\}$ faithfully represent a set of trees $\{t_i \mid i \in [n]\}$ in an index tree σ. Then:*

$$\forall q, i \in [n], v \in V_{t_i}.\ r_i(\mathcal{T}_{t_i}[\![q, v]\!]) \subseteq \mathcal{I}_{\sigma(T)}[\![q, r_i(v)]\!].$$

Thus running q starting at node v in a tree t_i produces a set of nodes $\mathcal{T}[\![q, v]\!]$ which correspond under r_i to the set of nodes produced by running q on the σ node corresponding to v: see Figure 6.

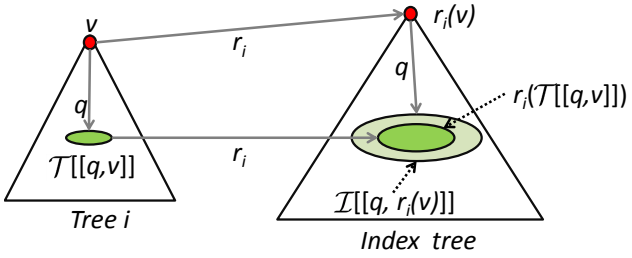


Figure 6: Indexes provide an over-approximation. Running the query on an index tree will always give a result which contains the result obtained running the query on any tree.

Executing a query on the index tree gives a set of index tree nodes $\mathcal{I}[\![q]\!] =_{\text{def}} \mathcal{I}[\![q, \texttt{root}_\sigma]\!]$. By Theorem 4.1 we have $r_i(\mathcal{T}_{t_i}[\![q]\!]) \subseteq \mathcal{I}_{\sigma(T)}[\![q]\!]$, for $i \in [n]$. So, using index consistency, we can look at $\cup_{v \in \mathcal{I}[\![q]\!]} \mathrm{I}(v)$, the inverted indexes of these nodes, to find out which trees may match the query.

Looking just at $\mathcal{I}[\![q, v]\!]$ may be too conservative:

EXAMPLE 4.2. *Consider two simple JSON documents, each describing a simple key-value pair, with keys labeled* K *and values labeled* V*:*

$$t_1 = \{\texttt{K} : \texttt{k1}, \texttt{V} : \texttt{v1}\} \quad t_2 = \{\texttt{K} : \texttt{k2}, \texttt{V} : \texttt{v2}\}$$

and an index tree with the shape:

$$\sigma = \{\texttt{K} : \{\texttt{k1}, \texttt{k2}\}, \texttt{V} : \{\texttt{v1}, \texttt{v2}\}\}\,.$$

Consider the query q=/^(K/k1)/V/ *which fetches the value with associated key* k1*. The query result* $\mathcal{I}[\![q]\!]$ *contains both nodes labeled* V *in* σ*, and as a result of running the query on* σ *we cannot eliminate any JSON tree. However, the sub-query* $q' = \texttt{K}/\texttt{k1}$ *is very selective, and only matches nodes in* t_1*. But when the algorithm executes the step for the* ^ *operator it "forgets" this fact.*

We next augment query processing on index trees to *record the reason* for a node to be included in the result; this makes the algorithm more selective. For each query step we maintain not just a set of index tree nodes, but also for each such node a set of the tree indices that contributed to selecting this node; this set is obtained from the inverted indexes. For example, in the above example, when matching the subquery ^(K/k1) on σ, we also retain the inverted index stored on k1, $\{1\}$, and carry this information through the query. So only t_1 appears in the final result. Since we need to maintain an index set for each node in the result, we define the semantics $\mathcal{J}[\![q,(u,v)]\!] \subseteq [n]$ of a query q with respect to pairs of nodes u, v; this semantics represents the set of trees which may produce v in the result when matching q against the root node u. If $\mathcal{J}[\![q,(u,v)]\!] = \emptyset$, then $v \notin \mathcal{T}_q[\![u]\!]$. The formal definition of $\mathcal{J}[\![q,(u,v)]\!]$ is shown in Figure 7.

- $q = \textbf{nm}$, $q = \varepsilon$, $q = /$:

$$\mathcal{J}[\![q,(u,v)]\!] = \begin{cases} \mathrm{I}(v) & \text{if } v \in \mathcal{I}[\![q,u]\!], \\ \emptyset & \text{otherwise.} \end{cases}$$

- $\mathcal{J}[\![q_1\, q_2,(u,v)]\!] = \cup_{w\in\sigma}(\mathcal{J}[\![q_1,(u,w)]\!] \cap \mathcal{J}[\![q_2,(w,v)]\!])$
- $\mathcal{J}[\![q_1 \mid q_2,(u,v)]\!] = \mathcal{J}[\![q_1,(u,v)]\!] \cup \mathcal{J}[\![q_2,(u,v)]\!]$
- $\mathcal{J}[\![q_1 \,\&\, q_2,(u,v)]\!] = \mathcal{J}[\![q_1,(u,v)]\!] \cap \mathcal{J}[\![q_2,(u,v)]\!]$
- $\mathcal{J}[\![q*,(u,v)]\!] = \cup_{n\geq 0}\mathcal{J}[\![q^n,(u,v)]\!]$
- $\mathcal{J}[\![\hat{}q,(u,v)]\!] = \begin{cases} \mathrm{I}(u) \cap (\cup_{w\in\sigma}\mathcal{J}[\![q,(u,w)]\!]) & \text{if } v = u, \\ \emptyset & \text{otherwise.} \end{cases}$

Figure 7: Refined index tree query matching function.

For any $u, v \in V_\sigma$, $\mathcal{J}[\![q,(u,v)]\!]$ is defined, and, as can be proved by induction, $\mathcal{J}[\![q,(u,v)]\!] \subseteq \mathrm{I}(u)$. In addition, we have the following property of $\mathcal{J}$. The proof is again by induction on q.

THEOREM 4.3. *Let* $\{r_i \mid i \in [n]\}$ *faithfully represent a set of trees* $\{t_i \mid i \in [n]\}$ *in an index tree* σ*. Then:* $\forall q, i \in [n], u, v \in V_{t_i}.\ v \in \mathcal{T}_{t_i}[\![q,u]\!] \implies i \in \mathcal{J}[\![q,(r_i(u),r_i(v))]\!]$

To find the complete set of trees that may match a query we union all witnesses: $\mathcal{J}[\![q]\!] =_{\text{def}} \cup_{v\in\sigma}\mathcal{J}[\![q,(\texttt{root}_\sigma,v)]\!]$. Then the above theorem implies that if q matches t_i then $i \in \mathcal{J}[\![q]\!]$, i.e. $\mathcal{J}[\![q]\!]$ does not miss any query results. Since $\mathcal{J}[\![q,(u,v)]\!]$ "remembers" more information than $\mathrm{I}(\mathcal{I}[\![q,u]\!])$, it provides a more precise approximation.

EXAMPLE 4.4. *Consider again Example 4.2. We obtain:* $\mathcal{J}[\![q]\!] = \{1\}$*, which filters out* t_2*.*

5. LATTICE SEMANTICS OF JPATH

We now present a single function which generalizes the matching functions of Sections 3 and 4. In the unified definition we treat all trees homogeneously, whether JSON trees or index trees. Matching returns an element of a *distributive lattice*: precise matching (as done on JSON trees) is modeled using the Boolean lattice $\mathbb{O}$, with just two elements $\bot \leq \top$; and imprecise matching (on index trees) is modeled using a *finite power-set lattice*, of sets of tree indices (the general finite powerset lattice $\mathcal{F}(X)$ consists of all finite subsets of a set X, ordered by inclusion).

Starting from the $\mathcal{J}$ function defined in Section 4.3 it is natural to model queries by matrices, viewing $\mathcal{J}[\![q,(u,v)]\!]$ as a matrix indexed by u and v. We use matrices with lattice-valued elements. Figure 8 shows the structure of our node-indexed matrices.

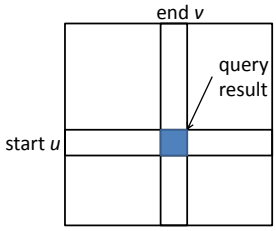


Figure 8: Matrix representation of a query operation q. Rows and columns are indexed by tree nodes. The start node u corresponds to a row, and the end node v corresponds to a column. The (u,v) entry is the result of running q starting at node u and ending at node v, taking this to be $\bot$ if node v is not part of the query result.

5.1 Notation

We work with distributive lattices, such as $\mathbb{O}$ and $\mathcal{F}(X)$, with a *join* operation $\vee$ and a *meet* operation $\wedge$:

$$(L, \bot, \top, \vee, \wedge)\,.$$

We consider *vectors* and *matrices* over L. Given a finite set of *indices* A the collection $\mathbf{Vec}_L(A)$ of L*-vectors over* A consists of all functions $\vec{x} : A \to L$. As usual, the index notation $\vec{x}[a]$ denotes the value of $\vec{x}$ at $a \in A$. Similarly, given finite sets A, B, the collection $\mathbf{Mat}_L(A,B)$ of L*-matrices over* A *and* B consists of all functions $M : A \times B \to L$, and we use the usual index notation $M[a,b]$ for the value of M at $a \in A$ and $b \in B$.

We can compose (multiply) vectors and matrices as usual. Given column vectors $\vec{x} \in \mathbf{Vec}_L(A)$, $\vec{y} \in \mathbf{Vec}_L(B)$, $M \in$

$\mathbf{Mat}_L(A,B)$, and $N \in \mathbf{Mat}_L(B,C)$, we define:

$$\begin{array}{rcl}(\vec{x}^t M)[b] & =_{\text{def}} & \bigvee_{a\in A}(\vec{x}[a] \wedge M[a,b])\,,\\ (M\vec{y})[a] & =_{\text{def}} & \bigvee_{b\in B}(M[a,b] \wedge \vec{y}[b])\,,\\ (MN)[a,c] & =_{\text{def}} & \bigvee_{b\in B}(M[a,b] \wedge N[b,c])\,.\end{array}$$

Note that $\wedge$ and $\vee$ play the roles of multiplication and addition. Using the distributivity of the lattice one can prove the associativity of binary operation, and there is an identity matrix $I_A \in \mathbf{Mat}(A,A)$, where $I_A[a,b] = \top$, if $a = b$, and $\bot$ otherwise. We write $\top_A$ for the top element of $\mathbf{Vec}(A)$: $\top_A[a] = \top$.

Both $\mathbf{Vec}_L(A)$ and $\mathbf{Mat}_L(A,B)$ inherit distributive lattice structures from L, via the point-wise orderings:

$$\begin{array}{rcl}\vec{x} \le \vec{y} & \equiv_{\text{def}} & \forall a.\ \vec{x}[a] \le \vec{y}[a]\,,\\ M \le M' & \equiv_{\text{def}} & \forall a, b.\ M[a,b] \le M'[a,b]\,.\end{array}$$

Sups and meets are point-wise, for example:

$$(M \vee N)[a,b] = M[a,b] \vee N[a,b]\,.$$

Composition commutes with binary joins, but, in general, not with binary meets.

Define the *diagonal operator* $\Delta : \mathbf{Vec}_L(A) \to \mathbf{Mat}_L(A,A)$:

$$(\Delta V)[a,b] = \begin{cases} V[a] & \text{if } a = b\,,\\ \bot & \text{otherwise.}\end{cases}$$

A matrix M with elements from $\mathbb{O}$ can be seen as a *relation* R, where $R(a,b) =_{\text{def}} (M[a,b] \neq \bot)$.

A *homomorphism* k between two finite distributive lattices L and L' is a function $k : L \to L'$ which preserves meets and joins:

$$\begin{array}{rcl}k(a \vee_L b) & = & k(a) \vee_{L'} k(b)\,,\\ k(a \wedge_L b) & = & k(a) \wedge_{L'} k(b)\,.\end{array}$$

Change of basis Given a homomorphism of finite distributive lattices:

$$k : L \to L'\,,$$

we can apply it to L-vectors and matrices to obtain L'-vectors and matrices:

$$k^o : \mathbf{Mat}_L(A,B) \to \mathbf{Mat}_{L'}(A,B)\,,$$

by function composition:

$$(k^o(M))[a,b] = k(M[a,b])\,.$$

Note that k^o commutes with the composition operations on vectors and matrices. k^o is also a lattice homomorphism, on the lattices with matrix values.

5.2 Labeled trees and queries

Consider a distributive lattice L. In this section we assume that Σ is finite and model both JSON and index trees using a single structure, called an *L-labeled tree*, of the form:

$$\tau = (V, C, \texttt{label})\,.$$

where V is a set of nodes, $C \in \mathbf{Mat}_L(V,V)$ is called the *child* matrix, and $\texttt{label} \in \mathbf{Mat}_L(V,\Sigma)$ is called the *labeling matrix*.

We define a child relation, written using the infix operator $\succ\!\!-$, defined by:

$$v \succ\!\!- w \equiv C[v,w] \neq \bot\ .$$

We require that the child relation makes the graph $(V, \succ\!\!-)$ a tree. We write $v \succ\!\!-_l w$ for $C[v,w] = l$. The child matrix is more expressive than the usual child relations as each child can be labeled with a value from L; in practice so far, the C matrix uses just $\bot$ and $\top$.

Any JSON tree t corresponds to an $\mathbb{O}$-labeled tree t^{m} in an evident way, with $\top \in C[v,w]$ iff $w \in C_t(v)$. Any indexed tree σ, with indices $[n]$, corresponds to an $\mathcal{F}([n])$-labeled tree $\sigma^{\text{m}} =_{\text{def}} (V', C', \texttt{label}')$, where $V' = V_\sigma$, C' is the evident $\{\bot, \top\}$-valued matrix corresponding to C_σ, and:

$$\texttt{label}'[v,a] =_{\text{def}} \begin{cases} \mathrm{I}_\sigma(v) & a \in \texttt{label}(v)\,,\\ \emptyset & \text{otherwise.}\end{cases}$$

Note that the labeling matrix takes over the role of both the node labels in the index tree (because it is indexed by V and Σ), and the inverted indexes (because the values are in L). The labeling matrix is also more flexible than the index tree construction, since it may have a different inverted index (lattice value) for each label.

The semantics of nodematchers **nm** is defined by a vector:

$$\mathcal{N}[\![\mathbf{nm}]\!] \in \mathbf{Vec}_L(\Sigma)\,.$$

As with C, the formalism allows the nodematcher vector to have arbitrary lattice-valued elements, but in our application these matrices contain just $\bot$ and $\top$.

Figure 9 gives the recursive definition of the matrix semantics of a query q when applied to a tree τ:

$$\mathcal{R}_\tau[\![q]\!] \in \mathbf{Mat}_L(V,V).$$

$$\begin{array}{lcl}\mathcal{R}_\tau[\![\mathbf{nm}]\!] & = & \Delta(\texttt{label}\,\mathcal{N}[\![\mathbf{nm}]\!])\\ \mathcal{R}_\tau[\![\varepsilon]\!] & = & \mathrm{I}_V\\ \mathcal{R}_\tau[\![/]\!] & = & C\\ \mathcal{R}_\tau[\![q_1 \mid q_2]\!] & = & \mathcal{R}_\tau[\![q_1]\!] \vee \mathcal{R}_\tau[\![q_2]\!]\\ \mathcal{R}_\tau[\![q_1 \,\&\, q_2]\!] & = & \mathcal{R}_\tau[\![q_1]\!] \wedge \mathcal{R}_\tau[\![q_2]\!]\\ \mathcal{R}_\tau[\![q_1 q_2]\!] & = & \mathcal{R}_\tau[\![q_1]\!]\,\mathcal{R}_\tau[\![q_2]\!]\\ \mathcal{R}_\tau[\![q*]\!] & = & \bigvee_{n\ge 0}(\mathcal{R}_\tau[\![q]\!])^n\\ \mathcal{R}_\tau[\![\hat{}q]\!] & = & \Delta(\mathcal{R}_\tau[\![q]\!]\top_V)\end{array}$$

Figure 9: Query semantics in terms of matrices.

In particular the semantics of $q*$ is well-defined as the sup is finite. This is because the elements of the power matrices $(\mathcal{R}_\tau[\![q]\!])^n$ all lie in the finite sub-lattice of L generated by the elements of the matrix $\mathcal{R}_\tau[\![q]\!]$ [5].

One can show that for each JSON tree t, $u \in \mathcal{T}_t\,[\![q, v]\!]$ iff $\mathcal{R}_{t^{\text{m}}}[\![q]\!][v,u] = \top$, and that for each index tree σ, with indices $[n]$, $\mathcal{J}[\![q,(v,u)]\!] = \mathcal{R}_{\sigma^{\text{m}}}[\![q]\!][v,u]$ (making the evident adjustments to the semantics of nodematchers).

5.3 Approximation results

Recall that the main idea of indexing is to produce a single tree that "summarizes" the structure of a collection of trees. Running the query on the index tree produces a set of tree indices that may match the query. There are two important desired properties of this process: (1) the index tree is "sound", never losing results; (2) the query function is "tight" not producing too many false positives. In this and the following section, we will formalize and prove these notions.

To prove soundness, we first define the notion of *conservative approximation* of a tree with the property that if a tree τ' is a conservative approximation of τ, then any query which matches τ is guaranteed to match τ'. Hence, once the index tree σ is a conservative approximation of all the trees in the collection, then we can guarantee soundness by using σ as the summary. The precise notion of conservative approximation is defined via subhomomorphisms between labeled trees.

A *subhomomorphism* $h : \tau \longrightarrow \tau'$ between two L-trees $\tau = (V, C, \texttt{label})$ and $\tau' = (V,' C', \texttt{label}')$ is a map $h : V \to V'$ such that:

Child consistency: If $v \mathrel{>\!\!-_l} w$ then $h(v) \mathrel{>\!\!-_{l'}} h(w)$, with $l \leq l'$, and

Label consistency:

$$\forall v \in V, l \in \Sigma.\ \texttt{label}[v, l] \leq \texttt{label}'[h(v), l].$$

We may regard h as a matrix $H \in \mathbf{Mat}_L(V, V')$, where:

$$H[v, w] = \begin{cases} \top & h(v) = w\,, \\ \bot & \text{otherwise.} \end{cases}$$

In terms of matrices, the above two conditions become:

Child consistency: $CH \leq HC'$.

Label consistency: $\texttt{label} \leq H\texttt{label}'$.

THEOREM 5.1. *For any query q we have:*

$$\mathcal{R}_\tau[\![q]\!]H \leq H\mathcal{R}_{\tau'}[\![q]\!]\,,$$

and so for any subhomomorphism $h : \tau \longrightarrow \tau'$

$$h(\mathcal{R}_\tau[\![q]\!][v, w]) \leq \mathcal{R}_{\tau'}[\![q]\!][h(v), h(w)]\,.$$

The intuition behind the second formula is that in Figure 6, where h takes the role of the function r_i. The proof is by induction on the structure of q and can be found in Appendix E.Intuitively, the theorem says that once a tree τ' conservatively approximates another tree τ, then the query result on τ' can only produce a more conservative result.

It is possible to define a notion of "lossless" approximation, with respect to JPath queries, via strong homomorphism between trees. A *strong homomorphism* $h : \tau \longrightarrow \tau'$, where $\tau = (V, \leq, \texttt{label})$ and $\tau' = (V,' \leq', \texttt{label}')$, is a map $h : V \to V'$ such that we have:

1. (a) If $v \mathrel{>\!\!-_l} w$ then $h(v) \mathrel{>\!\!-_{l'}} h(w)$, with $l \leq l'$.
   (b) For every $w \mathrel{>\!\!-_{l'}} h(v)$ we have:
   $$l' \leq \bigvee\{l \mid v' \mathrel{>\!\!-_l} v, \exists v' \in V.\ h(v') = w\}\,.$$
2. $\forall v \in V, l \in \Sigma.\ \texttt{label}[v, l] = \texttt{label}'[h(v), l]$.

Condition 1(b) ensures that there are no "superfluous" children in the relation C' or labels in $\texttt{label}'$.

In terms of matrices h obeys the above condition iff:

$$\begin{array}{rcl} CH & = & HC' \\ \texttt{label} & = & H\texttt{label}' \end{array}$$

THEOREM 5.2. *For any query q we have:*

$$\mathcal{R}_\tau[\![q]\!]H = H\mathcal{R}_{\tau'}[\![q]\!]$$

and so for any homomorphism $h : \tau \longrightarrow \tau'$,

$$h(\mathcal{R}_\tau[\![q]\!][v, w]) = \mathcal{R}_{\tau'}[\![q]\!][h(v), h(w)]\,.$$

The proof of Theorem 5.2 is similar to that of Theorem 5.1, replacing all inequalities with equalities.

Next we show that the matching results can be preserved between trees defined over different lattices via lattice homomorphisms. This is needed as an individual tree is defined over the Boolean lattice but the index tree over the powerset lattice.

Assume that we have a homomorphism of distributive lattices $k : L \to L'$, and two node matcher semantics $\mathcal{N}[\![\mathbf{nm}]\!]_L$ and $\mathcal{N}'[\![\mathbf{nm}]\!]_{L'}$, inducing two query semantics $\mathcal{R}_\tau[\![q]\!]_L$ and $\mathcal{R}_{\tau'}[\![q]\!]_{L'}$. Via the change of basis k, every L-labeled tree τ becomes an L'-labeled tree $k^o(\tau)$.

THEOREM 5.3. *If for all* **nm** *we have $k^o(\mathcal{N}[\![\mathbf{nm}]\!]_L) = \mathcal{N}'[\![\mathbf{nm}]\!]_{L'}$. Then for all q, L-labeled tree τ:*

$$k^o(\mathcal{R}_\tau[\![q]\!]_L) = \mathcal{R}_{k^o(\tau)}[\![q]\!]_{L'}\,.$$

PROOF. Using the definition of the semantics and the fact that change of basis commutes with matrix composition, I_V, and Δ, and $\wedge$, $\vee$, and $\top$. □

5.4 Indexing

With the properties proved in the previous section, we now show that the indexing method is sound and optimal for any given index tree. Suppose that we have

- an $[n]$-indexed collection of $\mathbb{O}$-labeled trees $t_i = (V_i, C_i, \texttt{label}_i)$,
- an *indexing* tree, by which we mean an $\mathcal{F}([n])$-labeled tree, $\sigma = (V, C, \texttt{label})$, and
- maps $h_i : V_i \to V \quad (i \in [n])$.

The following corollaries give conditions on the h_i that enable the answers of queries on t_i and σ to be related.

COROLLARY 5.4. ***Soundness:*** *Suppose that*

Child consistency *For all $v, w \in V_i$, if $v \mathrel{>\!\!-_i} w$ then $h_i(v) \mathrel{>\!\!-_x} h_i(w)$, with $i \in x$;*

Label consistency *For all $v \in V_i$, if* $\texttt{label}_i[v,a] = \top$ *then* $i \in \texttt{label}[h_i(v), a]$.

Then for all queries q, we have:

$$\mathcal{R}_{t_i}[\![q]\!][v,w] = \top \Rightarrow i \in \mathcal{R}_\sigma[\![q]\!][h_i(v), h_i(w)]\,.$$

i.e., if a query returns a result on a tree, it will return that tree when run on the index.

PROOF. The lattice homomorphism $k_i : \mathcal{F}([n]) \to \mathbb{O}$

$$k_i(x) = \begin{cases} \top & i \in x\,, \\ \bot & \text{otherwise.} \end{cases}$$

is a change of basis, which "projects" the index tree σ on t_i by essentially ignoring all labels in a node of σ except i, keeping one bit per node, indicating whether the node is labeled with i. Theorem 5.3 applies, and thus we have

$$k_i^o(\mathcal{R}_\sigma[\![q]\!]_{\mathcal{F}([n])}) = \mathcal{R}_{k_i^o(\sigma)}[\![q]\!]_{\mathbb{O}}.$$

Consider the function $h : V_i \to V$ defined by $h(x) = h_i(x)$. From the assumptions we can prove that h is a sub-homomorphism between t_i and $k_i^o(\sigma)$. As a consequence we have the inequality $\mathcal{R}_{t_i}[\![q]\!]H \le H\mathcal{R}_{k_i^o(\tau)}[\![q]\!] = Hk_i^o(\mathcal{R}_\sigma[\![q]\!])$.

Now suppose we have $\mathcal{R}_{t_i}[\![q]\!][v,w] = \top$. Then we have $\mathcal{R}_{t_i}[\![q]\!]H[v,h(w)] = \top$, and so, using the inequality, that $H\mathcal{R}_{k_i^o(\sigma)}[\![q]\!][v,h(w)] = \top$.
So $\mathcal{R}_{k_i^o(\sigma)}[\![q]\!][h(v),h(w)] = \top$, and the conclusion follows, using the definition of k_i. □

Theorem 4.3 is a consequence of Corollary 5.4: given r_i faithfully representing $\{t_i \mid i \in [n]\}$ in σ, one applies the corollary to t^{m}, σ^{m}, and the r_i.

COROLLARY 5.5. *Suppose that*

1. (a) $\forall v, w \in V_i$, *if* $v \succ_{-i} w$ *then* $h(v) \succ_{-x} h(w)$, *with* $i \in x$;
   (b) $\forall u \in V, w \in V_i$, *if* $u \succ_{-x} h_i(w)$, *with* $i \in x$, *then* $v \succ_{-i} w$ *for some* $v \in V_i$ *with* $h_i(v) = u$,
2. $\forall v \in V_i$, $\texttt{label}_i[v,a]$ *iff* $i \in \texttt{label}[h(v), a]$.

Then for all queries q, we have:

$$\mathcal{R}_{t_i}[\![q]\!][v,w] = \top \iff i \in \mathcal{R}_\sigma[\![q]\!][h_i(v), h_i(w)]\,.$$

PROOF. With k_i defined as previously, the assumptions ensure $h : t_i \to k_i^o(\sigma)$ is a strong homomorphism. □

We now show that the query semantics $\mathcal{R}$ is optimal: using a semantics which computes a "smaller" result will violate soundness for some set of $\mathbb{O}$-labeled trees. We define an *$\mathcal{F}([n])$-valued semantics* to be a function $\mathcal{D}$ associating an $\mathcal{F}([n])$-valued matrix $\mathcal{D}_\sigma$ to any indexing tree σ. Such a $\mathcal{D}$ is *sound* if Corollary 5.4 holds for it, i.e.: for any $[n]$-indexed collection of $\mathbb{O}$-labeled trees t_i, and maps $h_i : V_{t_i} \to V_\sigma$, obeying the conditions of the corollary, the conclusion of the corollary holds for $\mathcal{D}$.

COROLLARY 5.6. *$\mathcal{R}$ is the tightest sound $\mathcal{F}([n])$-valued semantics, i.e., for any sound semantics $\mathcal{D}$, indexing tree σ and $v, w \in V_\sigma$:*

$$\mathcal{R}_\sigma[\![q]\!][v,w] \subseteq \mathcal{D}_\sigma[\![q]\!][v,w]\,.$$

PROOF. Choose an indexing tree σ. For $i \in [n]$, let t_i be the projection of σ on i as defined in the proof of Corollary 5.4, and let h_i to be the identity on V_{t_i}. One can check that t_i, h_i', and σ satisfy the conditions of Corollary 5.5.

Now suppose that $i_0 \in \mathcal{R}_\sigma[\![q]\!][v,w]$, for $v, w \in V_\sigma$. Then, by Corollary 5.5, $\mathcal{R}_{t_{i_0}}[\![q]\!][v,w] = \top$. So, as $\mathcal{D}$ is sound, we have $i_0 \in \mathcal{D}_\sigma[\![q]\!][v,w]$, as required. □

6. QUERY MATCHING ALGORITHM

To analyze query complexity we assume that each lattice operation can be performed in unit time. We also assume that each node matcher **nm** denotes a map $\mathcal{N}[\![\mathbf{nm}]\!] : \Sigma \to \mathbb{O}$ which can be computed in unit time. In the following, m is the query size, m_0 is the number of / and $*$ operators in the query, n is the number tree nodes, and h is the depth of the index tree.

The semantics defined in Section 5 immediately gives us a recursive method for computing the query result: for each JPath query q we compute $\mathcal{R}[\![q]\!]$ inductively according to the structure of q. Let M_1, M_2 denote $n \times n$ matrices and $\vec{y}$ an n-dimensional vector. The computation involves the following matrix operations, where $\vec{y}^{\,t}$ denotes transposition:

$$M_1 \vee M_2 \quad M_1 \wedge M_2 \quad M_1 M_2 \quad M_1 \vec{y} \quad \vec{y}^{\,t} M_1 \quad \Delta(\vec{y}) \quad M^*\,.$$

Among these operations the most expensive are multiplication and Kleene star. All the other operations can be performed in time $O(n^2)$. Matrix multiplication can be performed in time $O(n^3)$, and M^* can be evaluated, employing $O(\log n)$ matrix multiplications and repeated squaring, by computing $(I \vee M)^{2^k}$ for $k = 1, 2, \ldots, \lceil \log n \rceil$ [3].

Our algorithm has running time $O(mn^2 + m_0 n^3 \log n)$. When the tree depth h is small we can use a sparse matrix representation to reduce the running time. Because $\mathcal{R}[\![q]\!][u,v] \neq \bot$ only if u is an ancestor of v, and each node can have at most h ancestors, each column of $\mathcal{R}[\![q]\!]$ has at most h non-bottom elements. With a sparse matrix representation the operation cost is:

$M_1 \vee \wedge M_2$	$M_1 M_2$	$M_1 \vec{y}$	$\vec{y}^{\,t} M_1$	$\Delta(\vec{y})$	M^*
$O(nh)$	$O(nh^2)$	$O(nh)$	$O(nh)$	$O(n)$	$O(nh^2 \log n)$

Total query matching time is $O(mnh + m_0 nh^2 \log n)$. When h is small, as is typical in practice, the algorithm is approximately linear in n, the size of the tree. To summarize:

CLAIM 6.1. *With the above notation $\mathcal{R}[\![q]\!]$ can be computed in time $O(mnh + m_0 nh^2 \log n)$.*

The previous algorithm computes the complete matrix $\mathcal{R}[\![q]\!]$ for every pair of nodes. This is too much, since the final result is just the row of the matrix corresponding to the tree root. This motivates the following "lazy" approach. Given a "selector" vector $y \in \mathbf{Vec}_L(V)$ and a query q, define

$A(q,y) =_{\text{def}} (\Delta y)\mathcal{R}[\![q]\!]$. $A(q,y)$ only keeps the rows in the matrix corresponding to non-bottom elements in y. The indicator vector of a node w is $\delta_w \in \mathbf{Vec}_L(V)$:

$$\delta_w[v] = \begin{cases} \top & v = w \\ \bot & \text{otherwise} \end{cases}$$

Our goal is to compute $A(q, \delta_{\texttt{root}})$; this is done using the recursive algorithm for A given in Figure 10. In the case of a $*$ operator, a matrix is squared $\lceil \log n \rceil$ times.

$$\begin{array}{rcl} A(\mathbf{nm}, y) & = & (\Delta y)\mathcal{N}[\![\mathbf{nm}]\!] \\ A(\varepsilon, y) & = & (\Delta y) \\ A(/, y) & = & (\Delta y)C \\ A(q_1 \mid q_2, y) & = & A(q_1, y) \vee A(q_2, y) \\ A(q_1 \,\&\, q_2, y) & = & A(q_1, y) \wedge A(q_2, y) \\ A(\hat{}\, q, y) & = & \Delta(A(q, y)\top_V) \\ A(q_1\, q_2, y) & = & A(q_1, y)A(q_2, A(q_1, y)\top_V) \\ A(q*, y) & = & ((((\Delta y) \vee A(q, y))^2)^{\cdots})^2 \end{array}$$

Figure 10: Matrix-based matching algorithm.

This method computes the same result as the matrix-based algorithm, except that instead of evaluating the full matrix $\mathcal{R}[\![q]\!]$, it evaluates only the elements of the matrix that have a bearing on the final result.

THEOREM 6.2. $A(q,y) = (\Delta y)\mathcal{R}[\![q]\!]$.

PROOF. The proof is again by induction on the structure of q. We show only the most involved case, for the sequence operator: $A(q_1\, q_2, y)$. First, notice that for any matrix M we have $M = M\Delta(M\top_V)$. First, for all u, v we have $M[u,v] \leq \vee_{v'} M[u,v']$ and so $M[u,v] \leq (M\top_V)[u]$. Hence $M = M\Delta(M\top_V)$.

Next we calculate that

$$\begin{array}{lll} A(q_1\, q_2, y) & & \\ = & A(q_1, y)A(q_2, A(q_1, y)\top_V) & \\ = & A(q_1, y)\,((\Delta A(q_1, y)\top_V)\mathcal{R}[\![q_2]\!]) & \text{definition of } A \\ = & (A(q_1, y)(\Delta A(q_1, y)\top_V))\,\mathcal{R}[\![q_2]\!] & \text{associativity} \\ = & A(q_1, y)\mathcal{R}[\![q_2]\!] & \text{above observation} \\ = & (\Delta y)\mathcal{R}[\![q_1]\!]\mathcal{R}[\![q_2]\!] & \text{definition of } A \\ = & (\Delta y)\mathcal{R}[\![q]\!]\,. \;\square & \end{array}$$

The technique used by the "top-down" approach from [8] to avoid useless context computations is similar to our lazy algorithm, albeit the details are quite different.

In the worst case, the above algorithm runs in time $O(mn^3 \log n)$ and space $O(mn^2)$. Employing fast matrix multiplication we can reduce the running time to $O(mn^\omega)$ for $\omega \approx 2.38$. We can also design an $O(2^{O(m)} n \log n)$-time algorithm based on automata, which may be suitable when $m \ll n$. The classic membership problem for semi-extended regular expressions (regular expressions with intersection operation) [23, 20] can be reduced to JPath queries. The above bounds match, within a $\log n$ factor, the best known running time of any algorithm for solving this problem.

7. EXPERIMENTS

We implemented the matrix based algorithm in Section 6 and ran it on a set of JSON logs produced by a search engine (the code ran as part of the "mapper" part of a map-reduce-like computation). The complete implementation takes a few thousands of lines of C# code. Parallelization is trivial and very effective by building a separate index for each partition of the data.

Here we report the results of experiments performed on a small log fragment, processed by a single machine — representative of the work performed by one mapper. We run our experiments on a 2.33GHz quad core 64-bit Intel Xeon machine with 16G memory, running Windows Server 2008 R2, using a single enterprise-grade hard drive.

We used a sparse matrix representation for storing the matching result, and followed the algorithm given in Section 6 quite closely. We did not optimize the storing and encoding of JSON objects for fast reading or parsing: they are stored uncompressed in plain text format. The core implementation is single-threaded and it neither parallelizes computations nor tries hard to overlap I/O with computation — other than the work performed by the operating system buffer cache.

The JSON objects in the log were quite large, representing complex denormalized relations merged from a number of sources where each source conforms to a small set of different schemas. The average JSON object was 40KBytes, containing 4000 nodes. The average depth of objects was about 20, and most nodes have up to 10 children. We used a 1.6GB log (about 40K objects). We segmented the log into a small number of pieces or roughly equal size; we varied the piece size from 200MB to the whole 1.6GB, and we built a separate index tree for each piece.

We varied the indexing policies, trying different thresholds before merging nodes (e.g., a threshold of 50 means that an index tree node with less than 50 children will keep all of them, but one with more than 50 will represent all of them as a single node with a star label). Table 1 shows the various costs incurred by the index tree (averaged over all pieces; the variance is small in all cases). The indexing build time and the final index tree size is determined by the data size and is largely independent of the merging threshold, so we only show the averages for all thresholds; both are linear in the input data size. The size of the index tree does depend on the merging threshold, but in all cases the index tree itself is small. The size of the entire index tree grows linearly with the size of data due to the inverted indexes, but it is always less than 8% of the data size. The inverted indexes require most overhead, but are very compressible.

To evaluate performance we used four analytical queries obtained from production jobs running on our data-sets (the dataset is immutable during query processing). These queries vary according to their query complexity and query selectivity (highly selective queries return few results). They are shown in Table 2 (we removed the quotes for better read-

data size (MB)	200	400	800	1600
ave. build time (second)	91	200	406	831
ave. index tree size (MB)	15	30	57	111
ave. nodes (th=50)	5376	5690	6233	4550
ave. nodes (th=100)	10646	11054	12934	13723
ave. nodes (th=150)	15677	16948	18621	17618

Table 1: Indexing cost. The number of nodes in the index tree varies with the node merging threshold, but the indexing time and index tree size do not. The number of nodes is shown for 3 different merging thresholds: 50, 100, 150.

ability). A low complexity query extracts the value of an attribute close to the root while a high complexity query retrieves the value deep in the object and according to many matching criteria. The "selectivity" column indicates the percentage of objects matching each query.

We ran all queries on the full dataset, summing up the times for all pieces, so all the times can be compared (since they process the same amount of data). Figure 11 shows the breakdown of query time into four parts, starting from the bottom: time to read and query the index tree; I/O time to read the JSON trees returned by the index tree; time to parse these JSON trees; and time to run the query on these trees. A threshold of 0 indicates that no index tree is used at all, the query is just run directly on all trees. As expected, the more selective the queries, the better the speed-up from using index trees. On the other hand, non-selective queries are not significantly slowed-down.

Based on the experimental results, we make the following observations:
(1) The overhead of consulting the index tree, shown by the bottom bar, is small. Even for a non-selective query (q_1) very little is added to the total query time.
(2) Indexing is effective for selective queries (q_3, q_4) even with a moderate merging threshold. For q_3, with a merging threshold of 100 provides 2 orders of magnitude speedup.
(3) As the size of a partition grows, more false positives are returned for low merging thresholds. E.g., for q_4 with a threshold of 150, the speedup for 200M size partitions is much better than 1600M size partitions.
(4) Indexing overhead (both space and index tree query time) is slightly better amortized over large partitions.
(5) The time to parse JSON objects always dominates, but complex queries (q_2, q_4) can incur significant matching costs (top bar). So even eliminating reading and parsing time, indexing still significantly speeds up queries such as q_4.

8. JPATH AND OTHER TREE QUERY LANGUAGES

Our approach of defining a core language is similar to work on supporting subsets of XQuery/XPath [8, 1, 24, 25, 19, 14]. Particularly relevant is the work on Core XPath and its relatives [7, 8, 13, 1]. Core XPath is both similar to and different from JPath. Our queries and nodematchers correspond to XPath locpath and Boolean expressions, which are, respectively, binary and unary queries. We do not have the root predicate or the parent, following, or preceding axes. We also do not have Boolean combinations of nodematchers, but, apart from negation, we provide Boolean operations on queries. Conjunction on binary queries is an important difference from core XPath and impinges on complexity; the extent to which general conjunction is useful seems to be a practical one whose answer requires further experience. We replace the mutual dependence between locpaths and Boolean expressions by the snap operator. A pragmatic difference is our focus on binary rather than unary queries. (see the cut operator in Appendix A).

As one would expect, there are connections between JPath and logic over labeled trees [13, 9, 2]. For example, if one takes JPath less Kleene star, but adding the descendant relation (defined in JPath by `/*`) then one obtains a language with the same expressive power as binary positive queries over trees with a child and descendant relation and predicates for each nodematcher, with the evident interpretation over labeled trees.

JSON has been adopted as the data model in many NoSQL databases [4, 18, 22, 17]. Such systems store large semi-structured data sets using JSON (and other tree-like formats [15]). These systems all provide different JSON-oriented query languages, sometimes by offering JSON column types for relational tables. JPath has been designed from the outset for simplicity, orthogonality of its operators, and efficient implementation and indexing.

9. SUMMARY

We have presented JPath, a structural query language for JSON documents. We have also introduced a method for indexing JSON data stores which enables fast query executions on whole collections of documents in a single run. We have argued about the correctness of our algorithms and about the complexity of their implementations. We have also provided a mathematical treatment which unifies query processing on documents and index trees using operations on matrices with lattice-valued elements; we find that lattices provide a natural vehicle for reasoning about approximations. We believe that this unified treatment sheds some interesting new light on approximating data structures, of which our index trees are an instance.

Interestingly, none of the proofs of our theorems depend on the fact that the documents are tree shaped (but some algorithms and the implementation are designed for handling trees); the theory could as well be applied to documents structured as graphs (acyclic or cyclic). We have not exploited the full generality of our formalism in our implementations, so there is potential for some interesting developments: e.g., applying JPath to graph-shaped data structures, using strong homomorphisms to obtain exact results, using additional degrees of freedom, such as maintaining one inverted index for each label, or using more complex lattices.

Id	Complexity	Selectivity	JPath Query
q_1	low	low (100%)	`/ClientId/`
q_2	high	medium (16%)	`/Events/(^/T/Event.Impression)/DS/(^/T/D.Top.WebSet)/DS/(^/T/D.WebSet)/DS/(^/T/D.Web)/DS/((^/T/D.Url)&(^/N/Title))/K/`
q_3	medium	high (0.03%)	`^(/Events//Page/Name/Page.History)/ClientId/`
q_4	high	high (0.02%)	`/Events/(^/T/Event.Impression)/DS/(^/T/D.Top.WebSet)/DS/(^/T/D.WebSet)/DS/((^/T/D.Web)&(^/TitleMap/13))/DS/((^/T/D.Url)&(^/N/Title))/K/`

Table 2: Queries used in the experiments.

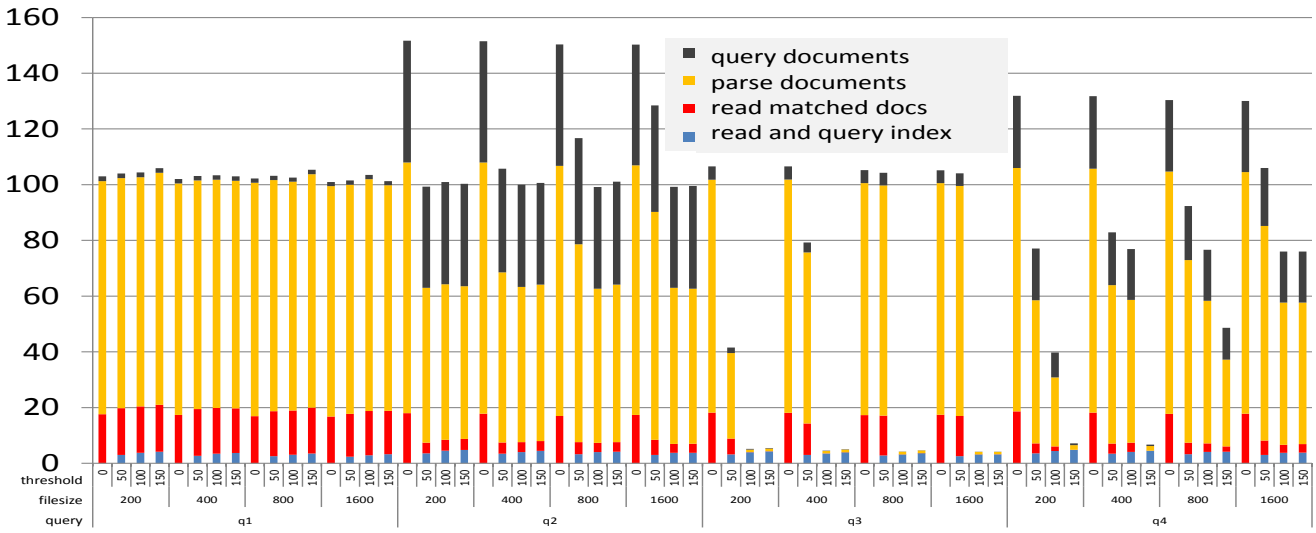


Figure 11: Query execution time (seconds) breakdown as a function of query selectivity, partition size and indexing policy. Lower is better. "Threshold" is the node merging threshold; 0 = no index used at all.

10. REFERENCES

[1] M. Benedikt and C. Koch. XPath leashed. *ACM Comput. Surv.*, 41(1), 2008.

[2] F. Bry, T. Furche, B. Linse, et al. Efficient evaluation of n-ary conjunctive queries over trees and graphs. In *WIDM*, pages 11–18, 2006.

[3] K. Cechlarova. Powers of matrices over distributive lattices – a review. *Fuzzy Sets and Systems*, 138(3):627–641, 2003.

[4] CouchDB. `http://couchdb.apache.org`.

[5] Y. Give´on. Lattice matrices. *Information and Control*, 7(4):477–484, December 1964.

[6] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructrured databases. In *VLDB*, pages 436–445, 1997.

[7] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In F. Neven, C. Beeri, and T. Milo, editors, *PODS*, pages 179–190. ACM, 2003.

[8] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[9] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, 2006.

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

[11] Json. Javascript object notation (JSON). http://www.json.org/, 2012.

[12] R. Kaushik, P. Bohannon, J. F. Naughton, et al. Covering indexes for branching path queries. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *SIGMOD Conference*, pages 133–144. ACM, 2002.

[13] C. Koch. Processing queries on tree-structured data efficiently. In *PODS*, pages 213–224, 2006.

[14] L. Libkin. Logics for unranked trees: An overview. *Logical Methods in Computer Science*, 2(3), 2006.

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

[16] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[17] MonetDB. `http://www.monetdb.org`.

[18] MongoDB. `http://www.mongodb.org`.

[19] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, et al. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD Conference*, pages 71–82, 2004.

[20] H. Petersen. The membership problem for regular expressions with intersection is complete in logcfl. In *STACS*, pages 513–522, 2002.

[21] P. Ramanan. Covering indexes for xml queries: Bisimulation - simulation = negation. In *VLDB*, pages 165–176, 2003.

[22] riak. http://wiki.basho.com.
[23] G. Rosu. An effective algorithm for the membership problem for extended regular expressions. In *FoSSaCS*, pages 332–345, 2007.
[24] Y. Wu, M. Gyssens, and J. Paredaens. A study of positive XPath with parent/child navigation. In *Principles of Database Systems (PODS)*, Vancouver, CA, 2008.
[25] Y. Wu, D. Van Gucht, M. Gyssens, et al. A study of a positive fragment of path queries: Expressiveness, normal form and minimization. *The Computer Journal*, 2010.

APPENDIX

A. PRACTICAL CONSIDERATIONS

In order to make our presentation more readable we have made some simplifying assumptions in the formalism. While building several implementations of these algorithms we have had to tackle additional issues, some of which are briefly addressed in this appendix.

Handling arbitrary JSON base types

So far we have only considered string matching for node labels. The only query operator which looks at the node labels is the *nodematcher*. In our implementations we have found it useful to add (typed and untyped) families of nodematchers to handle all legal JSON data types: strings, doubles, Booleans and null values.

We have also added type-checking nodematchers such as `IsString`, `IsBoolean`, `IsNumeric` which return 'true' only if the checked node label has the proper type, and also `IsArray`, `IsObject` and `IsLeaf` nodematchers. The query matching algorithm remains unchanged, as it only depends on the results produced by nodematchers, which are still lattice values.

Writing complex nodematchers

As we said in Section 3, nodematchers can be used to perform more complex computations on node labels (not just testing for equality). In fact, one can use any predicate on node labels, including regular expressions, arithmetic or string manipulations. These changes have no consequences for the query matching algorithm.

JavaScript is a natural language choice for writing nodematchers. Nodematchers can be written as anonymous JavaScript functions taking a single argument, the node label, and returning a Boolean value.

Here are some possible examples of useful nodematcher functions, written in JavaScript:

- `function(label){label == "price";}` — matches exactly the `price` label.
- `function(label){label.length == 5;}` — matches all labels with length 5
- `function(label){label.indexOf("price") >= 0;}` — matches all labels that contain the `price` sub-string
- `function(label){label % 2 == 0;}` — matches all even numeric labels.
- `function(label){label.test(/^[A-Za-z0-9]+$/);}` — matches all alphanumeric labels.

The syntax for writing such nodematchers we have chosen in our implementation requires supplying the JavaScript code in a properly quoted string, which unfortunately requires escaping the nested quotes. Here is an example query consisting of just one nodematcher:

```
JavaScript("function(l) { l == \"price\"; }")
```

Running arbitrary user-supplied code as part of the query execution on the server side is a difficult problem, which we

do not address in this document; JavaScript sandboxing offers security, and timers are used to abort long-running computations.

The "cut" operator

The query language presented so far only computes one bit per document, returning the whole document if it matches the query, or nothing if it does not. The queries may perform a lot of work to discover the document structure, but this information is lost. We have extended our query language with an additional operator called "cut", denoted by `!`, which allows efficient extraction of document fragments.

For example, we would like to extract the `"city"`'s of the `"exports"` fields of the two objects in Example 1. User queries must always contain exactly one cut operator:

$$userquery = query\,!\,query.$$

While matching a query on a tree or index tree t the cut operator is treated like a child operator `/`. However, after matching has been performed, the cut operator executed on a document tree is used to prune t, generating a set of its subtrees.

Let us consider the query $q = q_1\,!\,q_2$ applied to a tree t. Suppose that $\mathcal{T}_t\,[\![q_1]\!] = \{u_1, \ldots, u_k\}$, and that for each u_i we have $\mathcal{T}_t\,[\![q_2, u_i]\!] = \{v_{i1}, v_{i2}, \ldots, v_{im}\}$. The result returned to the user after running q on t is the set T' of proper sub-trees of t rooted at some u_i and spanning all the nodes in $\mathcal{T}_t\,[\![q_2, u_i]\!]$, all the way down to the leaves. Figure 12 depicts the cut operation. Note that the trees in T' overlap with each other, if some of the u nodes are descendants of others.

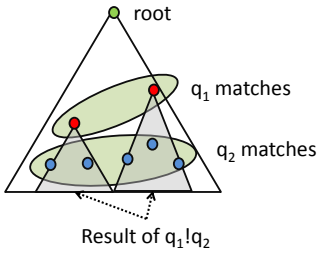


Figure 12: The "cut" operator: the results of the query $q_1\,!\,q_2$ are the sub-trees of t rooted at the results of q_1 spanned by the nodes matched by q_1/q_2.

To solve the problem posed above, of extracting just the `city`s, we would write the following query: `/"exports"/?!"city"`. The result of this query is the following set of documents: {`"city":"Moscow"`}, {`"city":"Athens"`}, {`"city":"Berlin"`}, and {`"city":"Amsterdam"`}. Notice that the result does not contain the node `"dealers"` and its children from the second document, because it is not spanned by the `"city"` subquery.

Recall from Section 2 that we eliminate arrays by transforming them into objects. If an array is entirely returned as part of the result, we perform the reverse conversion before returning the result to the user. E.g., the result of the query `^(/"headquarters"/"Belgium")/"location"/!()` is an array:

```
[  { "country": "Germany", "city": "Berlin" },
   { "country": "France", "city": "Paris" }  ]
```

However, the result of the query `^(/"headquarters"/"Belgium")/"location"!1` is

```
{ "1":{ "country":"France", "city":"Paris" }}
```

Because the latter query only returns a fragment of the array, the result is converted to an object, the label 1 being converted to a string to obtain a correct JavaScript object.

Lazy lattice computations

The algorithm in Section 6 performs a computation on matrices with lattice values for each kind of query constructor; query evaluation turns into matrix computations. The matrix computation should be performed *lazily*: instead of performing matrix operations right away, the algorithm builds an *expression tree* describing the lattice computations to be performed. Optimizations can be performed on this tree (e.g., common sub-expression elimination), and the expression is evaluated only after the complete query has been executed on an index tree. The evaluation of the expression takes advantage of lattice properties to avoid unnecessary computations. For example, as $\bot \wedge e = \bot$, we do not need to compute the value of e at all. The lazy lattice computations are orthogonal to the lazy computation of matrix elements from Section 6, and can be used in combination.

Mutable databases

The discussion so far did not address the issue of dynamic index maintenance when documents are inserted or deleted from the database. One of our implementations incrementally updates the index when deletions, insertions and updates occur in the database. The cost of incremental index updates is significant, e.g., the number of inverted indexes updated is proportional to the number of nodes in an inserted tree. We use several techniques to amortize the cost of updates:

- We treat deletions lazily, by marking documents deleted, and not updating the index. A background garbage-collection process trims the unneeded nodes.
- We batch insertions, by inserting multiple documents in a single traversal of the index. This means that the new documents are inserted into the index without any node merging, and that merging is performed only periodically.
- We split the index into a set of disjoint index sub-trees, each tree handing a subset of the documents; the complete index is the union of these sub-trees. We maintain the invariant that all insertions are performed in the last tree only. This makes the first n-1 trees read-only (in fact, delete-only). Queries are run against all the sub-trees, and the complete result is the union of the results. The last tree grows until it reaches a maximum size, at

which point it is "sealed", and a new final tree is created.

Paginated queries

Some queries can return a large number of results; since it is not practical to return an arbitrary number of documents in a single service request, in practice query execution is paginated: the execution of a query returns a set of documents and an opaque continuation token. A subsequent query that supplies the continuation token will continue returning additional results starting where the first query left off. (The token encodes the state of a server-side iterator over query results.)

Paginated queries may conflict with concurrent updates, performed while the query is executing. The semantics of our query execution guarantees that all documents that are not mutated (inserted/deleted) while the query runs will be matched during the query execution; the result is unspecified for documents that are updated concurrently with the query execution.

The design described using a index composed of multiple index trees, is quite suitable for handling the problem of concurrent updates as well. The query is run starting from the last index tree (the only one mutable), going towards the first index tree. The first n-1 index trees are essentially read-only, so running the query on them creates no conflicts with updates.

B. ALGORITHM BUILDTREE

This recursive algorithm in Figure 13 translates a JSON document in its tree representation. The algorithm is invoked with BUILDTREE(jsonTree, empty).

```
function BUILDTREE(ParseTree jsonDoc, BaseValue rootLabel)
    if jsonDoc is basevalue then
        parent = new Node
        parent.type = NodeTypeLeafParent
        label(parent) = rootLabel
        child = new Node
        child.type = NodeTypeLeaf
        label(child) = jsonDoc
        parent.children.Add(child)
    else if jsonDoc is object then
        parent = new Node
        parent.type = NodeTypeObject
        label(parent) = rootLabel
        for all pair in jsonDoc.keyValuePairs do
            child = BUILDTREE(pair.value, pair.label)
            parent.children.Add(child)
        end for
    else if jsonDoc is array then
        parent = new Node
        parent.type = NodeTypeArray
        label(parent) = rootLabel
        for i=0..(jsonDoc.elements.Length-1) do
            child = BUILDTREE(jsonDoc.elements[i], i)
            parent.children.Add(child)
        end for
    end if
    return parent
end function
```

Figure 13: Algorithm for transforming a JSON document to a labeled tree.

C. QUERY EXAMPLES

Here are the results of applying some queries to a document collection containing the two documents from Figure 1.

• **Query:** `/"exports"//"city"/`
Results: `"Moscow"`, `"Athens"`, `"Berlin"`, `"Amsterdam"`
Explanation: `"exports"` is a constant-string nodematcher, which only matches nodes labeled with the string `exports`. The `/` operator is used to extract the children of a node.

• **Query:** `(/*)/"country"`
Results: {`"country":"Germany"`}, {`"country":"France"`}, {`"country":"Germany"`}
Explanation: The star operator is used to traverse the trees to an arbitrary depth to discover all `"country"` labels. Within the set of three results, there are two which are identical, the first coming from the first document, and the third coming from the second document.

• **Query:** `(^/"location"/"country"/"France")/"headquarters"/`
Results: `"Belgium"`
Explanation: The query finds all documents which have `"location"` with a `"France" "country"`, then inquires about their `"headquarters"`. The `^` operator is used to return back in the JSON tree to the node that had the children; the `/"headquarters"` part of the query is applied at that point.

• **Query:** `(^/"location"/((^"country"/"France") & ("country"/"Germany")))/"headquarters"`
Results: {`"headquarters":"Belgium"`}
Explanation: Finds the `"headquarters"` of all documents who have `"location"` in both `"France"` and `"Germany"`. The `&` operator is used to perform a conjunction, and the `^` is used twice: the first instance to return to the parent of the node labeled `"location"` to search for the `"headquarters"`, and the second instance is used to ensure that the path `"country"/"Germany"` is sought at the same node where the path `"country"/"France"` had matched.

D. PROOF OF THEOREM 4.1

Let us fix $t = t_i$ and $v \in V(t)$. The proof by induction on the structure of the query is shown in Figure 14.

E. PROOF OF THEOREM 5.1

$q = \mathbf{nm}$ Follows from the label consistency condition.

$q = \varepsilon$ Obvious.

$q = \texttt{/}$ Follows from the child consistency condition.

$q = q_1 \ q_2$

$$\begin{array}{lll} M(\mathcal{T}[\![q, v]\!]) & & \\ = & M(\cup_{u \in \mathcal{T}[\![q_1, v]\!]} \mathcal{T}[\![q_2, u]\!]) & \text{def} \\ = & \cup_{u \in \mathcal{T}[\![q_1, v]\!]} M(\mathcal{T}[\![q_2, u]\!]) & \\ \subseteq & \cup_{u \in \mathcal{T}[\![q_1, v]\!]} \mathcal{I}[\![q_2, M(u)]\!] & \text{IH} \\ \subseteq & \cup_{u' \in \mathcal{I}[\![q_1, M(v)]\!]} \mathcal{I}[\![q_2, M(u')]\!] & \text{IH} \\ = & \mathcal{I}[\![q, M(v)]\!] & \text{def} \end{array}$$

$q = q_1 \texttt{ | } q_2$ From IH and monotonicity of set union.

$q = q_1 \texttt{ \& } q_2$ From IH and monotonicity of set intersection.

$q = q_1\texttt{*}$: We just need to prove that for any k-bounded approximation of the query result: $\cup_{0 \leq n \leq k} \mathcal{T}[\![q_1^n, v]\!]$ this property holds, where $k \leq |V(\sigma)|$. The case $k = 0$ is the same as the case for $q = \varepsilon$ above. For $k = 1$ the statement follows from the induction hypothesis. Given a proof for k we can prove the statement for $k+1$ using the sequence operator proof above, and the fact that union is monotone.

$q = \texttt{\^{}}q_1$ We have two cases:

- $\mathcal{T}[\![q_1, v]\!] = \emptyset$. The result is trivial, since $M(\mathcal{T}[\![q, v]\!]) = \emptyset$.
- $\mathcal{T}[\![q_1, v]\!] \neq \emptyset$. By the induction hypothesis, $M(\mathcal{T}[\![q_1, v]\!]) \subseteq \mathcal{I}[\![q_1, M(v)]\!]$. Thus we must have that $\mathcal{I}[\![\texttt{\^{}}q_1, M(v)]\!] = \mathcal{I}[\![q, M(v)]\!] = M(v)$, by the definition of the $\texttt{\^{}}$ operator. But, for the $\texttt{\^{}}$ operator, $\mathcal{T}[\![q, v]\!] \subseteq \{v\}$, and thus, applying M on both sides, $M(\mathcal{T}[\![q, v]\!]) \subseteq M(v) = \mathcal{I}[\![q, M(v)]\!]$.

Figure 14: Proof of Theorem 4.1.

LEMMA E.1. *For any $\vec{y} \in \mathbf{Vec}_L(V')$, and for any matrix $H \in \mathbf{Mat}_L(V, V')$ corresponding to a subhomomorphism h we have:*

$$\Delta(H\vec{y})H = H\Delta(\vec{y})$$

PROOF. A simple calculation. □

The proof of Theorem 5.1 is also by induction on the structure of q, shown in Figure 15.

$$\begin{array}{lll} \mathcal{R}_\tau[\![\mathbf{nm}]\!]H & = & \Delta(\texttt{label}[\![\mathbf{nm}]\!])H \\ & \leq & \Delta(H\texttt{label}'[\![\mathbf{nm}]\!])H \quad \text{(label consistency)} \\ & = & H\Delta(\texttt{label}'[\![\mathbf{nm}]\!]) \quad \text{(by Lemma E.1)} \\ & = & H\mathcal{R}_{\tau'}[\![\mathbf{nm}]\!] \end{array}$$

$$\begin{array}{lll} \mathcal{R}_\tau[\![\varepsilon]\!]H & = & \mathrm{I}_V H \\ & = & H\mathrm{I}_{V'} \\ & = & H\mathcal{R}_{\tau'}[\![\varepsilon]\!] \end{array}$$

$$\begin{array}{lll} \mathcal{R}_\tau[\![\texttt{/}]\!]H & = & CH \\ & \leq & HC' \quad \text{(child consistency)} \\ & = & H\mathcal{R}_{\tau'}[\![\texttt{/}]\!] \end{array}$$

$$\begin{array}{lll} \mathcal{R}_\tau[\![q_1 q_2]\!]H & = & \mathcal{R}_\tau[\![q_2]\!]\mathcal{R}_\tau[\![q_1]\!]H \\ & \leq & \mathcal{R}_\tau[\![q_2]\!]H\mathcal{R}_{\tau'}[\![q_1]\!] \quad \text{(by IH)} \\ & \leq & H\mathcal{R}_{\tau'}[\![q_2]\!]\mathcal{R}_{\tau'}[\![q_1]\!] \quad \text{(by IH)} \\ & = & H\mathcal{R}_{\tau'}[\![q_1 q_2]\!] \end{array}$$

$$\begin{array}{lll} \mathcal{R}_\tau[\![\texttt{\^{}}q]\!]H & = & \Delta(\mathcal{R}_\tau[\![q]\!]\top_V)H \\ & = & \Delta(\mathcal{R}_\tau[\![q]\!]H\top_{V'})H \\ & \leq & \Delta(H\mathcal{R}_{\tau'}[\![q]\!]\top_{V'})H \quad \text{(IH)} \\ & = & H\Delta(\mathcal{R}_{\tau'}[\![q]\!]\top_{V'}) \quad \text{(by Lemma E.1)} \\ & = & H\mathcal{R}_{\tau'}[\![\texttt{\^{}}q]\!] \end{array}$$

$$\begin{array}{lll} \mathcal{R}_\tau[\![q_1 \texttt{ | } q_2]\!]H & = & (\mathcal{R}_\tau[\![q_1]\!] \vee \mathcal{R}_\tau[\![q_2]\!])H \\ & = & \mathcal{R}_\tau[\![q_1]\!]H \vee \mathcal{R}_\tau[\![q_2]\!]H \\ & \leq & H\mathcal{R}_{\tau'}[\![q_1]\!] \vee H\mathcal{R}_{\tau'}[\![q_2]\!] \quad \text{(IH)} \\ & = & H(\mathcal{R}_{\tau'}[\![q_1]\!] \vee \mathcal{R}_{\tau'}[\![q_2]\!]) \\ & = & H\mathcal{R}_{\tau'}[\![q_1 \texttt{ | } q_2]\!] \end{array}$$

$$\begin{array}{lll} \mathcal{R}_\tau[\![q_1 \texttt{ \& } q_2]\!]H & = & (\mathcal{R}_\tau[\![q_1]\!] \wedge \mathcal{R}_\tau[\![q_2]\!])H \\ & = & \mathcal{R}_\tau[\![q_1]\!]H \wedge \mathcal{R}_\tau[\![q_2]\!]H \quad (H \text{ is function matrix}) \\ & \leq & H\mathcal{R}_{\tau'}[\![q_1]\!] \wedge H\mathcal{R}_{\tau'}[\![q_2]\!] \quad \text{(IH)} \\ & = & H\mathcal{R}_{\tau'}[\![q_1 \texttt{ \& } q_2]\!] \quad (H \text{ is function matrix}) \end{array}$$

$$\begin{array}{lll} \mathcal{R}_\tau[\![q\texttt{*}]\!]H & = & (\bigvee_{n \geq 0} \mathcal{R}_\tau[\![q]\!]^n)H \\ & = & \bigvee_{n \geq 0} \mathcal{R}_\tau[\![q]\!]^n H \\ & \leq & \bigvee_{n \geq 0} H\mathcal{R}_{\tau'}[\![q]\!]^n \quad \text{(IH)} \\ & = & H(\bigvee_{n \geq 0} \mathcal{R}_{\tau'}[\![q]\!]^n) \\ & = & H\mathcal{R}_{\tau'}[\![q\texttt{*}]\!] \end{array}$$

Figure 15: Proof of Theorem 5.1.