Scalable XML Query Processing using Parallel Pushdown Transducers

Peter Ogden
Imperial College London
London, United Kingdom
p.ogden12@imperial.ac.uk

David Thomas
Imperial College London
London, United Kingdom
d.thomas1@imperial.ac.uk

Peter Pietzuch
Imperial College London
London, United Kingdom
prp@doc.ic.ac.uk

ABSTRACT

In online social networking, network monitoring and financial applications, there is a need to query high rate streams of XML data, but methods for executing individual XPath queries on streaming XML data have not kept pace with multicore CPUs. For data-parallel processing, a single XML stream is typically split into *well-formed* fragments, which are then processed independently. Such an approach, however, introduces a sequential bottleneck and suffers from low cache locality, limiting its scalability across CPU cores.

We describe a data-parallel approach for the processing of streaming XPath queries based on pushdown transducers. Our approach permits XML data to be split into *arbitrarily-sized* chunks, with each chunk processed by a parallel automaton instance. Since chunks may be malformed, our automata consider all possible starting states for XML elements and build mappings from starting to finishing states. These mappings can be constructed independently for each chunk by different CPU cores. For streaming queries from the XPathMark benchmark, we show a processing throughput of 2.5 GB/s, with near linear scaling up to 64 CPU cores.

1. INTRODUCTION

The ability to process continuous streams of XML data at a high rate is an important requirement in many application domains. For example, the full "Firehose" stream from Twitter produces XML data at a rate of tens of megabytes per second [18] and is likely to grow significantly in the future. In other domains, such as web analytics, financial data processing, cellular network operations or real-time telematics, stream data rates of 10s or 100s of millions of items per second are not uncommon [1]. Even with static XML datasets, the advent of "big data" means that a single-pass stream processing model becomes the only viable choice when faced with processing 10s of terabytes or petabytes of data generated, for example, by community-driven websites such as Twitter or Wikipedia [32].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 14
Copyright 2013 VLDB Endowment 2150-8097/13/13... $ 10.00.

Users that want to query the structure of XML stream data must therefore rely on the efficient execution of XPath queries [31]. Modern CPUs place an emphasis on multi-threaded performance requiring XPath query processing be parallelised across many CPU cores to achieve high throughput for streams of XML data. While prior work investigated the execution of many concurrent XPath queries through task parallelism [37, 24, 6], it remains an open issue how to execute a small number of queries using data parallelism. For example, in the case of real-time Twitter analytics, a single, XPath query may be executed against the full Twitter Firehose stream to detect all retweets originating from a different geographical area to the original tweet.

A challenge when parallelising the execution of a single streaming XPath query is that it requires XML parsing, which is fundamentally a sequential process—the current state of a parser depends on all previous characters. To parse and query XML data in parallel, it must be split in a way that puts the parser in a well-defined state at the start of each data block in the stream. Proposed solutions include a sequential pre-processing step, which splits the XML data into well-formed fragments [23], and speculative execution based on heuristics to guess the starting state of the parser [36]. These approaches, however, either require enough state to store the entire XML parse tree before running queries, or limit scalability across CPU cores due to the costly sequential processing.

Our goal is to combine the parsing and querying of streaming XML data and parallelise it across many CPU cores, without needing to make any assumptions as to the structure of the data or perform any pre-processing. Our key idea is that it is possible to achieve high data parallelism in XPath query processing by permitting the *out-of-order* processing of potentially malformed parts of XML data. This can be done by executing an XPath query in parallel against individual parts of the data, considering all possible query results for a given part.

Based on this idea, we describe a new approach for executing XPath queries against XML data streams, which can scale to a large number of CPU cores with a constant memory footprint. It uses *Parallel Pushdown Transducers* (PP-Transducers), which take an XML byte stream as input, and output a stream of matched XML elements according to a set of XPath queries. Multiple PP-Transducers can be executed in parallel by splitting the XML data at *arbitrary* byte boundaries into *XML chunks*. Chunks do not need to be well-formed XML fragments and can be processed by separate transducers in parallel on different CPU cores.

Since the surrounding context of an XML chunk in the stream is initially unknown, a PP-Transducer maintains a *mapping* from all possible starting states to corresponding finishing states. As processing progresses, these mappings converge and are joined in a final, inexpensive sequential operation. We describe algorithms for the efficient construction of these mappings and a tree-based data structure to avoid redundant computation when maintaining mappings.

The subset of XPath supported natively by PP-Transducers is limited to child and descendant queries with no support for predicates. To increase expressiveness, complex queries are rewritten into multiple basic queries whose partial results are combined in an inexpensive sequential operation. We also demonstrate the use of query rewriting to support the parent and ancestor axes.

We evaluate the processing throughput and scalability of PP-Transducers against other streaming query techniques. For the standard XPathMark benchmark [29], PP-Transducers achieve a processing throughput of more than 2.5 GB/s, with near linear scaling up to 64 CPU cores. To the best of our knowledge, this constitutes the highest reported throughput for streaming XPath processing on commodity hardware. We show that this result is due to the better cache usage of PP-Transducers compared to existing parallel parser-based techniques, which require well-formed XML fragments.

In summary, the research contributions of the paper are:

(1) the design of parallel pushdown transducers for XML stream processing, which enable out-of-order processing of malformed chunks of XML data by maintaining a mapping from possible starting to finishing states (see §3 and §4);

(2) the description of an efficient tree-based data structure for maintaining state mappings in parallel pushdown transducers, which avoids redundant computation (see §4.2); and

(3) the results from an experimental evaluation of a C++ prototype implementation of parallel pushdown transducers, showing near linear scaling up to 64 CPU cores for an expressive set of XPath queries over XML stream data (see §5).

2. BACKGROUND

Next we give an overview of parallel XML processing approaches (§2.1), and give background on the standard XPath automata (§2.2), which provide the underlying theory for our new parallel pushdown transducers.

2.1 Parallel XPath Query Processing

Existing approaches for XPath query processing have been used in three domains: (1) XML stream processing; (2) XML parsing and querying; and (3) XML-capable database management systems (DBMS).

XML stream processing systems [17, 9, 3] query XML data incrementally with a constant memory requirement. Instead of considering data parallelism, research on XML stream processing [33] has concentrated on increasing either the expressiveness of each query [16], or the number of queries that can be executed in a single run [37].

For example, YFilter [9] and XMLTK [3] evaluate an XML stream against a large number of concurrent XPath queries. Such systems typically use automata, where large numbers of rules are checked against streaming XML data with small individual data items. *Push-down automata* have been used to find XML elements and, through a subset construction, to execute multiple XPath queries simultaneously [17].

Although push-down automata implementations require less state to be maintained compared to complete parse tree construction, they are fundamentally sequential algorithms. While current automata-based approaches can process many rules at the same time, only a single thread processes any given XPath query. In contrast, our work explores how to leverage data parallelism to execute a small number of XPath queries against high rate input data.

XML parsing and querying. XML data can be parsed to create a *parse tree*. While off-the-shelf XML parsers are single-threaded [28, 11], techniques have been proposed to parallelise some of their execution. One option is to run a sequential pre-processor, which splits the data into well-formed fragments that can be processed in parallel [23].

For a large number of CPU cores, however, this sequential pre-processing step becomes a bottleneck. Pan et al. [27] reduce the performance impact of this step by considering multiple possible starting states in the parser, but this only achieves a benefit with low parallelism: beyond 8 CPU cores, the complexity of the pre-processor limits scalability.

While the execution of XPath queries against an XML parse tree can be parallelised [35], this assumes that the tree can be represented in memory, which is infeasible for streaming XML data. Instead Fegaras et al. [12] parallelise query processing using the map-reduce model. Each query operation is realised as part of the reduce phase. The map phase, however, relies on well-formed XML fragments, introducing a sequential bottleneck. Their reported processing throughput on a shared-nothing cluster is several orders of magnitude lower than ours on a single machine.

XML-capable DBMS. Database engines such as Microsoft SQL Server [26] and MonetDB [4] support task-parallel execution of queries over XML data. They construct a relational index over the raw XML data to allow for fast query processing [19], exploiting optimised relational constructs. Dedicated XML DBMS such as Sedna [14] store and index XML data natively without an underlying relational engine.

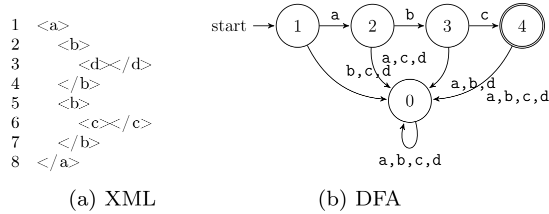
Using a pre-computed index, DBMSs can execute queries faster than approaches that read the XML data on-the-fly. However, index construction becomes unsuitable in a single-pass stream processing model with potentially unbounded streams, or with bounded streams in the terabyte or petabyte ranges that are larger than the processing limit of a given DBMS.

2.2 XPath Automata

Our approach allows for parallel processing of XML data using automata by carefully orchestrating the simultaneous execution of multiple automata. We assume a *deterministic pushdown automaton* (dPDA) for the execution of XPath queries. We base our approach on a simple automaton construction algorithm [17], which turns a set of node selection XPath queries into a *deterministic finite automaton* (DFA). It first creates a non-deterministic finite automaton and then performs a subset construction to obtain the DFA, supporting the following subset of XPath expressions:

$$\begin{aligned} P &::= /N \mid //N \mid P\,P \\ N &::= E \mid A \mid \mathit{text}(S) \mid * \end{aligned}$$

where E, A and S are element names, attributes and strings, respectively.



(a) XML (b) DFA

Figure 1: Sample automaton for XPath query `/a/b/c`

Each start element event, a symbol is pushed onto the stack, and each end element event causes a symbol to be popped from the stack. Pop transitions depend on the symbol on top of the stack. For the automaton described here, the symbols on the stack represent states. Push transitions place the current state on the stack and pop transitions return the execution to the state being popped.

More formally, a dPDA is defined as a 6-tuple of (Q, q_0, Σ, Γ, δ, F) where Q is the set of states, q_0 is the initial state and F is the set of accepting states. The input alphabet Σ is the set of opening and closing XML tags. The pushdown alphabet Γ is the same as Q, and the transition function δ has a push transition for each transition in the automaton and a pop transition for the inverse of each transition.

The constructed dPDA operates on the output of a lexer, which produces events for each opening and closing tag in the XML document. An opening tag event causes the current state of the automaton to be pushed onto the stack and a transition to occur. A closing tag event pops a state off the stack and sets the current state to the popped state.

The dPDAs created by this construction have a particular form: the set of input symbols that cause push transitions and the symbols that cause pop transitions are disjoint. This form of automaton is termed a *nested word automaton* [2].

We enhance the expressiveness of such automata by exploiting their ability to perform multiple queries simultaneously. As described in §3.2, this allows for complex XPath queries to be decomposed into several sub-queries of a supported form. We also employ *rewrite rules* to support queries that use parent and ancestor relationships [25], and not just child and descendent relationships. Both techniques are used to support the queries from the XPathMark benchmark (see §5).

Example. We use a running example to illustrate our automata construction and operation. We consider the XPath query `/a/b/c` on the XML document shown in Fig. 1a. For this query, we construct the DFA in Fig. 1b. Its states 1–4 represent parts of the query, whereas state 0 encodes XML elements that are not mentioned in the query.

3. PROCESSING XML OUT OF ORDER

We present the translation from the dPDA, described in the previous section, to a transducer (§3.1). After that, we explain the intuition behind executing multiple such transducers in parallel to achieve out-of-order processing (§3.2).

Automata are a naturally sequential method of computation because the current state is dependent on all of the data that has been processed. To process XML data in parallel, we split the data into *XML chunks*. Chunks are contiguous, non-overlapping sections of the input XML data. The boundaries of the chunks do not need to fall on XML element boundaries but, for simplicity of explanation, we only show cases in which this occurs. Boundaries that fall within

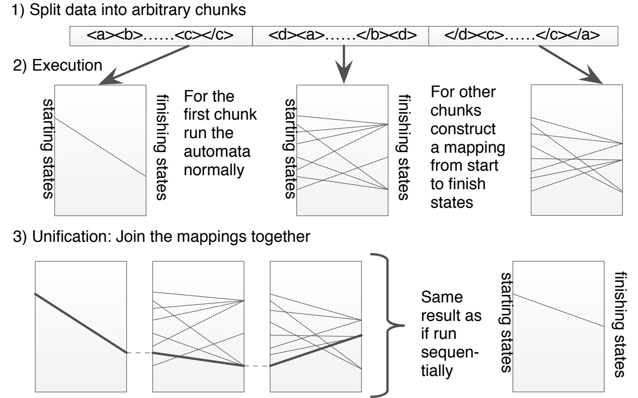


Figure 2: Processing XML data out-of-order by maintaining state mappings in multiple transducers

elements can be handled by considering all possible states of the lexer, analogously to the technique for pushdown transducers described below.

To process data out-of-order, our automaton considers all possible starting states at the start of an XML chunk. For each state, we model the execution of the automaton and determine the resulting output state. Some possible execution paths may result in an accepting state being reached. To keep track of accepting states, we use a transducer model. The transducer places symbols on an output tape instead of reaching accepting states, which captures the idea of multiple XPath rule matches in one XML stream.

3.1 Transducer Construction

The formal description of our transducer is based on the dPDA described in §2.2. The transducer places a symbol on the output tape when an accepting state in the automaton would have been reached. More formally, the transducer is a 6-tuple (Σ, Γ, Δ, Q, q_0, δ) where Σ is the input alphabet, Γ is the pushdown alphabet, Δ is the output alphabet, Q is the set of states, $q_0 \in Q$ is the initial state and δ is the transition function. Each transducer instance maintains a current state ($q \in Q$) and a finite stack of symbols ($z \in \Gamma^*$).

The transition function δ has three parts: $\delta_{plain} : Q \times \Sigma \rightarrow Q \times \Delta$ for transitions that do not affect the stack; $\delta_{push} : Q \times \Sigma \rightarrow Q \times \Gamma \times \Delta$ for transitions that push a value onto the stack; and $\delta_{pop} : Q \times \Sigma \times \Gamma \rightarrow Q \times \Delta$ for transitions that pop a value from the stack.

The output symbol for a transition may be ε, in which case no symbol is generated. To construct the transducer from the dPDA, Σ, Q, q_0 and Γ are reused from the dPDA, the output alphabet contains a symbol for each accepting state in the dPDA, and the transition function is modified so that each transition into an accepting state in the dPDA has an output symbol in the transducer.

A benefit of a transducer model is that it is possible to pipeline the execution of multiple transducers, with one operating on the output tape of another. Processing XML with multiple transducers has two distinct operations. The role of the first transducer is to parse the XML byte stream and create a stream of opening and closing tags. The second transducer takes the stream of these events and determines XPath rule matches. By modelling both operations as transducers, they can be combined into a single transducer.

3.2 Parallel Transducer Execution

To avoid the inherently sequential nature of basic transducers, our transducer operates on mappings from starting to finishing states. We give the intuition behind this approach first, followed by a formal definition in §4.1. As shown in Fig. 2, our approach has four phases:

(i) a *split phase* divides the XML data into a series of XML chunks (step 1). Chunks are not well-formed fragments;

(ii) in a *parallel phase*, the transducer executes multiple times over each chunk, once for each possible start state, in order to construct the state mapping (step 2). A mapping is a set of map entries, which relate a starting state and stack to a finishing state, finishing stack and output tape. As these mappings do not depend on the previous state of the transducer, they can be done in parallel for each chunk;

(iii) after all mappings have been constructed, there is a *join phase* that combines them with the initial state of the transducer. This results in the execution path that would have been followed if the transducer had been executed sequentially (step 3); and

(iv) an additional *filtering phase* to increase the expressiveness of our transducers: XPath queries with predicates are transformed into multiple non-predicate sub-queries. A separate sub-query is created for each element referenced in the predicate and the parent element. For example, the query `/a[b]/c` is rewritten into three sub-queries: `/a`, `/a/b` and `/a/c`. For rewritten queries, the filtering phase selects only matches for which the predicates hold.

While the split, join and filter phases are sequential, they are computationally less expensive than the parallel phase, which is executed by as many threads as CPU cores.

3.3 Convergence

The effectiveness of this approach depends on the amount of work needed to construct the mapping compared to sequential execution. A simple approach would be to run the transducer in each starting state, recording the finishing state and output tape for each. The number of possible execution paths, however, would be the same as the number of states in the automaton, making this approach impractical.

For DFAs and transducers executed in this way, the number of states that need to be considered remains the same or decreases after each input symbol is consumed in an XML chunk [20]. As the number of possible states decreases, the amount of processing for each input symbol also decreases, making the transducer more efficient for larger chunks of data. As long as mappings are processed by considering all entries with the same finishing state simultaneously, it is possible to construct the complete mapping efficiently. This requires a sufficient amount of input data to allow the finishing states to converge to a small number of possibilities.

A stack is required to enhance expressiveness when processing XML. This means that, in addition to the starting state, any states on the stack must also be considered as part of the mapping. Performing a pop transition may cause the number of considered states to increase. Therefore, we can no longer assume that the number of processed states for an input symbol decreases with the length of the input data.

As we show in §5, for a set of XPath queries, the divergence caused by transitions that pop values from the stack is outweighed by the convergence of other transitions, leading to a small number of distinct finishing states. For example, for data chunks of 10 MB, the number of transitions during out-of-order execution is 1.1×–3× compared to executing the transducer directly. The exact overhead depends on the structure of the XML data and the XPath queries (see §5).

4. PARALLEL PUSHDOWN TRANSDUCER

We now describe the execution of the parallel pushdown transducer (PP-Transducer) (§4.1) and explain how it can be implemented efficiently by taking advantage of state convergence to reduce redundant computation (§4.2).

4.1 Formal Description

For the PP-Transducer, the mappings constructed for each XML chunk are from a starting state and starting stack to a finishing state, finishing stack and outputs (i.e. XPath query matches). To define this new execution model, we extend the definitions of the in-order transducer from §3.1.

Each entry in the mapping is defined as $m \in M$ where $m = (q_s, z_s, q_f, z_f, o)$ and $M = Q \times \Gamma^* \times Q \times \Gamma^* \times \Delta^*$. We define q_S and q_f to be the starting and finishing states of the mapping, and z_s and z_f to be the starting and finishing stacks, respectively. We define o to be the output tape when running the transducer with q_s and z_s as the starting state and stack. A complete mapping is thus defined as $s \in S$ where $S = \mathcal{P}(M)$. Each entry in a mapping s represents a different possible starting state and stack.

The two processing functions of the transducer, F and J, are defined as $F : S \times \Sigma \rightarrow S$, which is the function executed on each input symbol, and $J : S \times S \rightarrow S$, which combines two mappings. When processing the beginning of the data, the starting state of the PP-Transducer is $\{(q_0, \varepsilon, q_0, \varepsilon, \varepsilon)\}$; for an out-of-order chunk, it is $\{(q, \varepsilon, q, \varepsilon, \varepsilon) \mid q \in Q\}$. In the former case, the PP-Transducer begins in a single starting state and, in the latter case, all possible starting states must be considered.

The stacks and output tape are represented as strings. To simplify the explanation, we overload the operator : to mean both concatenation of two strings, and the appending of an element to a string.

We realise the processing function $F(s, c)$ through a function $f : M \times \Sigma \rightarrow S$ that performs a transition on an entry in the mapping. This is analogous to a transition function in the original in-order transducer and emulates the action of the transducer for one possible starting state.

The function f is in turn realised by the four functions shown in Alg. 1. Three of the functions, f_{plain}, f_{push} and f_{pop}, wrap the equivalent transition functions of the in-order transducer and result in exactly one state being output. For these three functions, the next state is deterministic, so only a single state can result. The function $f_{unknown}$ accounts for the situation in which a pop transition occurs but there are no symbols on the finishing stack to pop. In this case, all possible symbols are considered, and a new entry in the mapping is created for each symbol. The symbol chosen is placed on the input stack, and the new finish state set as the result of the transition.

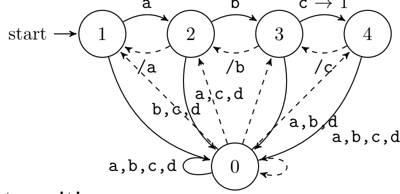
The unification function J unifies two state mappings by considering the cross product of all the entries in the two mappings, and then adding the result of any successfully unified pair to the new mapping. Pairs that cannot be unified are simply discarded. J is implemented in terms of a map entry unification function $j : M \times M \rightarrow M \cup \bot$ such that $J(s^s, s^f) = \{j(m^s, m^f) \mid \forall m^s \in s^s, m^f \in s^f\} \setminus \bot$.

Algorithm 1 Functions for manipulating the mappings during computation

$$
\begin{aligned}
F(s,c) &= \bigcup_{m \in s} f(m,c) \\
f(m,c) &= \begin{cases} \{f_{plain}\} & \text{if } (m[q_f],\, c) \in dom(\delta_{plain}) \\ \{f_{push}\} & \text{if } (m[q_f],\, c) \in dom(\delta_{push}) \\ \{f_{pop}\} & \text{if } (m[q_f],\, c,\, m[z_f]_0) \in dom(\delta_{pop}) \\ f_{unknown} & \text{if } m[z_f] = \varepsilon \text{ and } \exists z'.(m[q_f],\, c,\, z') \in dom(\delta_{pop}) \\ \emptyset & \text{otherwise} \end{cases} \\
f_{plain}(m,c) &= (m[q_s],\, z_s, q,\, m[z_f],\, (o : m[o])) \,|\, (q,\, o) = \delta_{plain}(m[q_f], c) \\
f_{push}(m,c) &= (m[q_s],\, z_s,\, q,\, (z : m[z_f]),\, (o : m[o])) \,|\, (q,\, z,\, o) = \delta_{push}(m[q_f], c) \\
f_{pop}(m,c) &= (m[q_s],\, z_s,\, q,\, zs,\, (o : m[o])) \,|\, (z : zs) = m[z_f] \text{ and } (q,\, o) = \delta_{pop}(m[q_f],\, c,\, z) \\
f_{unknown}(m,c) &= \{\forall z.\, (m[q_s], z_s : [z], q, \varepsilon, (o : m[o]) \,|\, (q,\, o) = \delta_{pop}(m[q_f],\, c,\, z)\}
\end{aligned}
$$

Algorithm 2 Unification rules for merging two map entries. Mapping elements used for unification are shown in boldface.

$$
\begin{aligned}
& j\left((q_s^s,\, z_s^s,\, \mathbf{q},\, \varepsilon,\, o^s),\, (\mathbf{q},\, \varepsilon,\, q_f^f,\, z_f^f,\, o^f)\right) && \text{:- } (q_s^s,\, z_s^s,\, q_f^f,\, z_f^f,\, o^s : o^f) && (1) \\
& j\left((q_s^s,\, z_s^s,\, \mathbf{q},\, z_f^s,\, o^s),\, (\mathbf{q},\, \varepsilon,\, q_f^f,\, z_f^f,\, o^f)\right) && \text{:- } (q_s^s,\, z_s^s,\, q_f^f,\, z_f^s : z_f^f,\, o^s : o^f) && (2) \\
& j\left((q_s^s,\, z_s^s,\, \mathbf{q},\, \varepsilon,\, o^s),\, (\mathbf{q},\, z_s^f,\, q_f^f,\, z_f^f,\, o^f)\right) && \text{:- } (q_s^s,\, z_s^s : z_s^f,\, q_f^f,\, z_f^f,\, o^s : o^f) && (3) \\
& j\left((q_s^s,\, z_s^s,\, \mathbf{q},\, (\mathbf{z} : z_f^s),\, o^s),\, (\mathbf{q},\, (\mathbf{z} : z_s^f),\, q_f^f,\, z_f^f,\, o^f)\right) && \text{:- } j\left((q_s^s,\, z_s^s,\, q_f^s,\, z_f^s,\, o^s),\, (q_s^f,\, z_s^f,\, q_f^f,\, z_f^f,\, o^f)\right) && (4) \\
& j\,(_,\, _) && \text{:- } \bot && (5)
\end{aligned}
$$



→ : push transition
--→ : pop transition (unlabelled from state 0)

Figure 3: Example of PP-Transducer from Fig. 1b

For two entries to be unified, the transducer must finish one chunk in the starting state of the next chunk, and the two stacks must consist of the same stack. This requires that two conditions are met: (i) the finishing state of the first entry must be the starting state of the second; and (ii) one of the starting stacks of the first entry and the finishing stack of the second must be a prefix of the other. When the stacks match, the combined entry has the starting state and stack of the first, and the finishing state and stack of the second. If one of the two stacks used for unification is longer than the other, the symbols common to both are removed, and the excess is added to the resulting state.

The unification function j for map entries is defined in Alg. 2 using Prolog-style unification rules. Rule 1 is the simple case in which no stack is considered. In this case, the finishing state of the entry in the mapping for the first chunk and the start state of the entry for the second entry must be the same. The resulting mapping contains the starting state and stack from the first entry and the finishing stack and state from the second. Rules 2 and 3 handle the cases in which only one of the entries has a stack, meaning the stack is carried through to the unified mapping. Rule 4 eliminates a common symbol off both the finishing stack of the first entry and the starting stack of the second. It is applied recursively until one or both of the stacks are empty. Finally, Rule 5 matches any entries that cannot be unified and returns a failed symbol.

Example. Continuing the running example, we want to process the XML from Fig. 1a using the automaton from Fig. 1b with two parallel threads. Map entries are presented in the form $(q_s,\, z_s) \rightarrow (q_f,\, z_f,\, o)$ to make the mapping property explicit. The transducer operates on complete tags rather than characters.

The first step is to turn the DFA into a transducer in preparation for out-of-order execution, as shown in Fig. 3. Push transitions are represented as solid arrows and result in the current state being pushed; pop transitions are represented by dashed arrows. Pop transitions from state 0 are the inverse of the push transitions but are unlabelled in the diagram for clarity.

The input XML data from Fig. 1a is split into XML chunks with lines 1–4 in the first chunk and lines 5–8 in the second. The starting state at the beginning of the first chunk is a map with one entry corresponding to the starting state of the automaton $\{(1,\, \varepsilon) \rightarrow (1,\, \varepsilon,\, \varepsilon)\}$. The first symbol consumed causes f_{push} to be executed on the entry, pushing 1 onto the stack, and the state is changed to 2. The mapping is therefore $\{(1,\, \varepsilon) \rightarrow (2,\, 1,\, \varepsilon)\}$. The execution of the transducer proceeds analogously for the rest of the chunk. As shown in Fig. 4, the final mapping becomes (M1).

A mapping (M2) for the second chunk is constructed in parallel. The initial mapping for this chunk considers all possible starting states in the deterministic pushdown transducer. First, the opening tags of the `b` and `c` elements are consumed. Each opening tag causes a symbol to be pushed on the stack for each entry in the mapping. The entry starting in State 2 also has an output symbol due to the transition into State 4, which results in the mapping (M3).

The closing tags for the `b` and `c` elements trigger pop transitions for each entry in the mapping. Each entry has two symbols on the stack, leading to two calls to f_{pop}. The resulting mapping (M4) is similar to the initial mapping (M2), but with the difference that a rule has been matched. This is expected because the same point in the XML tree has been reached as at the beginning of the chunk.

$$
\left\{ \begin{array}{ll} (1,\,\varepsilon) & \rightarrow (2,\,1,\,\varepsilon) \\ & \\ & \\ & \\ & \end{array} \right\} \text{(M1)}
\left\{ \begin{array}{ll} (0,\,\varepsilon) & \rightarrow (0,\,\varepsilon,\,\varepsilon) \\ (1,\,\varepsilon) & \rightarrow (1,\,\varepsilon,\,\varepsilon) \\ (2,\,\varepsilon) & \rightarrow (2,\,\varepsilon,\,\varepsilon) \\ (3,\,\varepsilon) & \rightarrow (3,\,\varepsilon,\,\varepsilon) \\ (4,\,\varepsilon) & \rightarrow (4,\,\varepsilon,\,\varepsilon) \end{array} \right\} \text{(M2)}
\left\{ \begin{array}{ll} (0,\,\varepsilon) & \rightarrow (0,\,0:0,\,\varepsilon) \\ (1,\,\varepsilon) & \rightarrow (0,\,0:1,\,\varepsilon) \\ (2,\,\varepsilon) & \rightarrow (4,\,3:2,\,1) \\ (3,\,\varepsilon) & \rightarrow (0,\,0:3,\,\varepsilon) \\ (4,\,\varepsilon) & \rightarrow (0,\,0:4,\,\varepsilon) \end{array} \right\} \text{(M3)}
\left\{ \begin{array}{ll} (0,\,\varepsilon) & \rightarrow (0,\,\varepsilon,\,\varepsilon) \\ (1,\,\varepsilon) & \rightarrow (1,\,\varepsilon,\,\varepsilon) \\ (2,\,\varepsilon) & \rightarrow (2,\,\varepsilon,\,1) \\ (3,\,\varepsilon) & \rightarrow (3,\,\varepsilon,\,\varepsilon) \\ (4,\,\varepsilon) & \rightarrow (4,\,\varepsilon,\,\varepsilon) \end{array} \right\} \text{(M4)}
\left\{ \begin{array}{ll} (0,\,0) & \rightarrow (0,\,\varepsilon,\,\varepsilon) \\ (0,\,2) & \rightarrow (2,\,\varepsilon,\,\varepsilon) \\ (0,\,3) & \rightarrow (3,\,\varepsilon,\,\varepsilon) \\ (0,\,4) & \rightarrow (4,\,\varepsilon,\,\varepsilon) \\ (2,\,1) & \rightarrow (1,\,\varepsilon,\,1) \end{array} \right\} \text{(M5)}
$$

Figure 4: Example of state mappings for the PP-Transducer from Fig. 3

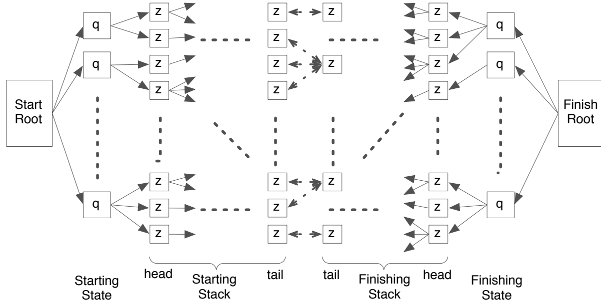


Figure 5: Double tree data structure to compress the state mapping and reduce redundant computation

The final closing tag requires symbols that are unknown to be popped from the stack, calling $f_{unknown}$ for each entry. All possible states that could be popped into are considered. The only states with pop transitions under the `</a>` closing tag are States 0 and 2; entries with all other finishing states are discarded. State 2 can only move into State 1 under a pop transition whereas State 0 can move into States 0, 2, 3 and 4. The result of performing this transition leads to the completed mapping (M5) for the chunk.

To get the complete result of processing the XML data, the final mappings from the two chunks must be unified. As described in §4.1, unification requires that (i) the finishing state of the first entry is the starting state of the second; and that (ii) one of the starting stacks of the first entry and the finishing state of the second must be a prefix of the other. The only two entries that meet these two conditions are the entry in the mapping (M1) (of the first chunk) and the last entry in the mapping M5 (of the second chunk). The resulting mapping, $\{(1,\,\varepsilon) \rightarrow (1,\,\varepsilon,\,1)\}$, indicates that the XML data matches the XPath query of the original automaton.

4.2 Reducing Redundant Computation

A naive implementation of the PP-Transducer would operate on each entry of the mapping independently, resulting in a running time proportional to the number of entries. As this is also proportional to the number of states in the PP-Transducer, it would not increase throughput. Since the per-symbol processing function f depends only on the finishing state and the topmost symbol of the finishing stack, a more efficient solution is to exploit the fact that many mappings have entries that share the same finishing state. All such entries can be processed in parallel.

To achieve this, we propose a data structure based on two trees, as shown in Fig. 5. The *start tree* (on the left) represents all of the starting states and stacks in the mapping, and the *finish tree* (on the right) represents all of the finishing states and stacks.

Each path from one root node to the other is an entry in the mapping. In the start tree, the immediate children of the root are the starting states, q_s, with each subsequent layer of nodes representing symbols in the starting stack, z_s, from right to left. The finish tree has the finishing states, q_f, as the first level, with the finishing stacks, z_f, growing from right to left, beginning from the second level. Multiple leaf nodes in the start tree may be connected to a single leaf node in the finish tree, but only one finish leaf may be connected to a start node.

Algorithm 3 *add_node*(node, root)**:** merges nodes that have the same output state and parts of the output stack

```
n ← root.children[node.state]
if n ≠ null then
    for all ch ∈ node.children do
        add_node(ch, n)
    n.start_nodes += node.start_nodes
else
    root.children[node.state] ← node
```

Algorithm 4 f_{plain}(node, c, root): changes the state of the current node based on character c

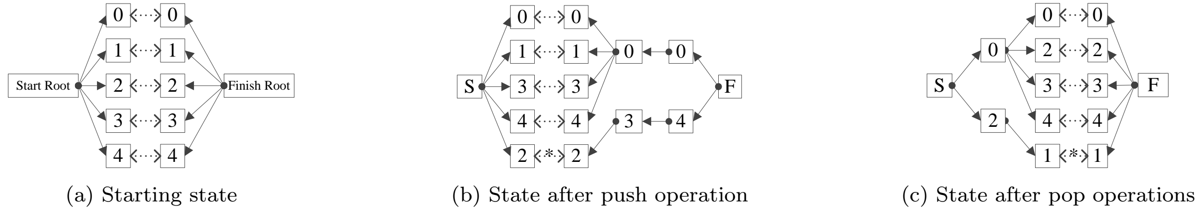
```
node.state ← δ_plain(node.state,c)
add_node(node, root)
```

The mappings are manipulated by operating on the first two levels of the finish tree. Nodes are added and removed from the root, and the rest of the tree grows from these operations. The start tree is only modified during a pop transition when there are no states on the finishing stack.

To process a symbol from the input tape, each node representing a finishing state (i.e. the first level of the finish tree) is considered in turn. For each node, the appropriate function f is invoked. The domains of the δ functions determine which function f is called. The function f_{plain} changes the state of a node but does not modify the structure of the tree; f_{push} creates a new node in the finish tree to represent the pushed state; and f_{pop} implements both popping an existing state from the stack and creating new entries if the popped state is not yet known. If there is a symbol on the stack for the popped state, the corresponding node is removed. If no such node exists, a new node is created in the start tree, thus pushing a state on the starting stack.

Each tree node has at most one child per symbol. If a transition causes two child nodes to have the same symbol, the nodes are merged using the function *add_node* defined in Alg. 3. Children of the combined node with the same symbol are merged recursively. The transition function F creates a new finish tree with only a root node. For each node in the first level of the old finish tree, F calls the corresponding function f. The new root node represents the mapping.

The f functions on the tree data structure are defined in Algs. 4–6. The functions f_{plain} and f_{push} only perform one operation: f_{plain} changes the state of a node, and f_{push} adds another node to the tree. The collapsing of nodes with the same state is delegated to the function *add_node*. The function f_{pop} has to support the cases when there are no states on the stack or when the popped state exists. The



(a) Starting state (b) State after push operation (c) State after pop operations

Figure 6: Example of trees representing mappings from Fig. 4

Algorithm 5 f_{push}(node, c, root): creates a new node for the pushed stack state

(next, pushed) ← δ_{push}(node.state, c)
node.state ← next
n ← `new` finish_node
n.state ← pushed, n.parent ← node
n.children ← node.children, node.children ← [n]
add_node(node, root)

Algorithm 6 f_{pop}(node, c, root): implements f_{pop} and $f_{unknown}$ depending on whether the state being popped exists

for all p where (node.state, c, p) ∈ dom(δ_{pop}) **do**
  next ← δ_{pop}(node.state, c, p)
  n ← node.children[p]
  if $n \neq null$ **then** // implements f_{pop}
    n.state ← next
    node.children.remove(n)
    add_node(n, root)
  else // implements $f_{unknown}$
    for all s in node.start_nodes **do**
      nf ← `new` finish_node, *ns* ← `new` start_node
      ns.parent ← s, *ns.state* ← p
      s.finish_node ← *null*, s.*children*[p] ← *ns*
      nf.start_nodes ← [*ns*], *nf*.state ← next
      add_node(*nf*, root)

first conditional branch assumes that the state exists and the corresponding node can be removed from the tree. The second branch handles pop transitions if the symbol on the stack is not yet known. This considers all possible symbols, adding a symbol to the input stack for each.

Example. We give an example of using this tree representation. Fig. 6a shows the starting state and consists of two trees of depth one. It connects each state to itself and corresponds to mapping (M2) from Fig. 4.

As symbols are pushed onto the finishing stacks, the finish tree grows to the right. For each push operation, a new set of nodes is placed between the parent and the rest of the tree. The updated tree, corresponding to mapping (M3), is shown in Fig. 6b. An asterisk (*) indicates a state that has matched an XPath query. These matches are stored in the leaves of the start tree.

States are popped by removing a node from the finish tree. The next two pop operations undo the push operations and return the data structure to the original state in Fig. 4. The final pop operation cannot pop an existing state. Instead nodes are created at the leaves of the start tree, resulting in the tree representing mapping (M5) shown in Fig. 6c.

5. EVALUATION

Our experimental evaluation has three aims: (1) to investigate processing throughput and execution times of our PP-Transducer approach for various types of benchmarking queries; (2) to demonstrate its scalability on many-core architectures; and (3) to explore the changes in processing throughput across a wider parameter set.

Prototype implementation. We implemented a C++ prototype version of the PP-Transducer approach. It uses standard optimisations to reduce the cost of data structure operations, such as a thread-local memory allocator. No platform-specific optimisations, such as custom scheduling or CPU-specific cache tuning, are used.

The PP-Transducer implementation operates by following the phases described in §3.2. The split phase splits XML data into chunks by skipping forward in the stream by a target chunk size (by default 10 MB) and searching sequentially for the next open angle bracket. Since only a few bytes are searched per chunk, the split phase only becomes a sequential bottleneck for small chunk sizes, or if the XML stream has a low tag density.

Once split, the chunks are assigned to parallel processors, each of which applies two transducers: the first transducer converts the chunk to a sequence of tag open and close events; the second realises the technique from §4, converting each chunk into a mapping of states.

After all parallel chunks have been processed, the mappings are joined to produce a list of all matches in the data. If a complex query was rewritten into multiple sub-queries, the filtering phase removes all matches for which the predicate conditions are not met. The matched XPath queries, along with the matched data, are stored in a vector and are output after all data has been processed.

A limitation of our implementation is that it assumes that an open angle bracket at the start of a chunk indicates the start of a tag. Although this means that it does not support XML data with comments or `CDATA` sections, this is not an intrinsic limitation of our approach. A PP-Transducer could be constructed on top of a simultaneous matching transducer [27], rather than a simple matching automaton [17], which supports comments and `CDATA` sections.

Comparison to other approaches. We compare PP-Transducers to three alternative XML processing techniques: parallelised versions of two popular XML stream processors (XMLTK and MxQuery); two XML parsers (PugiXML and Expat) to compare to only parsing the XML stream; and two XML-capable DBMSs (MonetDB and Sedna) to compare to highly-optimised DBMS engines. We chose PugiXML and XMLTK because they achieve the highest throughput for parsing and streaming, respectively.

XML stream processors. Both the XMLTK [3] and MxQuery [13] stream processors are single threaded. To exploit data parallelism for a fair comparison, we modify them to split the data into well-formed fragments, which are processed in parallel. Using the *Boost.regex* regular expression library [5], the processor searches the XML stream for clos-

Dataset	# XML tags	Max depth	Avg. depth	Avg. branch
XMark	334,095,625	13	5.55	3.67
Treebank	487,533,001	37	7.87	2.33
Twitter	275,931,225,001	9	3.95	15.94
$\text{Synth}_{d,b}$	*variable*	*d*	*variable*	*b*

Table 1: Properties of used XML datasets

Name	XPath query structure	# sub-queries	# sub-matches (1000s)	# matches (1000s)
A1	`/s/cs/c/a/d/t/k`	1	812	812
A2	`//c//k`	1	2,502	2,502
A3	`/s/cs/c//k`	1	2,502	2,502
A4	`/s/cs/c[a/d/t/k]/d`	3	4,712	531
A5	`/s/cs/c[descendant::k]/d`	3	6,402	1,070
A6	`/s/ps/p[pr/g and pr/age]/n`	4	12,766	644
A7	`/s/ps/p[ph or h]/n`	4	15,309	3,827
A8	`/s/ps/p[a and (ph or h) and (cc or pr)]/n`	7	22,967	1,440
B1	`/s/r/*/item[parent::sa parent::na]/name`	2	220	220
B2	`//k/ancestor::li/t/k`	3	25,502	6,225

Table 2: XPathMark rules used for query workload

ing tags that only occur in a well-defined location in the schema. To approximate fragments of a given target size, it skips forward by 10 MB, resulting in many elements in each fragment. We make the same assumption about no comments and `CDATA` sections, as above.

XML parsers. The input XML stream is first split into well-formed fragments, using the technique just described for XML stream processors. Each well-formed fragment is then queried in parallel, either by building a DOM tree for a fragment, or using a streaming SAX parser. For the DOM approach, we use the PugiXML parser and its built-in XPath library because it is one of the fastest C++ DOM parsers available [28]. We use the Expat SAX parser [11] to generate SAX events, followed by a transducer based on PP-Transducer (but without the out-of-order support) to execute the query.

XML DBMS. We use the MonetDB [4] and the Sedna [14] DBMSs to obtain a baseline comparison of how PP-Transducers compare to index-based XPath execution. MonetDB is a relational DBMS, which uses the Pathfinder module to map XML data to a relational data store. Sedna is a native XML DBMS without an underlying relational engine.

Both assume that the XML dataset is finite and can be loaded into the DBMS, which is not feasible for an infinite stream, or a dataset larger than the DBMS capacity. For example, we also tried to use Microsoft SQL Server as an example of a general-purpose DBMS with XML support, but it does not support XML data larger than 2 GB [26].

Datasets. Table 1 summarises the properties of the XML datasets used in our experiments. We choose the *Treebank* [30] and *XMark* [29] datasets to reflect two typical XML schemas. The Treebank schema has a root element with a large number of direct children, allowing well-formed fragments to be identified using opening and closing tags. In contrast, the XMark dataset has only six direct children off the root, each containing different amounts of data, making it hard to split them into well-formed fragments of equal size. To produce large datasets, we use a scaling factor of 200 for XMark and replicate the Treebank dataset 200 times, resulting in sizes of 22 GB and 17 GB, respectively.

As a typical streaming workload, we use a 44 GB dataset created by capturing 14 million tweets from the *Twitter* public streaming API, stored in the Twitter XML format. The Twitter data is shallow compared to XMark and Treebank but does contain recursive elements: a status element can contain a retweeted status which is itself a complete status. We replicate the Twitter dataset 250 times, resulting in a total data volume of more than 11 TB.

Since none of the datasets allows control over average node-depths or branching factors, we use an XML generator [34] to generate *synthetic* datasets by selecting random nodes from the Treebank schema. The generator also permits data items to be generated with sizes following a log-normal distribution with an adjustable scale factor.

XPath queries. For the XMark dataset, we use *XPathMark* [15] because it is designed to evaluate the performance of XPath query processors using a realistic query set. As listed in Table 2, we use the entire A query set and the first two queries from the B query set. As described in §3.2, PP-Transducers only support a subset of XPath directly. The first three A set queries are run unchanged, and the others are translated to sub-queries, which execute simultaneously. The results of the sub-queries are then processed to create the final result. Parent and ancestor queries are performed through query rewriting, as described by Olteanu [25]. When a query is split, the table shows the number of sub-queries and the number of occurred matches.

For the Treebank and synthetic datasets, random queries of the form `//a/b/c/d` are generated, in which each tag is one of the elements in the descriptive part of the tree. This emulates the search for data with a specific phrase structure. By default, 20 such Treebank rules are executed as a single query set, except when indicated otherwise.

For the Twitter dataset, we use queries to filter tweets that contain some user-specified metadata and report the location of the metadata in the XML Twitter stream. As an example, we use the query `//status/coordinates/coordinates` to select all tweets with embedded coordinates.

Experimental set-up. All experiments are performed on a 2.1 GHz AMD quad-socket machine, with 16 CPU cores per socket, for a total of 64 cores, running Linux Fedora 16. The machine has 128 GB of RAM, which allows all input XML data to be pre-loaded into memory. This removes the effects of disk IO and caches DBMS indices completely. For the timed experiments, the XPath query results are collected in memory and discarded, again to avoid IO operations.

5.1 Throughput

We first compare the throughput of PP-Transducer to other XML stream processors and parsers. Fig. 11 shows the throughput of different XML processing approaches executing a single query on the Twitter dataset. The single-threaded performance of PP-Transducer is comparable to all other approaches and more than twice that of Expat and MxQuery. MxQuery does not operate in a streaming mode when multiple independent queries are run simultaneously so these results are not included.

The parallelised versions of PugiXML and XMLTK are faster than PP-Transducer with a single thread of execution, but at 64 CPU cores PP-Transducer has 1.8 times the throughput of PugiXML and 10 times the throughput of XMLTK. The scaling behaviour of PugiXML is

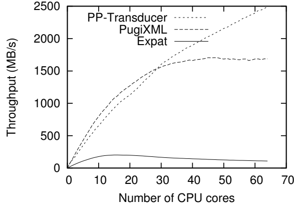


Figure 7: Scalability with different XPath processors

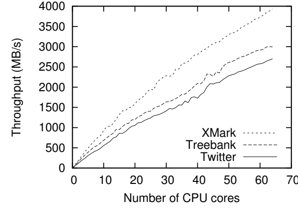


Figure 8: Scaling behaviour under different datasets

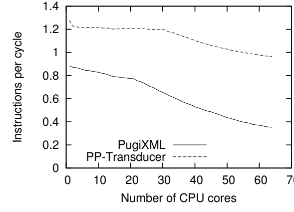


Figure 9: Reduction of IPC with more CPU cores

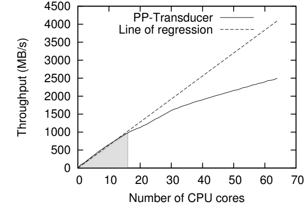


Figure 10: Line of regression for throughput per CPU core

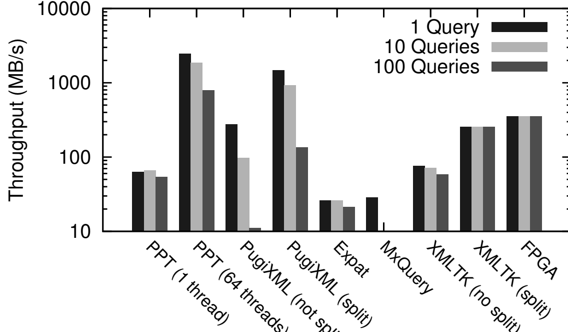


Figure 11: Throughput of querying the Twitter dataset

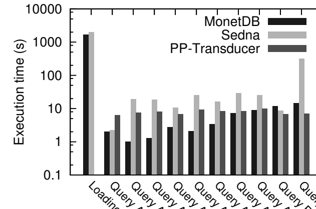


Figure 12: Execution times in comparison to DBMSs

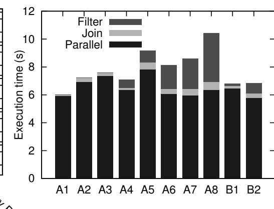


Figure 13: Breakdown of query execution time for PP-Transducer

explored in the next section; XMLTK is limited by the sequential overhead of splitting the data.

For a single query on the 11 TB Twitter dataset, PP-Transducer achieves an average throughput of 2.5 GB/s. To the best of our knowledge, this result is nearly an order of magnitude greater than the highest recorded throughput for streaming XPath processing reported in the literature [24].

To understand the query efficiency of PP-Transducer, we compare its execution time to the MonetDB and Sedna XML DBMSs. Both MonetDB and Sedna have a lengthy loading phase, during which the XML dataset is parsed and indexed, and they are unable to load the full 11 TB dataset.

Fig. 12 shows that, while the XML DBMSs can perform the queries more quickly, the time taken to load the data is several orders of magnitude greater than using PP-Transducer. MonetDB has a query execution time that is 20 times faster than PP-Transducer, but only after completing a load phase of around half an hour.

The execution time depends on the structure of the query. Queries that use the descendent axis (i.e. //) are less efficient when executed with PP-Transducer and Sedna, but more efficient with MonetDB. In the case of PP-Transducer, these rules add more transitions to the transducer, reducing the convergence of states. A DBMS is able to pre-compute all of these relationships while building the index, and MonetDB uses such an optimisation—queries A2 and A3 execute faster because they are shorter and need fewer comparisons of node relationships.

When used in a streaming fashion, the total throughput of an XML database is limited to the speed at which the indexing can be performed. For MonetDB, this is 13 MB/s—two orders of magnitude slower than PP-Transducer.

All three approaches require more time to process queries with predicates. In the case of PP-Transducer, the breakdown of execution times in Fig. 13 shows that this is mostly caused by the sequential post-processing step. Its running time is proportional to the number of matches for each subquery and is independent of the operations in the predicate.

5.2 Scalability

We compare the scalability of PP-Transducers in terms of CPU cores to the parallel versions of the PugiXML and Expat parsers, because they exhibit the best scaling, and any XML stream processor must at least parse the stream.

Fig. 7 shows the change in throughput for the Treebank dataset as the number of parallel threads on CPU cores is increased from 1 to 64. We use a set of 5 concurrent queries, as described in the previous section, and achieve a processing throughput of 2.5 GB/s. PP-Transducer exhibits strictly linear scaling up to 16 cores, then close to linear scaling from 16 to 64 cores—the interaction of the threads with the machine's memory hierarchy prevents perfect scaling. This scaling behaviour is similar on all datasets evaluated, as shown in Fig. 8, however the Twitter dataset give lower total performance due to the shallow tree structure.

The comparatively low throughout of the Expat parser in Fig. 7 is due to its memory allocator, which is shared among all threads. An increased number of utilised CPU cores leads to significant lock contention.

Up to 25 cores, PugiXML outperforms PP-Transducer because the overhead of managing state mappings is greater than that of constructing DOM trees. The increased memory bandwidth and greater cache utilisation needed for parallel DOM tree construction, however, cause the throughput of PugiXML to plateau for more than 30 cores. Fig. 9 shows how the *instructions per clock cycle* (IPC) decrease with larger CPU core counts for PugiXML; PP-Transducer does not suffer the same loss in IPC because its largest data structures are the transition tables, which are shared between threads. The thread-local tree data structures are small enough to fit into the L2 cache shared between pairs of CPU cores.

When investigating the performance of PP-Transducer under different parameters in §5.3, we concentrate on the region in which its scaling behaviour is linear. This is indicated in Fig. 10 as the shaded area up to 16 CPU cores—up to this point the line of regression closely matches the observed data. Beyond 16 cores, the scaling is sub-linear due to the machine's memory system.

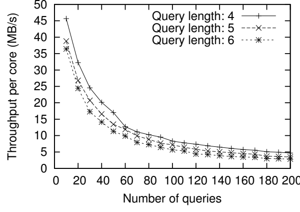


Figure 14: Throughput reduction for larger sets of queries

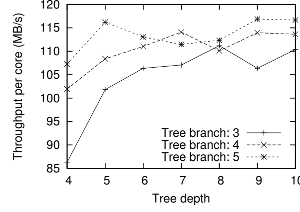


Figure 15: Improved throughput for deeper and wider XML trees

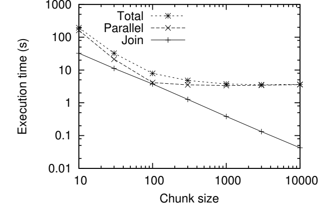
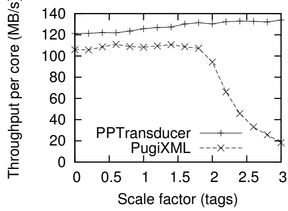
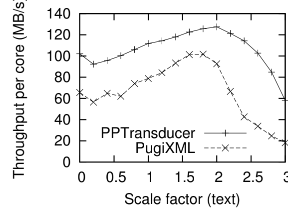


Figure 16: Execution time decrease for larger chunk sizes

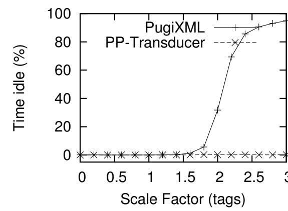


(a) Changing number of tags

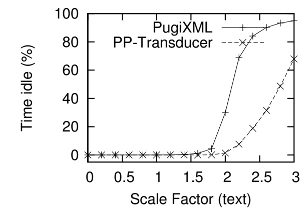


(b) Changing size of text

Figure 17: Decreased throughput as data skew increases



(a) Changing number of tags



(b) Changing size of text

Figure 18: Increased idle time as data skew increases

5.3 Parameter Exploration

In our approach, faster state convergence leads to greater throughput. The rate at which a large number of possible states converge to a smaller number depends on the properties of the processed data and the executed queries. Next we explore various parameters affecting performance.

XML tree shape. We consider the effect of the shape of the XML tree on the throughput of PP-Transducer. We use the $\text{Synth}_{\text{d,b}}$ dataset with different tree depths d and branching factors b, and 20 queries of the form `//a/b/c/d`.

We present the results in Fig. 15. As the XML tree depth and branching factor increases, the overhead also decreases, with a corresponding increase in throughput. The increased throughput is caused by the rate of state convergence, which depends on the shape of the XML data. The intuition is that a deeper tree has more potential for convergence than a shallow one—for convergence to occur, there must be a series of push operations, which is more common in deep trees. A larger branching factor also increases efficiency because it leads to an increase in the average depth of the tree.

Beyond a tree depth of 7 and a branching factor of 4, the throughput stops increasing, as the mappings for each chunk have reached the maximum level of convergence. The Treebank datasets therefore exhibits greater throughput than XMark, despite a larger number of concurrent queries.

XPath query set. We examine how throughput per CPU core is affected by the number of simultaneous XPath queries and their lengths. In Fig. 14, we show that the throughput of PP-Transducer quickly decreases as the number and length of queries increases. This is caused by two effects: (1) more states from a larger query set also increase the number of potential states, thus reducing the convergence rate; and (2) the throughput decreases further due to the large transition lookup tables, which incur more cache misses. Our approach trades the query parallelism in other automata-based stream processing systems for data parallelism on a smaller number of queries.

XML chunk size. Next, we explore how the XML chunk size affects the throughput of PP-Transducer using the same set-up as in the previous scaling experiments in §5.2.

Fig. 16 shows that the execution time increases quickly as smaller chunk sizes are used, with the lowest time achieved for chunks larger than 1 MB. Smaller chunk sizes cause the parallel and sequential phases of PP-Transducer to take more time. In the parallel phase, the smaller number of bytes per chunk reduces convergence, thus increasing the number of CPU cycles per byte. More chunks also increase the time required to join the mappings together sequentially in order to produce the final result. The parallel phase has the lowest execution time for 100 kB chunks, and the sequential phase becomes negligible at chunk sizes above 1 MB.

Distribution of data sizes. The Treebank dataset has a large number of data items under the root element. Parsing XML data in parallel using an off-the-shelf parser requires splitting it into well-formed XML fragments, and our current PP-Transducer implementation requires finding an open angle bracket to split the data. Both may become a sequential bottleneck as larger items appear in the XML data.

To explore this effect, we generate synthetic XML datasets based on the Treebank set of tags, with the size of each item generated according to a log-normal distribution. Adjusting the scale factor of the distribution allows us to introduce skew that creates more large items. To assess the impact of the splitting operation, we measure the proportion of time that threads spend idle waiting for work, in addition to the processing throughput.

We scale the sizes of each data item in different ways. In Fig. 17a, we increase the size of each item by making the XML tree broader and deeper. The larger items cause the splitting operation to take longer for PugiXML, which requires well-formed fragments. Beyond a scale factor of 2, the throughput starts to decrease as threads spend a significant amount of time in an idle state, waiting for a suitable split point to be found. PP-Transducer does not require well-formed fragments and thus exhibits negligible idle time, even for large scale factors.

In Fig. 17b, we maintain the same tree depth but vary the size of the text between tags. The results show that, for both PP-Transducer and PugiXML, the cost of splitting becomes significant as the scale factor increases. However,

PP-Transducer always performs better than PugiXML because the average distance to the next tag is shorter than to the end of a complete data item.

Fig. 18 confirms our hypothesis that the percentage of time threads spend idle directly correlates with the observed reduction in throughput. The Treebank dataset has a scale factor of less than one, which means that the throughput plateau of PugiXML in Fig. 7 cannot be attributed to lock contention.

6. RELATED WORK

We will now discuss previous work on parallel XML processing, with a particular focus on parallel parse tree construction and query execution using data-parallel methods.

XML stream processing. Instead of data parallel execution, previous research on XML stream processing focused on improving the expressiveness of XPath querying and executing large numbers of concurrent queries efficiently [16].

There are three general approaches for evaluating queries on streaming XML data: (i) automata-based techniques compile a set of rules into an automaton, which executes the query [9, 37]; (2) in contrast, array-based techniques, as used in TurboXPath [21], do not require the construction of an automaton but store pointers to elements in the XML tree and execution state; and (3) finally, stack-based algorithms, such as Twig2Stack [8], compactly represent a large number of partial matches that occur when queries with many predicates are executed.

Techniques that are not automata-based can provide better performance when queries contain many predicates, because they transform the XML byte stream into a specialised internal form. When this expressiveness is not required, automata have been shown to scale to 10,000s of queries [9]. Automata-based approaches also map more naturally to the speculative execution model that we use to achieve data-parallelism and do not require any sequential transformation of the input data. We leave an exploration of the applicability of our out-of-order techniques to Twig2Stack-style processing to future work.

A common assumption in prior work is that a large number of XPath queries are executed over an XML stream. Y-Filter [9] and XMLTK [3] execute thousands of small queries in parallel. They handle the state explosion of the subset construction by creating the DFA lazily. Zhang et al. [37] propose to execute multiple states in a non-deterministic finite automaton in parallel. The stream is still parsed sequentially but each starting state of the automaton is handled by a different thread. In contrast, PP-Transducer is designed to exploit data parallelism with a small set of queries, utilising a large number of CPU cores to process incoming XML streams at a high rate.

Commercial stream processing engines such as Microsoft StreamInsight and IBM InfoSphere Streams can operate on XML through the use of an XML adapter. Once the XML stream has been converted to an internal representation, it is possible to perform expressive and time-varying queries. However, the XML adapter introduces a throughput bottleneck because it only processes the stream sequentially.

XML stream processing systems that exploit FGPAs such as the one proposed by Moussalli [24] have focussed on using the parallelism in the FPGA fabric to execute a large number of queries simultaneously. These approaches process the stream sequentially, which limits the system throughput to around 300 MB/s—an order of magnitude less than our performance for a small number of queries. SCBXP [10] describe an approach to consuming multiple bytes of the XML stream in a single clock cycle but it is limited to an average of two bytes per cycle due to limited width of the content addressable memories used.

Parallel XML tree parsing. Early techniques for parallel parse tree construction relied on a sequential pre-parse phase, which splits XML data into well-formed fragments that can be processed in parallel [23]. While such an approach can scale to a small number of CPU cores, the sequential bottleneck of the pre-parse phase becomes an issue for larger numbers of cores.

To scale beyond this limit, techniques have been proposed that parallelise the pre-parse phase using concurrent matching transducers. Pan et al. [27] demonstrate that this can scale well, but it has not yet been shown how to integrate the output of an out-of-order pre-parser efficiently with a full parser in order to produce a complete tree. We avoid this problem by not requiring a separate parser.

An alternative approach is to infer the current state of the parser based on heuristics in the XML data, such as the beginning and end of comments [35]. This avoids the pre-parse phase at the expense of having to re-parse XML fragments if the initial guess is incorrect. All of the above techniques assume that the entire XML parse tree can be represented into memory and are therefore not applicable to large streaming XML datasets.

Parallel XML querying. There are several techniques for executing XPath queries that can exploit data parallelism after constructing an in-memory parse tree. The simplest is to rewrite an individual XPath query into multiple sub-queries. These sub-queries are executed in parallel and their results are joined in a sequential step [6]. This achieves a good speed-up because the sub-queries are typically simpler, resulting in reduced execution times even before parallelism is introduced. In order to achieve a large degree of parallelism, it is necessary to partition the tree in addition to the queries. This requires building the tree before partitioning, which is necessarily a sequential step.

An alternative method by Lui et al. [22] uses a parallel structured join algorithm. It partitions the XML elements and joins the results of inspecting each element in parallel. While the query and join operations are parallelisable, constructing the required data structures is a sequential step.

All of the above approaches rely on having well-formed fragments of the XML parse tree, which can be processed independently and joined together. In contrast, PP-Transducers operate on arbitrarily framed XML chunks, thus reducing the cost of splitting the data into work units in order to achieve increased scalability, at the cost of supporting a lower number of concurrent queries.

Out-of-order automata. The out-of-order execution of automata has been explored by the networking community in the context of intrusion detection across traffic streams. Johnson et al. [20] propose to reduce memory consumption when executing regular expressions by avoiding the buffering of packets that arrived out-of-order. In their automaton, they construct mappings of starting to finishing states for each packet, which are typically substantially smaller than the processed data and do not grow with the size of the data.

However, they only consider DFAs and do not provide an efficient algorithm for the construction of mappings.

Chandramouli et al. [7] consider the problem of processing disordered streams, e.g. when data items are missing from a stream. They focus on the loss of a few data items by reasoning about the possible effects of missing items on the final result. For each missing item, a potential execution path is constructed. Their automata model, however, is less expressive than a pushdown transducers. In addition, it does not scale as the number of missing data items increases—this prevents the use of their approach in our scenario because out-of-order XML processing quickly creates hundreds of thousands of missing events for each processor.

7. CONCLUSION

We have described PP-Transducers, a new automata-based execution model to query XML data streams using pushdown transducers in a data parallel fashion. We have given a formal description of the transducer's operation and described how it can be implemented efficiently. Our experimental results demonstrate the scalability of this approach: it manages to achieve near linear scaling up to 64 CPU cores and an overall processing throughput of more than 2.5 GB/s.

As part of future work, we plan to improve the ability of our approach to handle a larger number of concurrent XPath queries and enhance their expressiveness. One possible way of improving both is to design a hybrid approach that combines a parallel transducer with an index-based query engine. The index generation can be done in parallel by the transducer, with more sophisticated queries being answered by the query engine. In addition, we want to explore whether techniques based on subset construction on the transducer [27] are applicable. The goal is to make the amount of processing performed per character independent of the size of the processed XML chunks.

Acknowledgements.. This work was supported by a PhD CASE Award funded by the Engineering and Physical Sciences Research Council (EPSRC) and BAE Systems Detica.

8. REFERENCES

[1] V. Agarwal, D. Bader, et al. Faster FAST: Multicore acceleration of streaming financial data. *CS - Research and Development*, 23(3-4):249–257, 2009.

[2] R. Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009.

[3] I. Avila-Campillo, T. J. Green, et al. XMLTK: An XML toolkit for scalable XML stream processing. Technical report, PlanX, 2002.

[4] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery. In *SIGMOD*, pages 479–490, 2006.

[5] Boost Library. `http://boost.org/libs/regex`.

[6] R. Bordawekar, L. Lim, et al. Statistics-based parallelization of XPath queries in shared memory systems. In *EDBT*, pages 159–170, 2010.

[7] B. Chandramouli, J. Goldstein, and D. Maier. High-performance dynamic pattern matching over disordered streams. *VLDB*, 3(1):220–231, 2010.

[8] S. Chen, H. Li, et al. Twig2Stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In *VLDB*, pages 283–294, 2006.

[9] Y. Diao, P. Fischer, M. Franklin, et al. YFilter: Efficient and scalable filt. of XML doc. In *ICDE*, pages 341–342, 2002.

[10] F. El-Hassan and D. Ionescu. SCBXP: An efficient hardware-based XML parsing techn. In *SCPL*, pages 45–50, 2009.

[11] Expat Parser. `http://expat.sourceforge.net`.

[12] L. Fegaras, C. Li, U. Gupta, and J. Philip. XML query optimization in map-reduce. In *WebDB*, 2011.

[13] P. Fischer et al. XQuery: A lightweight, full-featured XQuery engine. `http://mxquery.org`, 2013.

[14] A. Formichev, M. Grinev, and S. Kuznetsov. Sedna: A native XML DBMS. *SOFSEM*, pages 272–281, 2006.

[15] M. Franceschet. XPathMark: Functional and performance tests for XPath. In *XQuery Implementation Paradigms*. Dagstuhl, 2007.

[16] N. Francis, C. David, and L. Libkin. A direct transation from XPath to nondet. automata. In *Workshop on Foundations of Data Management*, pages 350–361, 2011.

[17] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *ICDT*, pages 173–189, 2003.

[18] 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`.

[19] H. Jiang, H. Lu, et al. XParent: An efficient RDBMS-based XML database sys. In *ICDE*, pages 335–336, 2002.

[20] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Monitoring regular expressions on out-of-order streams. In *ICDE*, pages 1315–1319, 2007.

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

[22] L. Liu, J. Feng, et al. Parallel structural join algorithm on shared-memory multi-core systems. *Web-Age Information Management*, pages 70–77, 2008.

[23] W. Lu, K. Chiu, and Y. Pan. A parallel approach to XML parsing. In *Grid Computing*, pages 223–230, 2006.

[24] R. Moussalli, M. Salloum, et al. Accelerating XML query matching through custom stack generation on FPGAs. In *HPEAC*, pages 141–155. 2010.

[25] D. Olteanu, H. Meuss, et al. XPath: Looking forward. In *XML-Based Data Manag. and Multim. Eng.*, pages 892–896, 2002.

[26] S. Pal, V. Parikh, et al. XML best practices for Microsoft SQL Server 2005. Technical report, Microsoft, 2004.

[27] Y. Pan, Z. Zhang, and K. Chiu. Simul. transducers for data-parallel XML parsing. In *IPDS*, pages 1–12, 2008.

[28] PugiXML. `http://pugixml.org/benchmark/`.

[29] A. Schmidt, F. Waas, et al. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002.

[30] Treebank dataset. `http://goo.gl/c6O3X`, 2013.

[31] W3C XPath. `w3.org/TR/1999/REC-xpath-19991116`.

[32] Wikipedia database. `http://goo.gl/dSFy`, 2013.

[33] X. Wu and D. Theodoratos. A survey on XML streaming evaluation techniques. *VLDB Journal*, pages 1–26, 2012.

[34] XML Generator. `http://goo.gl/QWvUJQ`.

[35] C. You and S. Wang. A Data Parallel Approach to XML Parsing and Query. *IEEE HPCC*, pages 520–527, 2011.

[36] Y. Zhang, Y. Pan, and K. Chiu. Speculative p-DFAs for parallel XML parsing. In *HiPC*, pages 388–397, 2009.

[37] Y. Zhang, Y. Pan, and K. Chiu. A parallel XPath engine based on concurrent NFA exec. In *IPDS*, pages 314–321, 2010.