Grammar-aware Parallelization for Scalable XPath Querying

Lin Jiang, Zhijia Zhao
University of California, Riverside
ljian006@ucr.edu, zhijia@cs.ucr.edu

Abstract

Semi-structured data emerge in many domains, especially in web analytics and business intelligence. However, querying such data is inherently sequential due to the nested structure of input data. Existing solutions pessimistically enumerate all execution paths to circumvent dependencies, yielding sub-optimal performance and limited scalability.

This paper presents GAP, a parallelization scheme that, for the first time, leverages the grammar of the input data to boost the parallelization efficiency. GAP leverages static analysis to infer feasible execution paths for specific contexts based on the grammar of the semi-structured data. It can eliminate unnecessary paths without compromising the correctness. In the absence of a pre-defined grammar, GAP switches into a speculative execution mode and takes potentially incomplete grammar extracted either from prior inputs. Together, the dual-mode GAP reduces the execution paths from all paths to a minimum, therefore maximizing the parallelization efficiency and scalability. The benefits of path elimination go beyond reducing extra computation – it also enables the use of more efficient data structures, which further improves the efficiency. An evaluation on a large set of standard benchmarks with diverse queries shows that GAP yields significant efficiency increase and boosts the speedup of the state-of-the-art from 2.9X to 17.6X on a 20-core machine for a set of 200 queries.

Keywords XML, XPath, Grammar, Parallelization

1. Introduction

Semi-structured data, like XML and JSON, is widely used in many application domains, such as web analytics [42], financial data processing [31], pub/sub applications [27], enterprise data exchanges [8], sensor networks [12], and smart buildings [38]. The volume of semi-structured data grows rapidly. Take Twitter as an example, it produces tweets in semi-structured format at a rate of 600 million per day [20].

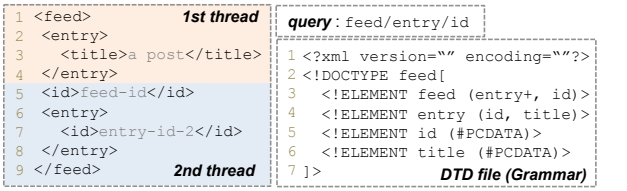

Figure 1. An illustration example (Facebook feed data [1])

To cope with fast data growth, users need efficient querying methods to extract information.

In comparison, the increase of modern CPU frequency has reached a plateau. Hence, exploiting parallelism is key to efficient query processing for semi-structured data. In this work, we focus on XPath queries for their basic roles in semi-structured data processing [1, 18, 23, 30].

However, parallelizing such queries is challenging due to the inherent nested structure of input data. Consider Facebook feed data [13] in Figure 1. Suppose using two threads to process a path query `feed/entry/id`, which aims to find the IDs of entries in the feed. Without examining the tags in the first half of the input, the second thread cannot determine if the `id` at line 5 is a match or not. Because it lacks the "context" of determining the answers to the query. As we will see later, such inherent dependence clearly manifests when the queries are formalized as *pushdown transducers* – a basic computation model for processing data that are defined by context-free grammars (Section 2).

State of The Art. Great efforts have been made on making parallel parsing possible for semi-structured data. With the parsing results, a DOM (Document Object Model) tree, the queries can be easily evaluated by traversing the tree. A practical issue with such methods is that the data could be too large to parse due to the huge memory consumption caused by constructing the parse tree. In addition, the query evaluation after parsing compromises data locality.

In comparison, Odgen and others [30] directly target the parallelization of XPath queries with a novel parallel pushdown transducer. By breaking the input into chunks and enumerating all execution paths for each chunk (except the first one), pushdown transducers are able to process different chunks in parallel. This method shows promising results for single-query processing and processing a small set of con-

[1] For illustration purpose, the data is simplified and slightly reordered.

PPoPP '17 Feb. 4–8, 2017, Austin, Texas, USA.

DOI: http://dx.doi.org/10.1145/3018743.3018772

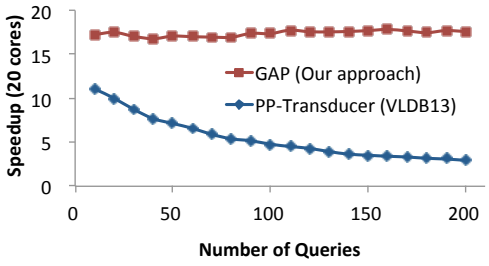

Figure 2. Scalability Comparison

current queries. However, as the query becomes more complex or the number of concurrent queries increases, the total number of execution paths also raises, resulting in a dramatic increase of parallelization cost. Consequently, the speedup quickly drops from 11.1X to merely 2.9X on a 20-core Intel Xeon server, as shown in Figure 2.

In this work, we raise and address some fundamental questions in the parallelization of pushdown transducers: *is it necessary to enumerate all execution paths? If not, how to know which execution paths are unnecessary?*

The key to answering the questions lies in the special properties of input data – *they are defined by (context-free) grammar, either explicitly or implicitly*. For example, XML data are defined by document type definition (DTD) or XML schema. Similar ways are used to define JSON data as well [15]. The grammar not only defines the valid structure of the data, but also adds constraints on the possible execution paths under a specific context. Consider the example in Figure 1. By checking the first element `<id>`, the second thread would be able to infer that it may be under the `<feed>` or an `<entry>`, but definitely not under a `<title>`, as the grammar does not specify an ID for the title.

To leverage these insights, we propose a *grammar-aware parallelization (GAP)* for querying semi-structured data, which can substantially reduce the amount of execution paths up to 200X according to our experiments. It brings in two major benefits. First, a thread only needs to keep a small number of execution paths (typically < 5), hence runs more swiftly. Second, when the number of execution paths drops to one, which happens quite often based on our experiments, it becomes possible to switch to more efficient data structures (from trees to stacks) to execute the pushdown transducers. Together, they dramatically reduce the parallelization cost, making efficient parallelization of larger-scale query processing possible, as shown in Figure 2.

Specifically, depending on the availability of an explicit grammar, GAP works in two modes: non-speculative mode and speculative mode. In non-speculative mode (Section 4), GAP extracts a static syntax tree from the given grammar file (e.g., a DTD file), and symbolically executes the pushdown transducer over the syntax tree to infer feasible execution paths under a specific context (e.g., a token `<id>`). With such knowledge, a parallel pushdown transducer only maintains a set of feasible execution paths and automatically switches to a stack when it detects a unique feasible execution path is left. In certain scenarios, an explicit grammar may not be available. In such cases, GAP enters into the speculative mode, where it first collects some partial grammar from prior runs. As the grammar is partial, the inferred feasible paths might be incomplete. Hence, a pushdown transducer runs speculatively with chances that the correct execution is left uncovered. To guarantee the correctness and reduce the penalty of misspeculation, a pair of validation and selective reprocessing mechanisms are proposed. With the dual-mode GAP, our approach is able to process semi-structured data either with or without a grammar file. Our evaluation results show that both non-speculative GAP and speculative GAP provide sustainable speedup, about 17.6X on a 20-core server, up to a set of 200 concurrent queries.

Contributions. This work makes several contributions:

- To our best knowledge, this work, for the first time, unveils the potential of leveraging input grammar to improve the efficiency of parallelization.
- It offers a rigorous inference of feasible execution paths from input grammar and an adaptive data structure switching mechanism to boost the efficiency of the basic computation model – pushdown transducers.
- It proposes a practical speculation scheme to cover the scenarios where the input grammar is unavailable.
- It, for the first time, enables a scalable parallelization of query processing that yields sustainable speedup as the query complexity increases.

In the following, we will first give the background and motivation (Section 2), then present an overview of GAP (Section 3), followed by the details of GAP in non-speculative mode (Section 4) and speculative mode (Section 5). We show our evaluation results in Section 6, related work in Section 7, and conclusion of this work in Section 8.

2. Background

In this section, we first present some basic concepts for querying semi-structured data. Then we introduce the computation model for processing queries – *pushdown transducer* and its parallelization challenges.

2.1 Querying Semi-Structured Data

Querying semi-structured data is a routine operation in stream processing and many database systems that supports semi-structured data. For instance, YFilter [10] and XMLTK [2] leverage automata to simultaneously process a large number of XPath queries against an XML stream with a constant memory requirement. These studies focus on improving the expressiveness of queries, instead of exploiting data parallelism. On the other hand, database engines, like Microsoft SQL Server [32] and MonetDB [3] provide task-level parallelization for querying XML data, but require pre-computed index to accelerate query processing, which

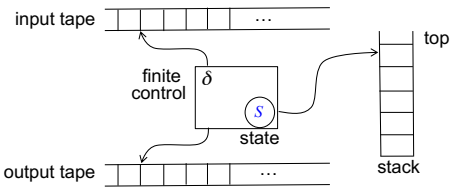

Figure 3. A Diagram of The Pushdown Transducer

becomes unsuitable for single-pass processing model due to the substantial cost of index construction.

Pre-parsing v.s. On-the-fly Querying. There are two basic strategies to process queries: (i) first parse the semi-structure data into a parse tree (i.e., DOM tree in XML context), then answer the queries by searching the tree. (ii) Process the queries on-the-fly without constructing any tree structure. There are three limitations with the first strategy. First, parsing the semi-structured data requires a large memory footprint due to the construction of DOM tree, which is almost infeasible for large semi-structured data or streaming data. Second, parallel parsing itself is quite challenging, existing solutions suffer from either load imbalance or sequential bottleneck caused by a sequential preprocessing. At last, it needs to traverse the data again after the parsing, compromising the data locality. For the above reasons, we choose the second strategy in this work. To enable on-the-fly query processing, queries are implemented using stack-based computation model – pushdown transducers [18, 30], which run over the semi-structured data stream and report matches as it proceeds. We next describe such a computation model.

2.2 Pushdown Transducers

Informally, a pushdown transducer is a finite automaton agumented with a stack and an output tape, as illustrated by Figure 3. With the stack, the pushdown transducer is able to memorize prior states (i.e., history). Such memorization is the essential for processing semi-structured data where certain "context" is needed. A pushdown transducer consumes the input stream by moving a pointer forward along the input tape one by one. Based on the input symbol, the control logic (defined as transitions) examines the current state and the status of the stack, then makes adjustments to the state and stack, and write symbols to the output tape.

A generic pushdown transducer can be defined as follows.

Definition 1. *A pushdown transducer is a 6-tuple (Σ, Γ, Δ, Q, q_0, δ) where Σ is the input alphabet, Q is the set of states, $q_0 \in Q$ is the initial state, Γ is the stack alphabet – a set of symbols that can be pushed onto stack, Δ is the output alphabet, and δ is a mapping $\delta : Q \times \Gamma \times \Sigma \rightarrow Q \times \Gamma \times \Delta$ which is also called the transition function.*

There are three factors that may affect the moving of a pushdown transducer: (i) current input symbol $c \in \Sigma$ from the input tape, (ii) current state $s \in Q$, and (iii) the element $e \in \Gamma$ on top of the stack. As a response, the pushdown transducer may take three actions: (i) move the current state to the next, (ii) update the top of the stack either by `push` or `pop`, and (iii) write a symbol $c' \in \Delta$ to the output tape. Note that, depending on the definition of transitions, the three factors may not always take effects at every transition, and the same is true for the actions.

We next describe the *specific pushdown transducers* for query processing using the running example in Figure 4. It contains a grammar with recursion (between elements `a` and `b`) to demonstrate its generality.

States Q. For any give path query, a corresponding finite automaton can be created based on a simple automaton construction algorithm [18]. The states in the finite automaton are exactly the states in the pushdown transducer. The accept state(s) correspond to the matches of the query. In the running example, a path query `a/b/c` is converted to a finite automaton of five states, as shown in Figure 4-c. `state 4` corresponds to the match of query.

Alphabets Σ **,** Δ **,** ***and*** Γ. In the context of XPath querying, the input alphabet Σ is the set of valid tokens (e.g., XML tags) resulted from a lexer. The output alphabet is actually the same as the input alphabet (i.e., $\Delta = \Sigma$) since the queries only return some parts of the input data stream. As to the stack alphabet, it can be either the state set Q or input alphabet Σ. Based on the convention [30], we set $\Gamma = Q$.

Transitions δ_{plain} **,** δ_{push} **,** ***and*** δ_{pop}. There are three types of transitions in XPath querying pushdown transducers.

1. Plain transitions δ_{plain}: $Q \times \Sigma \rightarrow Q \times \Delta$. They do not influence the stack. For example, the transition after reading a text token `x` is a plain transition.
2. Push transitions δ_{push}: $Q \times \Sigma \rightarrow Q \times \Gamma \times \Delta$. When an start tag (e.g., `<a>`) is read, the transducer first pushes the current state onto the stack, then transition to the next state based on the finite automaton.
3. Pop transitions δ_{pop}: $Q \times \Sigma \times \Gamma \rightarrow Q \times \Delta$. When a close tag (e.g., `</d>`) is met, the transducer pops the stack and sets the current state to the popped state.

Example. Figure 4-d illustrates the transitions of the pushdown transducer for processing query `a/b/a/c`. Among the ten steps, five of them are push transitions and the others are pop transitions. When the pushdown transducer moves to the accept state (`state 5`), it reports a match to the query and outputs the match to its output tape.

Inherent Dependence. Based on the definition of pushdown transducers, it is obvious to notice two types of dependence in its execution depending on the variables: (i) At every step of the transition, the next state depends on either the immediate prior state or the element on top the stack (which is actually an even earlier state). We refer to such dependence as *state dependence*. (ii) Similarly, at each step of the transition, the content of stack depends on stack before the transition and sometimes also depends on the prior state. We refer to this type of dependence as *stack dependence*. In general, a tight dependence chain is formed from the first step to the

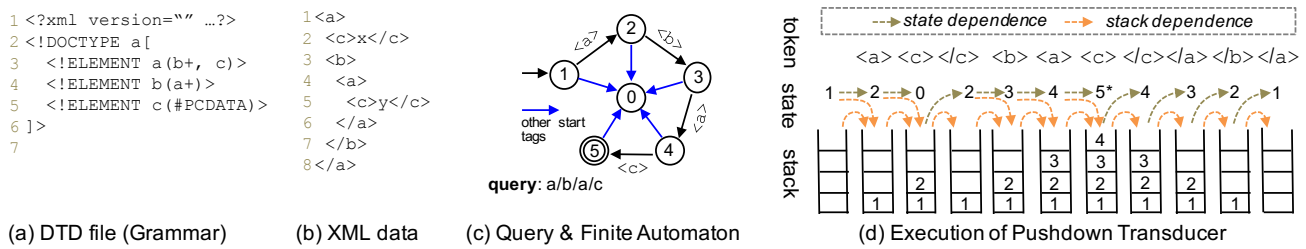

Figure 4. A running example with a recursive grammar. '*' after state 5 means a match.

current step, making the parallelization of such computation model quite challenging, as illustrated by Figure 4-d.

2.3 Parallel Pushdown Transducers

To circumvent the inherent dependence in pushdown transducers, Ogden and others [30] propose a parallel pushdown transducer. Basically, it works in three major phases:

- *Split phase*. It first cuts the input data into equal-sized chunks, each of which may contain some broken tags.
- *Parallel phase*. For each chunk, except the first one, it enumerates all execution paths, each starting from a different state with an empty stack. A mapping m is maintained from each pair of starting state and stack (q_s, z_s) to a 3-tuple of finishing state, stack and output tape (q_f, z_f, o), that is $m = (q_s, z_s, q_f, z_f, o)$, where $m \in M = Q \times \Gamma^* \times Q \times \Gamma^* \times \Delta^*$. In our running example (Figure 4), it needs to enumerate all the six states (paths) and keeps a mapping for each of them.

  During the processing of a chunk, different execution paths may lead to the same finishing state and stack (called *path convergence*). To improve the efficiency, a double-tree data structure is employed to compress the mapping and reduce some redundant computations when path convergence occurs.

  For a chunk with missing start tags (due to partitioning), it is possible that a pop operation is needed while the stack is empty. In this case, it also has to enumerate every possible element that could be popped out from the stack (i.e. Γ). In such cases, a single path diverges to multiple ones, referred to as *path divergence*. In our running example, if the second thread starts from Line 5, when it processes the end tag `</a>` at Line 6, it encounters a path divergence, similarly to the end tags `</b>` and `</a>` at Lines 7 and 8.
- *Join phase*. Finally, mappings from different chunks are linked by matching the finishing state and stack in a chunk with the starting state and stack of its following.

An additional *filtering* phase may be added to enhance the expressiveness of the transducers (e.g., to handle predicates in XPath queries). Though the split, join and filter phases are sequential, they carry much less computations than the parallel phase and incur marginal cost [30].

Parallel pushdown transducers show promising results for single query or a small set of concurrent queries. However, as the query becomes more complex and as the number of queries increases, enumerating all execution paths tends to be less practical due to large amount of extra computation. To address this question, we propose a new scheme of parallelization, *grammar-aware parallelization (GAP)*, that leverages the grammar of the semi-structured data to guide the design of parallelization. In the following, we first give an overview of GAP before elaborating its major components.

3. Overview of GAP

In this section, we give the high-level design of GAP, a new parallelization scheme that leverages input grammar to boost the efficiency of parallelization.

Grammar Availability and Dual-mode GAP. The key of GAP is the awareness of semi-structured data grammars. A substantial amount of semi-structured data come with grammars, such as the auction data from ebay developer API [9] and the publication data from DBLP [7]. Actually, six out of ten datasets from UW's XML data repository [40] contain some DTD file(s). On the other hand, it is not uncommon that the grammar of some semi-structured data is unavailable, either because there does not exist a pre-defined grammar, or because the access to the DTD or XSD is not granted.

To cover both scenarios, GAP can run in two modes: *non-speculative mode* and *speculative mode*. When the grammar of the semi-structured data is available (e.g., in the form of DTD), GAP enters into non-speculative mode, where the execution correctness is always guaranteed, thanks to the rigorous feasible path inference from a complete grammar (as explained shortly). In another word, there is no speculation or prediction involved in this mode. On the other hand, when the grammar is unavailable, GAP switches into speculative mode. In this mode, GAP first collects some partial grammar by inferring it from previous runs (of the same data corpus). This part is illustrated as the first step of GAP in Figure 5.

Feasible Path Inference. After determining the execution mode, GAP obtains either a complete grammar or a partial grammar. With the grammar, GAP aims to infer which paths are feasible given a specific context, that is, a symbol from the input alphabet Q (e.g., a tag `<c>` in Figure 4-b).

To achieve this, GAP follows two steps: *static parse tree generation* and *symbolic execution of pushdown transducer*.

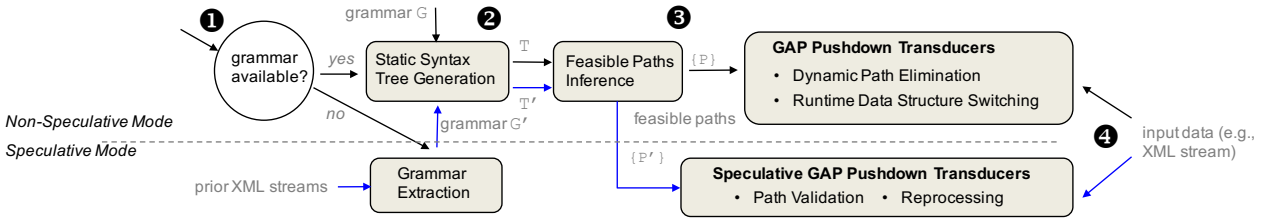

Figure 5. Overview of GAP. Labeled numbers show the major steps when predefined input grammar is available (e.g., DTD or Schema). Blue lines show the speculative execution flow in the absence of a pre-defined grammar.

In the first step, it takes the input grammar G as input and generates a static syntax tree T, which embodies the legal nesting structures of the input data. For example, element `b` has to be an immediate child of element `a` (see Figures 4-a and 4-b). In the second step, it executes the pushdown transducer symbolically by executing on every path of the static syntax tree to infer the feasible starting states when meeting a specific input symbol (i.e., the context). The results of the second step is a hash table, called *feasible path table*, where the key is input symbol and the value is the set of feasible paths, as shown in Table 1. Since the syntax tree is static and

Input symbol	Feasible paths/states
<a>	{1, 3}
	{2, 4}
<b>	{2}
...	...

Table 1. Feasible Paths Table

typically small, a full traversal is affordable. When the grammar is available beforehand, both steps can be done offline.

GAP Pushdown Transducers. To leverage the results from feasible path inference, we design a new type of parallel pushdown transducer – *GAP pushdown transducers*. First, when it is needed, a GAP pushdown transducer can inquire the feasible path table to determine and remove unnecessary paths, which is referred to as *dynamic path elimination*.

Second, whenever a unique feasible path is left, either caused by path elimination or path convergence, the GAP pushdown transducer switches from the default double-tree data structure to a stack and executes exactly like a sequential pushdown transducer. This makes the GAP pushdown transducer run swiftly without any house-keeping work. This feature is referred to as *runtime data structure switching*.

Speculative GAP. As explained earlier in this section, when a pre-defined grammar is unavailable, only a partial grammar might be collected. Inferring feasible paths from such an incomplete grammar may result in some feasible paths missing. Consequently, feeding such incomplete information to a GAP pushdown transducer may cause the correct execution path being excluded. This scheme is generally referred to as *speculative execution*. There are two basic requirements for effective speculative execution. First, it has to ensure the correctness. GAP pushdown transducers ensure this in the join phase. When no match of mappings is found during the linking of two consecutive chunks, a misspeculation is reported. Under this situation, a reprocessing is initiated immediately on the missepculated chunk to ensure the correctness. Second, the misspeculation penalty should be low enough so that it will not cancel out the benefits of speculation. To achieve this, GAP pushdown transducers carefully divide chunks into even smaller pieces such that reprocessing is done selectively.

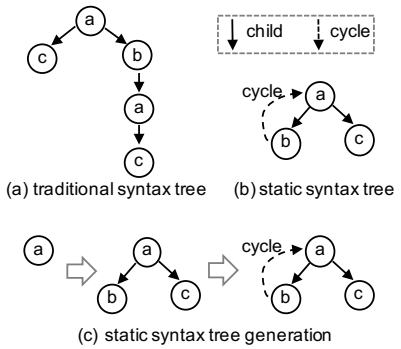

Figure 6. Static Syntax Tree

Figure 5 illustrates both the basic and speculative GAP pushdown transducers. We next elaborate the two modes of GAP in the following two sections.

4. Non-Speculative GAP

When the grammar of the semi-structured data is available, GAP runs in non-speculative mode. In this section, we first describe the two steps to infer feasible paths: static syntax tree generation and symbolic pushdown transducer execution, then present more details about the new features of GAP pushdown transducers.

4.1 Static Syntax Tree Generation

Grammars written in textual rules are hard to interpret directly. To effectively leverage the grammar, we need a data structure to capture all the nesting relations among different input symbols/elements in a concise form.

In fact, the nesting relations can be naturally represented as a tree structure. A traditional syntax tree (or parse tree) indicates the derivation of a concrete sentence based on its (context-free) grammar. Figure 6-a shows the syntax tree of the running example (Figure 4-b). Note that such a syntax

Algorithm 1 Construction of Static Syntax Tree

```
1: procedure STATIC_SYNTAX_TREE_CONSTRUCTION(DTD)
2:     for each element el in DTD do
3:         if el is the first element then
4:             root = create_node(el); //assume the 1st is root
5:             root.cycle = null; // represent recursion if exists
6:             nodes = root;
7:         else/* search el in tree root */
8:             nodes = tree_search(root, el);
9:         for node in nodes do
10:            for each subelement sub of el do
11:                if sub.is_ancestor(node) then // recursion
12:                    node.cycle=sub; // set a backward pointer
13:                else
14:                    child = create_node(sub);
15:                    child.cycle=null;
16:                    node.add_child(child);
17:    return root;
```

tree can grow to very large size as the input gets longer, meanwhile it may not capture all the possible derivation of its context-free grammar.

To represent all the derivation relations in the grammar with a concise format, we introduce *static syntax tree* – a tree generated purely based on the input grammar. As shown in Figure 6-b, in a static syntax tree, each child element must appear and only appear once. The size of a static syntax tree depends on the input grammar, rather than input data, hence will not increase as data grows.

The construction of a static syntax tree is straightforward. Basically, a construction algorithm scans the grammar file, identifies the root element and its child/descendant elements iteratively until all leaf elements (elements without any child elements) are included, as shown in Algorithm 1. For recursive grammars, the algorithm labels nodes that refer to an ancestor by setting the field $cycle$. Figure 6-c illustrates the major steps of executing the algorithm on the running example. The time complexity of this algorithm is $O(n^2)$ where n is the number of elements in the DTD file.

4.2 Symbolic Execution of Pushdown Transducer

The goal of symbolic execution is to find out feasible execution paths for each input symbol. Formally, we define the feasible execution path as follows.

Definition 2. *Given an input symbol $c \in \Sigma$, for any syntactically correct inputs, if there exists an input such that the pushdown transducer can transition to state $s \in Q$ right before it reads c, then we call the execution path that starts from state s as a* feasible execution path *for symbol c.*

According to the definition, to infer the feasible paths for an input symbol, it needs to test all the possible inputs of syntactically correct semi-structured data, which is impossible. We address this by running the pushdown transducer symbolically on the static syntax tree. Basically, the transducer walks on every path of the static syntax tree and records the state right before meeting the start tag and end tag of a node on the path. For example, if the current state is `s` and the current node is `n`, then it adds a `key:value` pair of `<n>:s` and `</n>:s'` into the hash table, where `s'` is the next state after consuming symbol `<n>`.

Algorithm 2 Feasible Paths Inference

```
1: procedure PATH_INFERENCE(root, init_state)
2:     current = init_state; /* current state in automaton */
3:     node = root; /* current node in static syntax tree */
4:     stack.push(root); /* for depth-first traversal */
5:     while stack is not empty do
6:         node = stack.pop();
7:         /* handle a cycle in a recursive grammar */
8:         if node.cycle != null and node.done == false then
9:             /* current state and node in the cycle */
10:            current' = next_state(current, node.start);
11:            n = node.cycle;
12:            /* pushdown transducer moves to next state */
13:            next = next_state(current', n.start);
14:            if next != null then
15:                queue.add(n);
16:                while queue is not empty do
17:                    n = queue.remove();
18:                    next = next_state(current', n.start);
19:                    hash.add(n.start, current');
20:                    hash.add(n.end, next);
21:                    current' = next;
22:                    nodes = DEF_CHILDREN(n, current');
23:                    if nodes is not empty then
24:                        queue.add(nodes);
25:            node.done = true; /* done with this cycle */
26:        if node.visited == false then
27:            next = next_state(current, node.start);
28:            hash.add(node.start, current);
29:            hash.add(node.end, next);
30:            current = next;
31:            node.visited = true;
32:        for each child of node do
33:            if child.visited == false then
34:                stack.push(child);
35:        if node is a leaf then
36:            current = next_state(current, node.end);
37:    return hash;
38:
39: /* get node's children whose start tags are defined in state */
40: procedure DEF_CHILDREN(node, state)
41:    defined_children[];
42:    for each child of node do
43:        if next_state(state, node.start) != null then
44:            defined_children.add(node);
45:    return defined_children;
```

Recursion Handling. For a grammar with recursion, the static syntax tree contains one or more `cycle`(s). In this case, the algorithm unfolds the cycles. However, a naive unfolding would result in dead loops. To address this, we use the finite automaton to guide the unfolding and only unfold it to the extent that is necessary for revealing feasible states.

Algorithm 2 shows the pseudocode of the symbolic execution. It takes the query automaton and static syntax tree as inputs and outputs feasible path hash table. The time complexity of symbolic execution is $O(n + g \cdot |query|)$, where n is the number of nodes in the syntax tree, g is the number of cycles in the syntax tree, and $|query|$ is the length of the query. The time complexity of dealing with cycles depends on the length the query.

Example. Figure 7 illustrates an execution of Algorithm 2 on the running example with the finite automaton in Figure 4-c and the static syntax tree in Figure 6-c. Note that though there is a cycle in the static syntax tree (labeled at node b), the algorithm terminates quickly after it finds that b is not defined at state 4.

```
/* H: hash table, S: stack, Q: queue */
node=a current=1 S{a} H{}  // line 2-4
node=a current=2 S{b,c} H{<a>:1, </a>:2}  // line 26-34
node=c current=0 S{b} H{<a>:1, </a>:2, <c>:2, </c>:0}  // line 26-34
since c is a leaf node, current=2 (i.e., the next state after </c>)  // line 35-36
node=b (b has a cycle) current'=3 n=a Q{a}  // line 6-15
current'=4, Q{c} H{<a>:{1,3}, </a>:{2,4}, <c>:2, </c>:0}  // line 16-24
/* since b is not defined at state 4, only c added to queue Q{c} */
current'=5 Q{} H{<a>:{1,3}, </a>:{2,4}, <c>:{2,4}), </c>:{0,5}}  // cycle is done
node=b current=2, since b is still unvisited, process b  // line 26
S{} H{<a>:{1,3}, </a>:{2,4}, <b>:2, </b>:3, <c>:{2,4}, </c>:{0,5}}  // finish
```

Figure 7. Algorithm 2 execution on the running example.

4.3 GAP Pushdown Transducers

GAP pushdown transducers are based on the basic parallel pushdown transducers (Section 2.3) with two novel features:

- *Dynamic path elimination*: GAP pushdown transducers leverage feasible path inference to dynamically eliminate impossible execution paths at runtime, based on specific local context.
- *Runtime data structure switching*: Benefited from path elimination and path convergence, GAP pushdown transducers switch between a stack and a double-tree data structure at runtime to maximize the efficiency.

At high-level, GAP pushdown transducers follow the same flow as basic parallel pushdown transducers: (i) *split phase*, (ii) *parallel phase*, and (iii) *join phase*. The main differences between GAP pushdown transducers and basic parallel pushdown transducers lie in the parallel phase. In this phase, each transducer processes an input chunk while maintaining a set of possible execution paths through a set of mappings M. A mapping is defined as follows:

Definition 3. *Given an input chunk i, a mapping m^i for i is a 5-tuple $m^i = (q_s, z_s, q_f, z_f, o)$, $m^i \in M = Q \times \Gamma^* \times Q \times \Gamma^* \times \Delta^*$ where q_s and z_s are the starting state and stack right before processing i, q_f, z_f, and o are the finishing state, stack, and output tape after processing i.*

Basically, a mapping embodies the basic information of an execution path. For example, a mapping (1, ϵ, 4, 1:2:3, 1) says that if the transducer starts from state 1 with an empty stack, then after processing the input chunk, it will end at state 4 with a stack of 1:2:3 (":" separates items in a stack) and an output tape containing a match. Such information is used later for joining the results of different transducers.

Maintaining the mappings brings a significant amount of cost, comparing to a single path execution in a sequential transducer [30]. To reduce such parallelization cost, GAP transducers feature two new optimizations: *dynamic path elimination* and *runtime data structure switching*. Both of them happen in the parallel phase.

Dynamic Path Elimination. A GAP transducer leverages feasible path inference to prune impossible paths in three situations on the fly:

(1) At the beginning of a chunk (except the first one), a GAP transducer uses the first symbol of the chunk as the key to look up the feasible path hash table. In return, it obtains a list of feasible execution paths Q_{hash}, which is often a strict subset of all execution paths Q (i.e., $Q_{hash} \subseteq Q$). Then it processes the chunk with only the paths from Q_{hash}. For example, a GAP transducer that starts at Line 5 of the XML data in Figure 4-b would look up <c> in the hash table (the last line of Figure 7), and keeps only paths {2, 4}.

(2) During the processing of a chunk, path divergences might happen – when a pop operation is needed while the stack is empty, the parallel transducer has to enumerate all possible states that could be popped out from the stack. Prior work [30] extends the finite automaton by defining the transitions for all end tags, and only includes states that have a defined transition for the given end tag (denoted as Q_{fa}). Since the inference is only based on the finite automaton (i.e., query), regardless the input grammar, the state amount of reduction is limited [2]. In comparison, when a path divergence occurs, a GAP transducer inquires the feasible path hash table that is built based on both the query and the input grammar. Using the given end tag as the key, it can obtain a potentially smaller set of possible states (i.e., $Q_{hash} \subseteq Q_{fa}$). Since the path divergence may happen multiple times during the processing of a chunk, the feasible path inference-based elimination has great potential to reduce the path maintenance cost. Following the example in situation (1), when the GAP transducer processes Line 6, it looks up in the hash table and chooses paths {2, 4}, instead of {2, 4, 0} – the choice of prior work [30].

(3) When processing the first start tag right after a path divergence (e.g., the tag <b> in </d></c><b>, supposing that path divergence happened at both </d> and </c>), a GAP transducer also inquires the feasible path table with the first tag (<b>). The purpose is to further reduce the number of execution paths. This is achieved the taking the intersection between the set of execution paths before the first start tag Q_{old} and the returned path set from feasible path table inquiry Q_{hash} (i.e., $Q_{old} \cap Q_{hash}$).

Runtime Data Structure Switching. To reduce the cost of mapping maintenance, basic parallel pushdown transducers use a double-tree data structure [30], which compresses the mapping and reduces redundant computation in mapping updates. While being more efficient than maintaining each mapping separately, it is much slower than the stack-based execution of a sequential pushdown transducer.

[2] In evaluation, we found it often fails to reduce the possibilities of popped-out states due to the inclusion of a special state that handles unrelated tags (e.g., state 0 in the running example).

An interesting observation we found is that the number of feasible execution paths often drops to one, thanks to the dynamic path elimination and path convergence. To leverage this insight, we allow GAP transducers switch back to stack-based execution once a single feasible path is left.

More specifically, when a path elimination is performed, the GAP transducer checks the number of feasible execution paths. If the transducer finds only one path is left, then it switches to use a stack to continue its execution until the next path elimination check; otherwise it uses a double-tree data structure to maintain multiple execution paths. As discussed earlier, a path may diverge into multiple ones under certain circumstances (see Section 2.3). In such cases, the GAP transducers switch to a double-tree data structure right after the divergence. Since path elimination can happen multiple times for a GAP transducer depending on the contents in the chunk, the data structure switching can also occur multiple times. We hence refer to it as *runtime data structure switching*. Note that the switching cost is usually negligible as the switching typically occurs less than 5 times in millions of transitions, according to our experiments.

Finally, once each GAP pushdown transducer finishes its chunk, the mappings from different transducers are joint pair by pair using the same rules as basic parallel pushdown transducers [30]. Basically, to join a pair of mappings m^1 and m^2, it requires their states and stacks are matched appropriately (e.g., $m^1(q_f) = m^2(q_s)$).

In sum, GAP pushdown transducers are based on basic parallel pushdown transducers and work in three phases. With the two novel features, dynamic path elimination and runtime data structure switching, their execution cost can be potentially reduced.

5. Speculative GAP

In certain scenarios, a pre-defined grammar may not be available. For example, the grammar has not been explicitly defined or its access is not granted. GAP addresses this challenge with speculative execution. In this section, we first describe a method for extracting partial grammars from input data, then introduce the speculative execution of GAP pushdown transducers, including a local reprocessing technique to reduce the misspeculation penalty.

5.1 Partial Grammar Extraction

In the absence of a pre-defined grammar, we leverage the insight that *input data implicitly dictate the grammar to a certain extent, though may not cover the entire grammar*. Specifically, we design a method that automatically extracts a partial static syntax tree from prior inputs.

Many applications process semi-structured data from the same source repetitively. For example, web analytics that receive semi-structured data from a specific social network website or data exchanges between two enterprises. In such scenarios, the input data from run to run tend to follow a similar *structure* as they are all defined by the same "hidden" grammar. Hence, it provides an opportunity to "learn" grammar from earlier runs.

Algorithm 3 shows a straightforward way to extract a (partial) static syntax tree directly from an input stream. More sophisticated grammar learning algorithms are also available from prior studies [35, 36].

Algorithm 3 Static Syntax Tree Extraction from Input Data

```
1:  procedure STATIC_SYNTAX_TREE_EXTRACTION(stream)
2:      while stream has next tag t do
3:          if t is the first element then /* root node */
4:              root = create_node(t);
5:              stack.push(root);
6:          else if t is a start tag then
7:              parent = stack.top();
8:              /* find a child of parent with tag t */
9:              child = find_child_by_tag(parent, t);
10:             if child != null then
11:                 stack.push(child);
12:             else /* create a new child for parent */
13:                 node = create_node(t);
14:                 parent.add_child(node);
15:                 stack.push(node);
16:         else if t is an end tag and t == stack.top().end then
17:             stack.pop();
18:         else
19:             print("errors in inputs");
20:     return root;
```

5.2 Speculative GAP Pushdown Transducers

Speculative GAP pushdown transducers are augmented with the capabilities of validating the speculative execution paths and initiating appropriate reprocessing when speculation fails. Since Section 4.3 already describes the basics of GAP pushdown transducers, here we focus on the aspects of speculative execution.

Speculative Execution. With the extracted grammar, GAP can infer feasible paths in the same way as inferring from a pre-defined grammar. The difference lies in the results – the feasible paths may be incomplete due to missing parts of the extracted grammar. Consequently, GAP transducers run speculatively, with the potential of missing the true path. Similarly to other speculation schemes, speculative GAP transducers first run with speculated data (i.e., feasible paths from extracted grammar), then rely on the validation and reprocessing to guarantee the correctness.

Note that when a speculative GAP transducer looks up an incomplete feasible path table, it may not even find any corresponding feasible paths for the inquired input symbol, simply because the symbol does not appear in the extracted grammar. In such situations, the GAP transducers degrade themselves to basic parallel pushdown transducers and enumerate all execution paths.

Another difference with the non-speculative transducers is about the third scenario of dynamic path elimination (see Section 4.3). When processing the first start tag after a path divergence, it does not take the intersection between two path sets Q_{old} and Q_{hash}. Instead, it simply uses Q_{hash} to replace Q_{old}, for two reasons: first, this creates chances

to correct an earlier misspeculation; second, it allows the reprocessing to be performed selectively (as explained next).

Validation and Reprocessing. Once every GAP transducer finishes its chunk (speculatively), they enter into the join phase, where the validation and reprocessing are performed.

According to Section 4.3, the feasible path table is used in three scenarios: (1) at the beginning a chunk (except the first one); (2) in the occurrence of a path divergence; (3) when processing the first start tag after a path divergence. Since the feasible path table is extracted from a partial grammar, it might miss the true execution path. Hence, a validation is required for each of the above three scenarios, respectively. For example, during the joining of a pair of mappings from two consecutive chunks, if the starting states of the latter chunk do not include the correct finishing state of the earlier chunk, a misspeculation is detected. In either case, when a validation fails, a GAP transducer would reprocess the failed part with the corresponding correct starting state. Note that a misspeculation at the beginning of of a chunk does not necessarily mean that the whole chunk needs to be reprocessed. Thanks to the other two dynamic path elimination scenarios, an earlier misspeculation might be corrected by a latter path elimination operation. Hence, the reprocessing of a chunk is usually performed selectively.

6. Evaluation

We evaluate GAP on a set of real-world XML benchmarks and examine its efficiency and scalability on a variety of path queries, including a set of 200 queries.

Implementation. We prototyped GAP in C language. It includes four major components: a static syntax tree generator that takes a DTD/XSD grammar as input and outputs a static syntax tree; a symbolic pushdown transducer executor that runs symbolically over a given static syntax tree; a multi-threaded GAP pushdown transducer implemented using Pthread and can be tuned into either *speculative mode* or *non-speculative mode* (by default, it is set to speculative mode), as well as a grammar extractor that can be enabled either online (for streaming data) or offline (for storage data).

Methodology. We compare GAP with the state-of-the-art parallel pushdown transducers [30], which has shown comparable performance to PugiXML [22], a popular C++ XML processing library and much better performance than Expat [6]. Table 2 summarizes the versions used in our comparison. For speculative GAP versions, we randomly choose a portion of the complete grammar [3].

All experiments run on a 20-core machine equipped with two Intel 2.13 GHz Xeon E7-L8867 processors. The machine runs openSUSE Leap 42.1 and has GCC 4.8.5. All programs are compiled with "-O3" optimization flag. The timing results reported are the average of 10 repetitive runs. We do not report 95% confidence interval of the average when the variation is not significant.

[3] To ensure the partial grammar is meaningful, we randomly and recursively remove leaf elements from the original grammar.

Method Versions	Abbreviation
Parallel pushdown transducer [30]	PP-Transducer
Non-speculative GAP	GAP-NonSpec
Speculative GAP with 20% grammar	GAP-Spec(20%)
Speculative GAP with 40% grammar	GAP-Spec(40%)
Speculative GAP with 80% grammar	GAP-Spec(80%)

Table 2. Method Versions in Evaluation

Benchmarks. Table 3 lists datasets used in our experiments as well as their statistics. Except XMark, all the datasets are from a commonly used UW XML data repository [40], which covers a variety of XML applications. To make a fair comparison with prior work [30], we also use a scaling factor that replicates the datasets, resulting in sizes from 600MB to 6GB. We do not report results for larger datasets because the measurements are stable due to the large amount of repetitive computations involved in semi-structured query processing.

Dataset	#tags	d_{max}	d_{avg}	Dataset	#tags	d_{max}	d_{avg}
Lineitem	34,781,152	3	2.94	DBLP	33,321,292	6	2.9
SwissProt	29,770,302	5	3.55	NASA	237,230,520	8	5.58
Protein	42,597,466	7	5.15	XMark	23,328,398	13	5.55

Table 3. XML datasets (d means depth)

We use queries from XPathMark[14] for its designated purpose of XPath query evaluation and its realistic query structures. As listed in Table 4, the query set covers the whole set of A-type queries as well as two B-type queries in XPathMark. When predicates, parents or ancestors are used, the queries are translated into subqueries or rewritten, such that they can be merged into a single pushdown transducer. The right-most two columns show the number of subqueries and number of matches in the datasets.

Single-Query Performance. The speedup for single-query processing is shown in Figure 8 (left). The baseline is the sequential pushdown transducer.

Among the five versions, GAP-NonSpec yields the best speedup on all tested queries, leading to an average of 15X speedup. In comparison, PP-Transducer achieves least speedup except for benchmarks XM1 and XM2, in which cases the GAP-Spec(20%) performs the worst. It is easy to notice the clear trend among the three speculative versions. As the extracted syntax tree becomes more complete, the performance improves. However, the trend varies across different benchmarks, implying that the performance of speculative versions are less predictable. It is worth to note that even with 20% syntax tree, the GAP-Spec still yields better performance than PP-Transducer (13.2X v.s. 11.6X).

To better understand the performance results, we profiled the average number of starting execution paths, as shown in Table 5. The last row of single query block shows the geometrical mean of the number of starting execution paths for each version. It clearly shows a big discrepancy between PP-Transducer and GAP-NonSpec. The discrepancy confirms the effectiveness of the proposed dynamic path elimination and demonstrates its benefits in improving the performance.

Query	Dataset	Query structure	#sub	#matches
NS1	NASA	/ds/d/tb/ts/tl/tit	1	2,119,760
NS2	NASA	//ds/d/tit	1	3,395,250
PT1	Protein	/pd/pe/r/ri/xs/x/u	1	279,402
PT2	Protein	/p/pe//u	1	1,949,416

Query	Dataset	Query structure	#sub	#matches
DP1	DBLP	/dp/ar/au	1	1,107,323
DP2	DBLP	//dp//ed	1	31,940
DP4	DBLP	/dp/ar[tit]/jn	3	558,046
XM3	XMark	//k/ancestor::li/t/k	3	241,556

Query	Dataset	Query structure	#sub	# matches
NS3	NASA	/ds/d[descendant::tit or descendant::na or descendant::kw]/an	5	1,826,250
NS4	NASA	/ds/d[tit and al]/r/s/o/au/ln	4	191,250
PT3	Protein	/pd/pe/r[aci/acs or at or ct or nt]/ri/ats/at	6	5,668,287
DP3	DBLP	/dp[mt/au or mt/tit or mt/yr or pt/au or pt/tit or pt/yr or ...]/au	43	1,107,323
XM1	XMark	/s/r/*/item/[parent::af]/name	1	3,851
XM2	XMark	//s[r/*/item[parent::au parent::af ... parent:: eu]/mb/m/t/k/ b or ...]/pp/ps/ name	18	178,500

Table 4. XPath queries. #sub shows the number of sub-queries in each query structure.

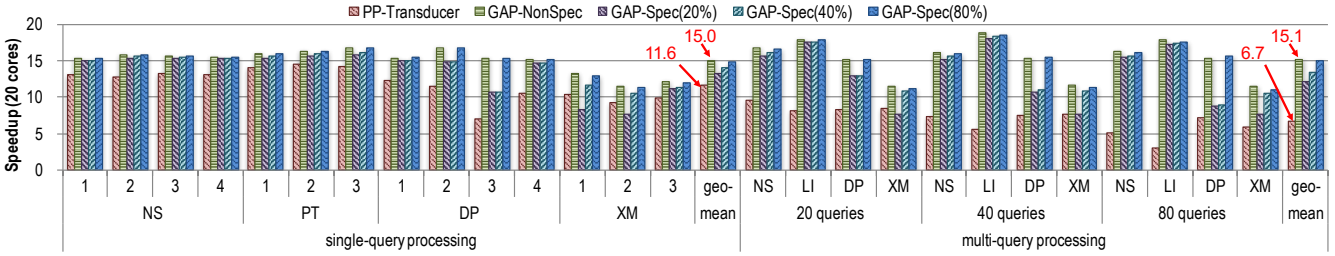

Figure 8. Speedup of Single-Query and Multi-Query Processing on A 20-core Machine

Query Set		PP-Trans.	GAP-NonSpec	GAP-Spec 20%	GAP-Spec 40%	GAP-Spec 80%
single query	NS	7.7	1.0	5.7	4.8	1.1
	LI	8.4	1.1	5.3	1.3	1.1
	DP	9.0	1.7	3.1	3.1	1.6
	XM	12.4	1.9	6.3	5.5	3.7
	Geo	9.2	1.4	4.9	3.2	1.7
80 queries	NS	275.0	1.1	192.4	157.6	5.3
	LI	166.0	1.0	60.6	51.4	28.5
	DP	96.0	3.2	18.9	17.4	2.5
	XM	285.0	5.4	119.8	75.2	33.4
	Geo	188.0	2.1	71.7	57.0	10.6

Table 5. Average Number of Starting Execution Paths

Query (set)		GAP-Spec(20%) cost	GAP-Spec(20%) acc.	GAP- Spec(40%) cost	GAP- Spec(40%) acc.
single query	DP1	0.003%	94.12%	0.003%	94.12%
	DP3	0.002%	94.12%	0.002%	94.12%
	DP4	0.003%	94.12%	0.004%	94.12%
	XM1	26.85%	62.50%	0%	100%
	XM2	25.10%	47.83%	0%	100%
query set	DP (20)	0.003%	88.24%	0.002%	88.24%
	DP (40)	0.002%	94.12%	0.003%	94.12%
	DP (80)	0.002%	96.43%	0.002%	96.43%
	XM (20)	24.18%	56.52%	0%	100%
	XM (40)	24.14%	56.52%	0%	100%
	XM (80)	26.02%	54.17%	0%	100%

Table 6. Speculation Accuracy and Reprocessing Cost

Multi-Query Performance. To evaluate the performance of multi-query processing, we use 12 groups of queries, with three different sizes: 20 queries, 40 queries, and 80 queries.

Multi-query processing speedup is shown in Figure 8 (right). Overall, the results follow a similar pattern as single-query processing. However, the performance differences among different versions become larger, especially between PP-Transducer and four GAP versions. Specifically, GAP-NonSpec produces 15.1X, similarly to its single-query performance, while PP-Transducer only yields 6.7X speedup. The reason is that when processing multiple queries concurrently, the pushdown transducers grow larger with more number of states. As a result, PP-Transducer ends up enumerating more paths, causing higher parallelization cost.

Table 5 reports the number of starting paths for all five versions. In single-query processing, the gap between PP-Transducer and GAP-NonSpec is about 10X (9.2 vs. 1.4 on average). While in multi-query processing, the gap quickly increases up to hundreds of times (188 vs. 2.1 on average).

Speculation Accuracy and Cost. Table 6 reports the speculation accuracy and the cost of misspeculation, including both single queries and query sets. The ones that are not listed do not have any misspeculation. The misspeculation cost is typically a very tiny portion of the total execution time, except queries to XM dataset which shows more than 24% misspeculation cost when using 20% grammar (GAP-Spec(20%)). There are two main reasons causing such relatively high cost: First, the speculation accuracy of these queries are relatively low, slightly higher than 50%, resulting in a substantial amount of reprocessing; Second, a few elements that appear quite often in the dataset are missing in the 20% partial grammar, in which cases, the speculative GAP transducers degrade themselves to basic parallel transducers and enumerate all execution paths.

Scalability. We measured the scalabilities in terms of both the number of cores and the number of queries.

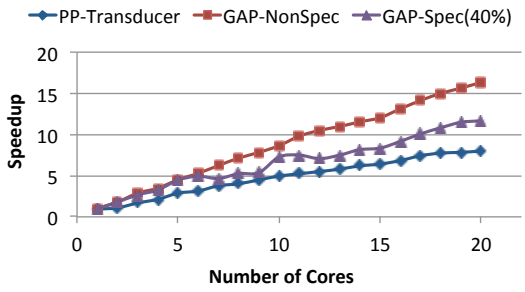

Figure 9. Scalability over Number of Cores

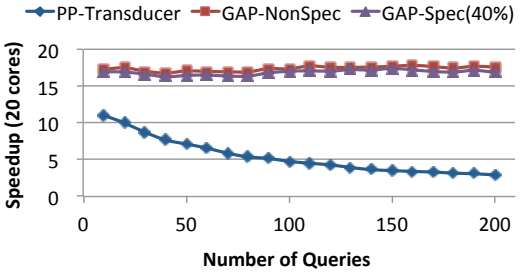

Figure 10. Scalability over Number of Queries

Figure 9 shows the scalability in terms of number of cores. All three versions show good scalability – the speedup linearly increases up to at least 20 cores. Meanwhile, it also clearly shows the trend of their differences – as the number of cores increases the performance gap among these three versions will become even larger.

Figure 10 shows the scalability in terms of number of queries. PP-Transducer shows a sharp decrease as the number of queries increases, which aligns well with the results reported by prior work [30]. In comparison, the two GAP versions show no degradation at all up to at least 200 queries, thanks to the dynamic path elimination.

7. Related Work

There is a large amount of efforts on making parsing parallel for both context-free grammars and non-CFG grammars [16, 17, 29, 37, 41]. They provide valuable insights to this work. In particular, [16] leverages some rules extracted from the input structure to facilitate the parallel expression evaluation. Next, we will focus our discussions on XPath querying and its parallelism exploitation.

XML querying. Many prior work study the expressiveness of XPath querying and the execution of concurrent queries efficiently, including automata-based techniques [2, 44], array-based methods like TurboXPath [21] and stack-based algorithms like Twig2Stack [5]. This work uses an approach that combines a number of small queries into a single DFA and evaluates them simultaneously [19]. Y-Filter [11] and XMLTK [2] are based on this method and address state explosion with lazy DFA.

Some earlier work exploiting data parallelism in XPath queries rewrite an XPath query with predicates into several sub-queries, execute the sub-queries in parallel and merge their results sequentially [4]. However, this approach demands more hardware resources meanwhile exposes limited parallelism. The work by Zhang and others [44] executes multiple states in the NFA in parallel. Although each active state is handled by a different thread, the XML data is still processed sequentially. An alternative method proposed by Liu and others [24] uses a parallel structured join algorithm, where both query and join operations are parallelizable. However, constructing the required data structures is still a sequential step. This work is primarily based on the PP-Transducer [30] which can operate on arbitrarily framed XML chunks. More details are given in Section 1 and 2.3.

Parallel XML Parsing. There are two basic ways of data-level parallel parsing for XML: the partition-oriented and the merging-oriented [25, 33, 39, 43, 45]. The work by Lu and others, for example, first extracts the high-level structure of XML documents through a quick prescan [25], and parses each part of the document in parallel. In comparison, Wu and others [43]'s method cuts XML documents into chunks directly, parses them and merges the result together.

Other Parallelization. Recent work on parallelizing finite automata [28, 34, 46, 47] provides useful insights, but cannot be applied directly to pushdown transducers, due to the involvement of stack. Zhao and others design stack-based automata that predict future function calls to enable parallel Just-in-Time compilation [48]. Saeed and others [26] exploit the rank convergence in parallel dynamic programming, sharing some of the high-level ideas of this work.

8. Conclusion

This paper presents GAP, a novel parallelization scheme that leverages the grammar of semi-structured data to improve the parallelization efficiency. Depending on the availability of a pre-defined grammar, GAP can run in both speculative mode and non-speculative mode. In either case, GAP is able to infer feasible execution paths by generating a static syntax tree and executing the pushdown transducer symbolically on the tree. By feeding such information to the GAP pushdown transducers, unnecessary execution paths can be eliminated on the fly. In addition, GAP transducers feature a runtime data structure switching to further take advantage of path elimination and maximize the efficiency. Evaluation on real-world datasets and queries confirm the benefits of grammar-aware parallelization which yields consistent speedup to a large number of concurrent queries.

Acknowledgments

We thank all anonymous reviewers for their constructive comments and our paper shepherd Armando Solar for his great help with the final preparation. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1565928.

References

[1] D. Arroyuelo, F. Claude, S. Maneth, V. Makinen, G. Navarro, K. Nguyen, J. Siren, and N. Valimaki. Fast in-memory xpath search using compressed indexes. *Software: Practice and Experience*, 45(3):399–434, 2015.

[2] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. Xmltk: An xml toolkit for scalable xml stream processing. 2002.

[3] P. Boncz, T. Grust, M. Van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490. ACM, 2006.

[4] R. Bordawekar, L. Lim, A. Kementsietsidis, and B. W. Kok. Statistics-based parallelization of xpath queries in shared memory systems. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 159–170, 2010.

[5] S. Chen, H. Li, J. Tatemura, W. Hsiung, D. Agrawal, and K. S. Candan. Twig^{2}stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 283–294, 2006.

[6] J. Clark, C. Cooper, and F. Drake. The expat xml parser, 2011.

[7] DBLP Team. Welcome to DBLP. `http://dblp.uni-trier.de/`, 2016. Retrived: 2016-07-01.

[8] F. S. de Boer, M. M. Bonsangue, J. Jacob, A. Stam, and L. Van der Torre. Enterprise architecture analysis with xml. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 222b–222b. IEEE, 2005.

[9] E. developers program. What is the ebay api? `https://go.developer.ebay.com/what-ebay-api`, 2016. Retrived: 2016-07-01.

[10] Y. Diao, P. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 341–342. IEEE, 2002.

[11] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 341–342, 2002.

[12] A. Dunkels et al. Efficient application integration in ip-based sensor networks. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 43–48. ACM, 2009.

[13] Facebook. Public feed API. `https://developers.facebook.com/docs/public_feed`, 2016. Retrieved: 2016-07-01.

[14] M. Franceschet. Xpathmark: Functional and performance tests for xpath. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[15] F. Galiegue and K. Zyp. Json schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, 2013.

[16] A. Gibbons and W. Rytter. *Efficient parallel algorithms*. Cambridge University Press, 1989.

[17] A. Gibbons and W. Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Information and Computation*, 81(1):32–45, 1989.

[18] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata. In *International Conference on Database Theory*, pages 173–189. Springer, 2003.

[19] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, pages 173–189, 2003.

[20] D. Halstead. What sort of network and storage setup will be required to ingest the entire twitter firehose for 1 year. `http://goo.gl/kFXDH`. Retrieved: 2016-07-01.

[21] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *VLDB J.*, 14(2):197–210, 2005.

[22] A. Kapoulkine. pugixml: a light-weight, simple and fast xml parser for c++ with xpath support. `http://pugixml.org/`. Retrieved: 2016-07-01.

[23] L. Libkin, W. Martens, and D. Vrgoc. Querying graph databases with xpath. In *Proceedings of the 16th International Conference on Database Theory*, pages 129–140. ACM, 2013.

[24] L. Liu, J. Feng, G. Li, Q. Qian, and J. Li. Parallel structural join algorithm on shared-memory multi-core systems. In *The Ninth International Conference on Web-Age Information Management, WAIM 2008, July 20-22, 2008, Zhangjiajie, China*, pages 70–77, 2008.

[25] W. Lu, K. Chiu, and Y. Pan. A parallel approach to xml parsing. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, GRID '06, pages 223–230, 2006.

[26] S. Maleki, M. Musuvathi, and T. Mytkowicz. Parallelizing dynamic programming through rank convergence. *ACM SIGPLAN Notices*, 49(8):219–232, 2014.

[27] A. Mitra, M. Vieira, P. Bakalov, W. Najjar, and V. Tsotras. Boosting xml filtering with a scalable fpga-based architecture. *arXiv preprint arXiv:0909.1781*, 2009.

[28] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *ASPLOS '14: Proceedings of 19th International Conference on Architecture Support for Programming Languages and Operating Systems*. ACM Press, 2014.

[29] A. Nijholt. The cyk approach to serial and parallel parsing. 1991.

[30] P. Ogden, D. Thomas, and P. Pietzuch. Scalable XML query processing using parallel pushdown transducers. *Proceedings of the VLDB Endowment*, 6(14):1738–1749, 2013.

[31] S. O'Riain, E. Curry, and A. Harth. Xbrl and open data for global financial ecosystems: A linked data approach. *International Journal of Accounting Information Systems*, 13(2):141–162, 2012.

[32] S. Pal, V. Parikh, V. Zolotov, L. Giakoumakis, and M. Rys. Xml best practices for microsoft sql server 2005. *Retrieved*, 11:2004, 2004.

[33] Y. Pan, W. Lu, Y. Zhang, and K. Chiu. A static load-balancing scheme for parallel xml parsing on multicore cpus. In *Proc. of the 7th International Symposium on Cluster Computing and the Grid (CCGRID)*, 2007.

[34] J. Qiu, Z. Zhao, and B. Ren. Microspec: Speculation-centric fine-grained parallelization for fsm computations. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 221–233. IEEE, 2016.

[35] Y. Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science*, 76(2-3):223–242, 1990.

[36] Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, 1992.

[37] G. Satta and O. Stock. Bidirectional context-free grammar parsing for natural language processing. *Artificial Intelligence*, 69(1):123–164, 1994.

[38] L. Schor, P. Sommer, and R. Wattenhofer. Towards a zero-configuration wireless sensor network architecture for smart buildings. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 31–36. ACM, 2009.

[39] B. Shah, P. Rao, B. Moon, and M. Rajagopalan. A data parallel algorithm for xml dom parsing. *Database and XML Technologies, Lecture Notes in Computer Science*, 5679, 2009.

[40] D. Suciu. XML data repository. `http://www.cs.washington.edu/research/xmldatasets/`, 2003. Retrieved: 2016-07-01.

[41] N. Takashi, T. Kentaro, K. Taura, and J. Tsujii. A parallel cky parsing algorithm on large-scale distributed-memory parallel machines. 1997.

[42] A. H. Wang. Don't follow me: Spam detection in twitter. In *Security and Cryptography (SECRYPT), Proceedings of the 2010 International Conference on*, pages 1–10. IEEE, 2010.

[43] Z. Y. Yu Wu, Qi Zhang and J. Li. A hybrid parallel processing for xml parsing and schema validation. In *Balisage: The Markup Conference 2008*.

[44] Y. Zhang, Y. Pan, and K. Chiu. A parallel xpath engine based on concurrent NFA execution. In *16th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2010, Shanghai, China, December 8-10, 2010*, pages 314–321, 2010.

[45] Z. Zhao, M. Bebenita, D. Herman, J. Sun, and X. Shen. Hpar: A practical parallel parser for html–taming html complexities for parallel parsing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):44, 2013.

[46] Z. Zhao and X. Shen. On-the-fly principled speculation for FSM parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 619–630, 2015.

[47] Z. Zhao, B. Wu, and X. Shen. Challenging the "embarrassingly sequential": Parallelizing finite state machine-based computations through principled speculation. In *ASPLOS '14: Proceedings of 19th International Conference on Architecture Support for Programming Languages and Operating Systems*. ACM Press, 2014.

[48] Z. Zhao, B. Wu, M. Zhou, Y. Ding, J. Sun, X. Shen, and Y. Wu. Call sequence prediction through probabilistic calling automata. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 745–762. ACM, 2014.