Hindawi
Scientific Programming
Volume 2021, Article ID 5559941, 16 pages
https://doi.org/10.1155/2021/5559941

Research Article

Pipelined XPath Query Based on Cost Optimization

Rongxin Chen,[1,2] **Zongyue Wang**,[1] **and Yuling Hong**[1]

[1]*Computer Engineering College, Jimei University, Xiamen 361021, China*
[2]*Digital Fujian Big Data Modeling and Intelligent Computing Institute, Xiamen 361021, China*

Correspondence should be addressed to Zongyue Wang; wangzongyue@jmu.edu.cn

Received 16 February 2021; Accepted 7 May 2021; Published 27 May 2021

Academic Editor: Cristian Mateos

XPath query is the key part of XML data processing, and its performance is usually critical for XML applications. In the process of XPath query, there is inherent seriality between query steps, which makes it difficult to parallelize the query effectively as a whole. On the other hand, although XPath query has the characteristics of data stream processing and is suitable for pipeline processing, the data flow of each query step usually varies a lot, which results in limited performance under multithreading conditions. In this paper, we propose a pipelined XPath query method (PXQ) based on cost optimization. This method uses pipelined query primitives to process query steps based on relation index. During pipeline construction, a cost estimation model based on XML statistics is proposed to estimate the cost of the query primitive and provide guidance for the creation of a pipeline phase through the partition of query primitive sequence. The pipeline construction technique makes full use of available worker threads and optimizes the load balance between pipeline stages. The experimental results show that our method can adapt to the multithreaded environment and stream processing scenarios of XPath query, and its performance is better than the existing typical query methods based on data parallelism.

1. Introduction

Semistructured data [1] are very common in the field of web application and information integration. As a powerful semistructured data description tool, XML has become a standard of data storage and exchange. XPath [2] is a specific language to find information in XML data, which is the basis of XML data processing. Its performance is directly related to the processing ability of XML applications. With the popularity of multicore computing environment, it has become a common way to take advantage of multithreaded parallel computing to improve the performance of application. The parallel XPath query technology for multicore computing has been developed in recent years. From the perspective of the parallel pattern, the implementation technology of XPath parallelization includes data parallelism [3–5], task parallelism [6], and pipeline parallelism [7].

According to the semantics of XPath, the query is mainly the process of locating the nodes of the XML tree according to the node relation conditions. After obtaining the evaluation result of the preceding query step, the current query step is evaluated, and then the result is passed to the following query step. The processing between query steps has inherent seriality; hence, it is difficult to parallelize the XPath query effectively as a whole. On the other hand, the query process has the characteristics of data flow processing [8], which is suitable for the pipelined processing style. However, the evaluation results of each query step often differ greatly, so there is a load imbalance between query steps. Due to the simplicity of implementation, most of the existing XPath parallelization methods adopt data parallelism or task parallelism. In data parallelism, the data objects processed by the query step are partitioned before parallel processing while the query step is partitioned and then evaluated in parallel in task parallelism. These methods increase the parallel opportunity by partitioning the data or query step. The components of the partition are independent of each other and can be processed in parallel, and thus, they belong to horizontal parallelization [9]. However, in such kind of parallelization, as long as there is a dependency between the query steps of XPath, it is still serial processing in essence. Pipelining is vertical parallelization which takes producer-

consumer mode as the working characteristics of the pipeline stage. If the query steps of XPath are organized in the pipeline, the whole evaluation process of XPath can be conducted in vertical parallelization. However, due to the load imbalance between query steps in XPath queries, the performance of pipelining is greatly limited. In pipeline construction, cost estimation is usually used to provide guidance for pipeline optimization [10] to deal with load imbalance.

In order to adapt to the data stream processing scenarios of XPath and make full use of multicore resources to improve query performance, this paper proposes a pipelined XPath query method (PXQ) based on cost optimization. Our method has the following features: (1) pipelined query primitives are used to process the query steps based on relation index, and query primitives are easy to be concatenated to support the generation of large pipeline stages; (2) the cost estimation model based on XML statistics is introduced to guide the generation of the pipeline stage to achieve load balancing; and (3) flexible pipeline construction is adopted. The number of pipeline stages is determined according to the number of available worker threads, which can make full use of available threads and avoid thread context switching during pipeline execution. Compared with the XPath query method based on horizontal parallelization, the experiment shows that our method has better performance.

The remainder of the paper is structured as follows. Section 2 introduces the related work. Section 3 presents some technical background of this paper, including XML coding, relation index, and XPath evaluation. Section 4 describes the PXQ method in detail, including preprocess, pipelined query primitive, cost model, and query pipeline creation. In Section 5, comparative experiments were carried out. The last section gives the conclusion.

2. Related Work

This paper focuses on the performance of XPath query in a multicore computing environment. The related work involves parallel XPath query technology and XML stream processing technology.

In the parallel XPath query technology, the most common method is based on data parallelism. Typical studies in this area include those carried out by Bordawekar et al. [3, 11]. In [3], three strategies for parallelization of XPath queries are proposed: data partitioning, query partitioning, and hybrid partitioning, which are verified manually. Reference [11] further introduces a cost estimation model to guide the partitioning work to support the automatic generation of parallel query plans. After that, Sato et al. [5] carried out XPath parallelization research on the new XML query platform BaseX [12] on the basis of Bordawekar's work. In the previous work [4], we proposed an efficient XPath evaluation method pM^2 based on relation matrix. The pM^2 method consists of two data-parallel execution phases including the construction of the matrix and the evaluation of query primitives. Query execution is carried out by relation matrix search, and parallel query primitives are executed in data parallelism. Our method in this paper also adopts the similar index mechanism.

XML is semistructured, and XPath needs to support rich query semantics. These complex factors lead to many challenges in XML data stream processing. XPath query contains basic operations such as forward axes, backward axes, and predicate. The unity of various operations should be considered when processing XML data stream. Barton et al. [13] gave a method to support backward axes through constraint transformation, and various operations can be processed in stream. Kwon et al. [14] used sequencing twig patterns to support the evaluation of various value-based predicates in stream processing. In different application scenarios, XML data stream processing has different technical characteristics. Multiquery is a common scenario in server-client application. Peter et al. [15] proposed a path navigation mechanism based on NFA, and multiple queries can be mapped to NFA for simultaneous processing. Similarly, Kim et al. [16] transformed the collection of XPath expressions into multiple FSA-based query indexes for parallel processing of XML streams. In addition, they combined an in-memory MapReduce model to handle twig pattern joins over XML streams, further improving the overall efficiency. Fegaras et al. [17] presented an XML algebra, which adopts caching stream data mechanism and query decorrelation technology to adapt to multicast XML stream processing scenarios. Kim et al. [18] used GPU to accelerate XML stream processing with multiple queries. In their method, both XML query and XML stream are transformed into matrix indexes, and the Boolean operations of bit-AND are used for query processing to adapt to GPU computing. In the aspect of workflow application scenarios, Zinn et al. [19] presented a workflow processing method for distributed environment, which processes XML data collections in a pipeline style and optimizes it by static type inference. XProc [20] is a pipeline description language for programming users proposed by W3C. Lopes and Carrico [21] gave XML pipeline rules based on template, which supports automatic generation of XProc description pipeline.

It is usually necessary to utilize a cost estimation model for pipeline construction and scheduling optimization in stream processing. The technologies of stream processing in the traditional relational database and workflow field are useful for optimizing XML stream processing. For instance, Spiliopoulou et al. [22] combined the cost estimation of relational operators in the pipeline when optimizing parallel query with a large number of join operations. In [23], a lightweight measurement for the operator was used to optimize scheduling and allocate CPU resources to the pipeline stage. In [24], a runtime-aware adaptive schedule mechanism is proposed to minimize the operator processing latency and the latency difference between different tasks in long-running stream processing applications. Jiang et al. [25] used random variables to model the pipeline execution times to construct the optimal pipeline under required timing constraints. In terms of XML cost estimation, Bordawekar et al. [11] proposed a statistics-based estimation method to estimate the computation cost of XPath query

steps. Cardinality and selectivity are used to estimate the evaluation results of axis operation and predicate operation of XPath, respectively. Aboulnaga et al. [26] optimized the query by estimating the selectivity of XML path expression, and the selectivity is calculated according to the summarized information stored in path trees and Markov tables. The evaluation cost of XPath is not only related to the specific evaluation method but also related to the encoding of XML data stream and the application of index. Therefore, the cost estimation model for XPath needs to be considered from several related aspects.

3. Preliminaries

3.1. XML Encoding. XML data are a kind of semistructured data, which need specific encoding to facilitate access. The commonly used XML encodings include region encoding [27] and Dewey order encoding [28]. The XML encoding processed by our method is a region encoding represented by a 6-tuple. For example, the region encoding of node u is < *ID, nodeType, tagName, begin, end, level* >, where *id* is the node ID. As the node ID is unique, u can be denoted by the ID value of the node. *NodeType* is the node type. In this paper, we consider the most frequently used element and attribute nodes, so *nodeType* ∈ {ELEMENT, ATTRIBUTE}; *tagName* is the tag name of the node; *begin* is the starting position of the node in the document; *end* is the end position of the node; *level* is the level value of the node in the document tree. The XML document in Figure 1(a) contains 12 element nodes and 2 attribute nodes. The corresponding document tree is shown in Figure 1(b). The tag name of each node is marked in the circle, and different nodes with the same tag name are distinguished by numbers. The string beside the circle is a simplified region encoding, which contains the node ID value, the document begin position of the node, the document end position of the node, and the level value. For example, the code "3 [21, 47, 2]" of node C1 indicates that the ID value of the node is 3, and the position starts from the 21st byte to the end of the 47th byte in the XML document, and the level value is 2.

3.2. Relation Index. There are various possible relations between any two XML nodes u and v. The relations include parent (denoted as "PA"), child (denoted as "CH"), ancestor (denoted as "AN"), descendant (denoted as "DS"), attribute (denoted as "AT"), preceding-sibling (denoted as "PS"), following-sibling (denoted as "FS"), preceding (denoted as "PP"), and following (denoted as "FF"). The relation between two nodes is directional, and the relation types can be inferred from one direction to another. In this paper, the relation between two XML nodes u and v refers to the relation from node u to node v. We specify that node u is prior to node v in XML document order, that is, the ID value of u is less than that of v. Therefore, the relation type is limited to $r \in \{\text{DS, CH, AT}\}$, which represents the relation of descendant, child, and attribute, respectively. Generally, queries can be effectively optimized based on a specific XML document index [29, 30]. Our method uses a relation index to support the efficient processing of XPath queries.

Definition 1 (relation index). It refers to the storage structure that records the effective relation between XML nodes. An index entry is represented by a tuple as $\langle u, v, r_{u \longrightarrow v} \rangle$, which indicates the unique relation type between node u and v is r and $r \in \{\text{DS, CH, AT}\}$. The relation index of node u refers to the relation index set of node u and all its subsequent nodes v in document order that have DS, CH, or AT relation with node u. To save the storage space, the node ID is used to represent the node, and the index entry is simplified to $\langle \text{id}_v, r_{u \longrightarrow v} \rangle$; then, the relation index for node u is a tuple set of all v nodes corresponding to node u, which is described as $\mathcal{I}_u = \cup_j \{\langle \text{id}_{vj}, r_{u \longrightarrow vj} \rangle\}$. For an entire XML document with N nodes, the index space is expressed as $\mathcal{I} = \cup_{i \in N} \mathcal{I}_{ui} = \cup_{i \in N} \{\cup_{j \in N} \{\langle \text{id}_{vj}, r_{ui \longrightarrow vj} \rangle\}\}$.

The node relation index for the XML case in Figure 1 is shown in Table 1. The index entry is described as < node ID, relation type >.

3.3. XPath Evaluation. Path expression is the basic form of XPath, which describes the sequence of each query step. In XPath syntax, the "/" symbol is used to divide query steps. Each step locates nodes in the XML tree according to the current node. Each step may contain the following components: (1) axis operation: used to traverse the XML tree according to certain node relation; (2) node test: used to filter node name, with "*" to indicate no filtering; and (3) predicate: or branch query, which is used to filter nodes by their attributes, child node characteristics, or position. These components can be combined into complex XPath expression to control traversal and node selection in the XML tree. This paper uses the subset {/, //, *, @, []} of XPath, which covers the core functions of XPath. Its syntax is shown in Table 2.

Since XPath completes the query by evaluating each query step, the improvement of the evaluation performance depends on the implementation of the query step. To get better performance, our method is based on relation index and evaluates the query step through node relation search.

4. Proposed Method

4.1. The Framework of PXQ. The method proposed in this paper consists of two main phases: one is the preprocess phase and the other is the pipeline construction and query phase. As shown in Figure 2, the preprocess phase includes the parsing of XML documents, the creation of relation index, and the acquisition of XML statistics. After the XML document is parsed, the region encoding of XML nodes will be obtained. The relation index information of all XML nodes is obtained by index creation. According to the need of cost estimation, the statistical information of XML is obtained in this phase. The pipeline construction and query phase includes four steps: primitive extraction, cost estimation, pipeline stage generation, and pipeline stage evaluation. The first three steps are pipeline construction and the

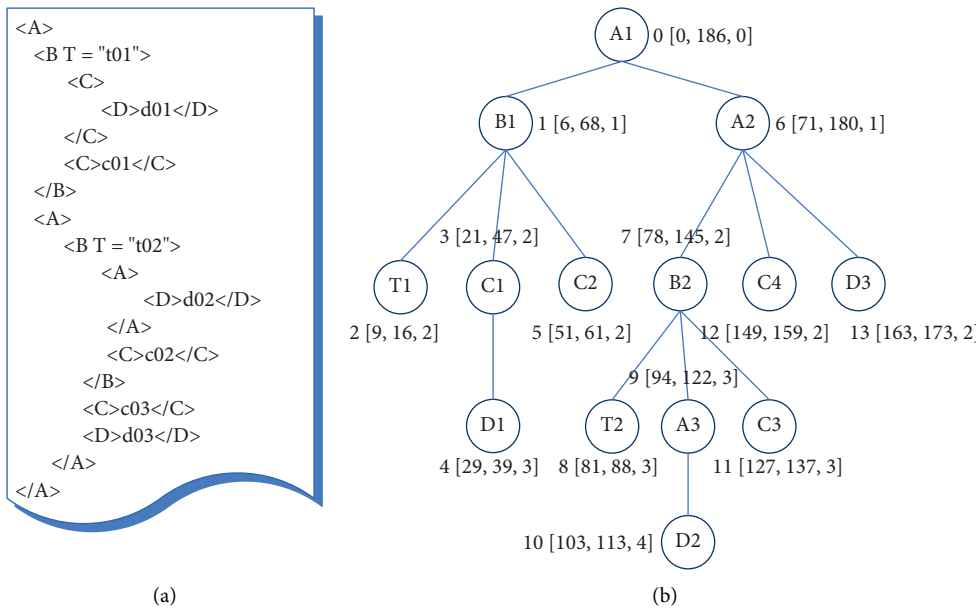

Figure 1: XML encoding. (a) XML document and (b) document tree with region encoding.

Table 1: Relation index.

Node	Index entry
A1	<1, CH>, <3, DS>, <4, DS>, <5, DS>, <6, CH>, <7, DS>, <9, DS>, <10, DS>, <11, DS>, <12, DS>, <13, DS>
B1	<2, AT>, <3, CH>, <4, DS>, <5, CH>
C1	<4, CH>
A2	<7, CH>, <9, DS>, <10, DS>, <11, DS>, <12, CH>, <13, CH>
B2	<8, AT>, <9, CH>, <10, DS>, <11, CH>
A3	<10, CH>

Table 2: Syntax of XPath subset.

Expression name	Syntax
Path expression	Path::= Step('/' Step)*
Query step	Step:: = Axis Tag | Axis Tag '[' Pred ']'
Axis	Axis::= PA | CH | AN | DS | AT | PS | FS | PP | FF
Tag for node test	Tag::= String | *
Predicate	Pred::= Path | Pred 'or' Pred | Pred 'and' Pred | 'not' '(' Pred ')' | Num

last step is pipeline query. Primitive extraction refers to generating a set of query steps represented by pipelined query primitives according to the input XPath expression. The cost estimation step is to calculate the cost of each query step according to XML statistics and cost estimation model. According to the cost estimation results and the number of available threads, the primitive sequence is further partitioned and then pipeline stages are generated. The pipeline query plan is composed of pipeline stages. After pipeline construction, the thread is assigned to each pipeline stage for evaluation and obtaining the query results.

The symbols and their meanings are shown in Table 3. The superscript with D indicates that the symbol has the statistical property of XML document.

4.2. Pipeline Mechanism in PXQ. The basic idea of PXQ is to use pipelined query primitives as the basic units of execution. By partitioning pipelined phases and allocating work threads, pipelined parallel processing is carried out. In the process of pipeline construction, the cost estimation is used to guide the partition of query primitive sequence, so as to realize the optimization of load balancing and the effective utilization of threads in each stage. It involves technical points such as pipelining query primitive, pipeline stage, and cost estimation.

4.2.1. Pipelined Query Primitive. Pipelined query primitives are the basic processing units for query evaluation in the PXQ method. Each query step of XPath corresponds to a

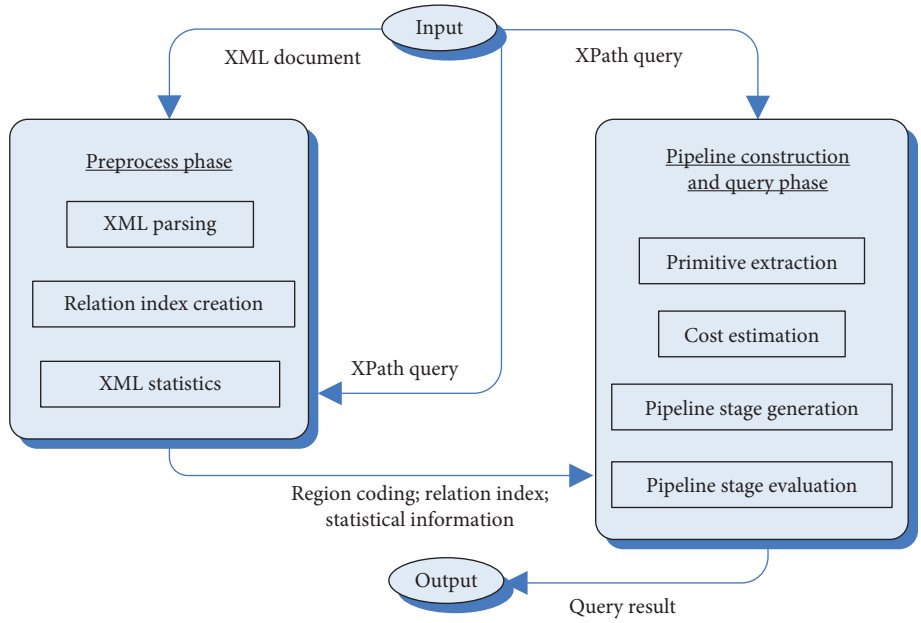

Figure 2: Processing framework of PXQ.

Table 3: Symbols and their meanings.

Symbol	Meaning
$\mathscr{E}_u$	The region encoding of an XML node u. u is generally represented by the ID value.
$\mathscr{I}_u$	The relation index of an XML node u (see Definition 1)
M_{Ep}	The sequence of pipelined query primitives corresponding to the XPath expression *Ep*. If *Ep* is a query step, it represents a query primitive.
N_τ^D	The number of all nodes with tag name τ in XML document D.
$N_{\tau*}^D$	The number of all nodes in the subtrees whose roots are nodes with tag name τ in XML document D.
$N_{\tau 1\theta\tau 2}^D$	The number of nodes with relation θ and tag name $\tau 2$ contained in the subtrees whose roots are nodes with tag name $\tau 1$ in XML document D. That is, the number of nodes satisfying the relation condition $\tau 1\theta\tau 2$ in XML document D.
$V_{\tau 1\theta\tau 2}$	Containing-mean (see Definition 3)
$P_{\tau 1\theta\tau 2}$	Filtering-rate (see Definition 4)

pipelined query primitive, and the entire XPath query expression corresponds to a sequence of pipelined query primitives. PXQ uses query primitives to realize basic query function. By looking up the relation value in the relation index, the query results that meet the query conditions are returned. The pipelined query primitives in PXQ include nonfilter primitives and filter primitives as shown in Figures 3(a) and 3(b), respectively. They correspond to the general axis operation and predicate operation in XPath. The solid connection line in the figure represents the data transmission path of the actual evaluation process, while the dotted connection line is the logical data transmission path, which is actually transmitted in the form of unit data (see Definition 2). Figure 3(a) shows a nonfilter primitive with one input and two outputs, where the dotted output is a logical output that exists only if the successor primitive is a filter one. A filter primitive shown in Figure 3(b) has two inputs and two outputs, and the dotted line output exists when the successor primitive is a filter one. In order to avoid the synchronous waiting between primitives and improve the throughput of pipelining, PXQ introduces the mechanism of using unit data to transfer historical values.

Definition 2 (unit data). It refers to the data parameters transferred in the process of pipeline, which is represented by $\mathscr{U}$. $\mathscr{U}$ consists of two data fields: one is the current field represented by $\mathscr{U} \cdot \mathscr{E}$, which is used to record the current node information, and the other is the history field represented by $\mathscr{U} \cdot \mathscr{E}^h$, which is a set used to record the historical node information. The specific history information with index position i can be obtained by using $\mathscr{U} \cdot \mathscr{E}_i^h$.

The size of the history field in unit data determines the depth of nested predicates that can be processed. The space of history field can be reused according to the specific query to improve the storage efficiency. We set the node ID value of $\mathscr{U} \cdot \mathscr{E}$ to -1 as the end flag of execution. Adjacent query primitives process unit data through accessing the blocking queues in primitives. The basic process is to take a unit data from its own blocking queue when the primitive is executed. After processing, the result is put into the blocking queue in the successor primitive in the form of unit data. The

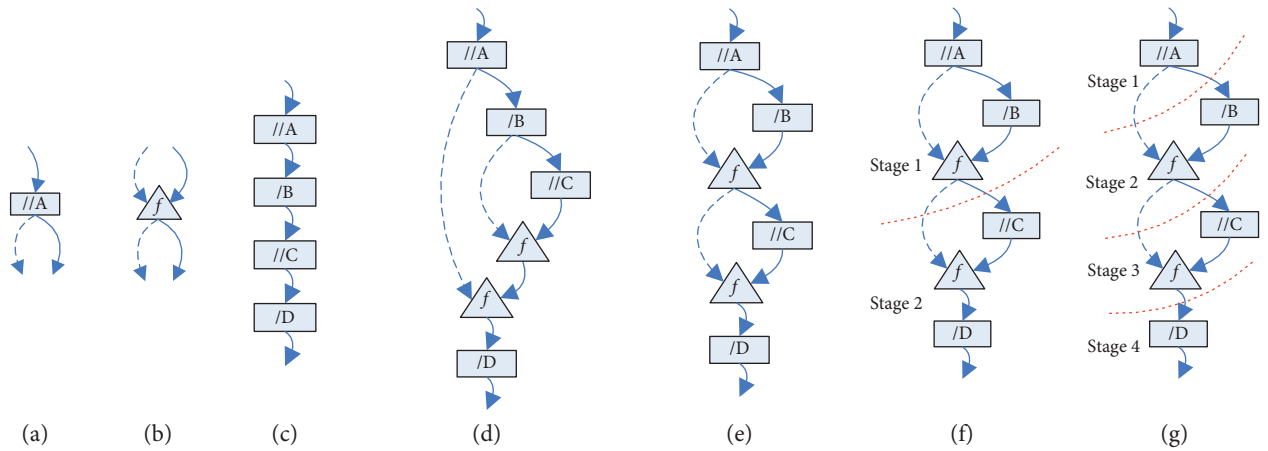

Figure 3: Pipelined query primitives and pipeline structures. (a) Nonfilter primitive, (b) filter primitive, (c) structure of "//A/B//C/D," (d) structure of "//A[./B[.//C]]/D," (e) structure of "//A[./B][.//C]/D," (f) two-stage pipeline, and (g) four-stage pipeline.

workflow of the pipelined query primitive is shown in Figure 4(a).

4.2.2. Pipeline Stage and Pipeline. The pipeline stage of PXQ is an executable functional component, which contains one or more pipelined query primitives and communicates with each other through blocking queues. The pipelining stage is to wrap pipelined query primitives so as to facilitate the allocation of worker thread. When there are more than two query primitives in a pipeline stage, the primitives are organized in a pipelining way, which is convenient for processing in a consistent way.

Figure 4(b) shows the workflow of a pipelining stage with three pipelined query primitives where each primitive execution is a subworkflow as shown in Figure 4(a). Before the execution of the first primitive, it is necessary to detect whether the unit data contain the pipeline end tag. Multiple primitives are organized in nested loops in order, and the condition of exit processing is that the blocking queue in each primitive is empty.

Since the query primitives are pipelined, they are easy to be concatenated. The pipeline phase is composed of query primitives and then concatenated to form a complete pipeline query plan. Figures 3(c)–3(e) show the pipeline structure of several typical XPath query expressions, such as serial axis operation, nested predicate, and juxtaposed predicate. Figure 5 shows the time-space diagram of the pipeline shown in Figure 3(g). The pipeline consists of four stages, in which stage S1 and stage S2 each contain two query primitives. Four instances are processed in the figure. The pipelined query primitives transfer unit data by operating the blocking queue in the primitives. There is a producer-consumer relationship in the precursor and successor primitives. Under the multithreaded condition, there are race conditions because the queues in the query primitives in the adjacent stages are operated by different threads. To ensure the correctness of execution results, the blocking queue in primitive is utilized for synchronization.

4.2.3. Cost Estimation. To make full use of worker threads and avoid thread context switching, it is necessary to allocate a worker thread for each pipeline phase. When the number of primitives exceeds the number of available threads, the requirement of allocating one thread for each phase can be satisfied by partitioning primitive sequence. Load balance of each stage should be considered to reduce the idle waiting between stages and improve the efficiency of pipeline. However, cost estimation is a necessary means for load balancing. An XPath query expression may contain multiple query primitives, and the cost of primitives is generally different. The cost of primitives includes not only the computation cost but also the communication cost of adjacent primitives, which is related to the amount of data transferred. Because PXQ uses the query method based on the node relation, the amount of data is reflected in the number of nodes.

4.3. Preprocess. The preprocess phase prepares the data to be queried and the statistics needed for pipeline construction and query. This phase takes the parsing process of XML documents as the processing framework. During XML parsing, the index is created and the statistical information is obtained synchronously. The details are as follows:

(1) *XML Parsing.* The XML parsing in preprocess adopts event-based parsing method similar to SAX [31]. The result of parsing is a sequence of XML nodes described by region encoding and arranged in document order.

(2) *Relation Index Creation.* During XML parsing, at the end of each XML node, the relation between the node and the XML nodes it contains is calculated. Since the encoding of XML nodes is generated in document order during XML parsing, the region encodings of XML nodes can be used to calculate the relation and record the results in the relation index.

(3) *XML Statistics.* At the beginning of preprocess, the input XPath query expression is analyzed. According to the tag names and node relations involved in the

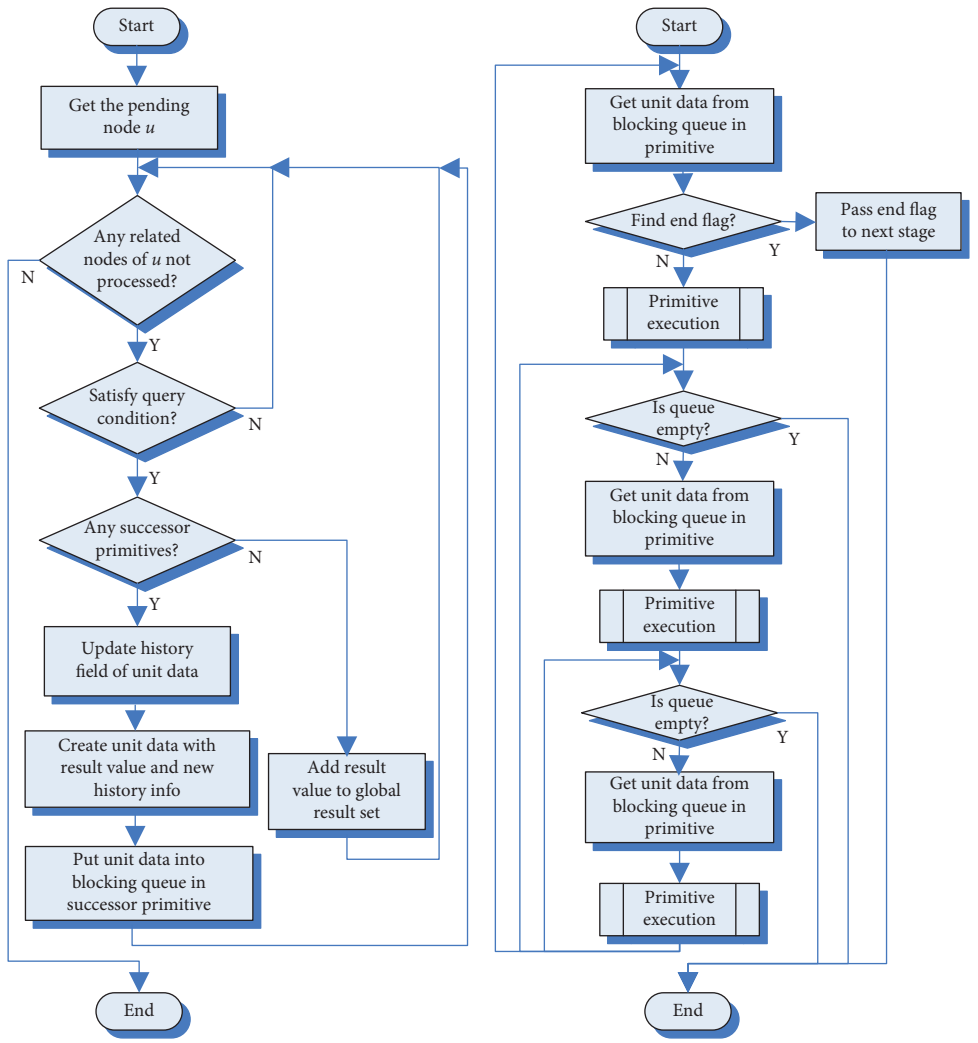

Figure 4: Key workflows of PXQ. (a) Workflow of pipelined query primitive and (b) workflow of the pipeline stage.

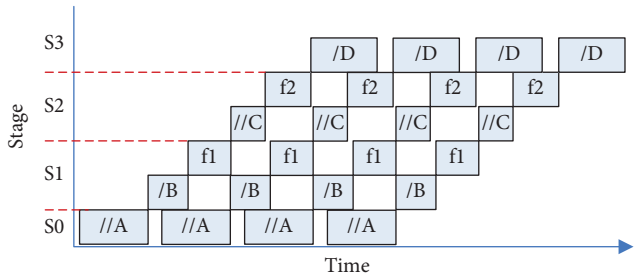

Figure 5: Time-space diagram of pipelining.

analysis results, the statistical variables used in cost estimation are extracted. These variables are accumulated and calculated in the process of XML parsing. Generally, the types of statistical variables include N^D_τ, $N^D_{\tau *}$, and $N^D_{\tau 1 \theta \tau 2}$. The formulas for the last two variables are $N^D_{\tau *} = \sum_{u \in D} N^u_{\tau *}$ and $N^D_{\tau 1 \theta \tau 2} = \sum_{u \in D} N^u_{\tau 1 \theta \tau 2}$, respectively, where $N^u_{\tau *}$ is the number of all nodes in the subtree whose root is node u with tag name τ, and $N^u_{\tau 1 \theta \tau 2}$ is the number of nodes satisfying the condition of relation $\tau 1 \theta \tau 2$. In the last part of preprocess, containing-mean $V_{\tau 1 \theta \tau 2}$ and filtering-rate $P_{\tau 1 \theta \tau 2}$ are further calculated.

The whole procedure of preprocess is described in Algorithm 1.

Line 1 in Algorithm 1 analyzes the input XPath query expression to get information about statistical variables. The analysis result A includes tag name set $A.\tau 1$, $A.\tau 1^*$, and $A.\tau 2$ and relation condition string set $A.\tau 1 \theta \tau 2$. Lines 4–22 show that during parsing, when the XML node start tag is encountered, a new XML node item is created; when the XML node end tag is encountered, in addition to updating the document end position information of the XML node item (line 9), the main work is to accumulate statistical variables and create index entries. Lines 7 and 9 carry out XML parsing to obtain the region encoding of the current node; lines 10~16 are used for statistical variables accumulation; lines 17–20 are used to calculate the relation between the

Input: XML document $\mathscr{D}$, XPath query $\mathscr{P}$.
Output: XML node region coding $\mathscr{E}$, relation index $\mathscr{I}$, XML statistics $\mathscr{S}$.
(1) $A \leftarrow$ Analyze($\mathscr{P}$);
(2) id $\leftarrow 0$, level $\leftarrow 0$;//record the node ID and the level value of the current node
(3) $N^D_\tau \leftarrow 0$, $N^D_{\tau^*} \leftarrow 0$,$N^D_{\tau 1 \theta \tau 2} \leftarrow 0$;//initializing statistical variables
(4) **while**(!EOF($\mathscr{D}$))
(5) p $\leftarrow$ Parsing($\mathscr{D}$);
(6) **if**(p is StartElement)//when a header tag is encountered
(7) u $\leftarrow$ id, $\mathscr{E}_u \leftarrow$ CreateNewNode(p), $\mathscr{E} \leftarrow \mathscr{E} \cup \{\mathscr{E}_u\}$, id $\leftarrow$ id + 1, level $\leftarrow$ level+1;
(8) **if**(p is EndElement)//when the tail tag is encountered
(9) UpdateNode($\mathscr{E}_u$.end);
(10) **if** (($\mathscr{E}_u$.tagName = τ)$\wedge\tau \in A.\tau 1$)$N^D_\tau \leftarrow N^D_\tau + 1$;
(11) **foreach** node id $k \in (u, \text{id})$
(12) **if**(($\mathscr{E}_k$.tagName = τ)$\wedge\tau \in A.\tau 1^*$)$N^D_{\tau^*} \leftarrow N^D_{\tau^*} + 1$;
(13) **if**(($\mathscr{E}_k$.nodeType = ELEMENT)$\wedge$($\mathscr{E}_k$.tagName = τ_2)$\wedge$($\mathscr{E}_k$.level-1 = level)) θ = "/";
(14) **else if**(($\mathscr{E}_k$.nodeType = ELEMENT)$\wedge$($\mathscr{E}_k$.tagName = τ_2)) θ = "//";
(15) **else if**(($\mathscr{E}_k$.nodeType = ATTRIBUTE)$\wedge$($\mathscr{E}_k$.tagName = τ_2)$\wedge$($\mathscr{E}_k$.level-1 = level)) θ = "@";
(16) **if**($\tau 1 \theta \tau 2 \in A.\tau 1 \theta \tau 2$) $N^D_{\tau 1 \theta \tau 2} \leftarrow N^D_{\tau 1 \theta \tau 2} + 1$;
(17) **if**(($\mathscr{E}_u$.level = $\mathscr{E}_k$.level-1)$\wedge$($\mathscr{E}_k$.nodeType = ELEMENT)) $r_{u \longrightarrow v} \leftarrow$ CH;
(18) **else if**(($\mathscr{E}_u$.level $\neq$ $\mathscr{E}_k$.level-1)$\wedge$($\mathscr{E}_k$.nodeType = ELEMENT)) $r_{u \longrightarrow v} \leftarrow$ DS;
(19) **else if**(($\mathscr{E}_u$.level = $\mathscr{E}_k$.level-1)$\wedge$($\mathscr{E}_k$.nodeType = ATTRIBUTE)) $r_{u \longrightarrow v} \leftarrow$ AT; ,
(20) $\mathscr{I}_u \leftarrow \mathscr{I}_u \cup \{\langle k, r_{u \longrightarrow v}\rangle\}$;
(21) level $\leftarrow$ level-1;
(22) $\mathscr{I} \leftarrow \mathscr{I} \cup \mathscr{I}_u$;//end while
(23) $V_\tau \leftarrow N^D_{\tau^*}/N^D_\tau$, $V_{\tau 1 \theta \tau 2} \leftarrow N^D_{\tau 1 \theta \tau 2}/N^D_{\tau 1}$, $P_{\tau 1 \theta \tau 2} \leftarrow N^D_{\tau 1 \theta \tau 2}/N^D_{\tau 1^*}$;
(24) $\mathscr{S} \leftarrow \{V_{\tau 1}, V_{\tau 1 \theta \tau 2}, P_{\tau 1 \theta \tau 2}\}$;
(25) **return** $\{\mathscr{E}, \mathscr{I}, \mathscr{S}\}$;

Algorithm 1: Preprocess ($\mathscr{D}$, $\mathscr{P}$).

current node and the nodes contained in the subtree whose root is current node, and then the index entry is created. Line 23 calculates the containing-mean and filtering-rate required according to the analysis result A. Since the statistics accumulation and index entry creation are based on region encodings of the parsed nodes, there is no need to scan the XML document again.

4.4. Pipelined Query Primitive Design. In PXQ, nonfilter primitives are the implementation of axis operations in XPath. For example, the primitive *GetDescendant* is used to obtain the descendants of nodes, and *GetChild* is used to obtain the children of nodes. Filter primitives are the implementation of predicate operations in XPath, including basic filter primitive *FilterInput1byInput2* and variants of filter primitives, such as filter primitives with *AND* condition, filter primitive with *OR* condition, and filter primitive with *NOT* condition. M_{Ep} is used to represent the pipelined query primitive corresponding to the XPath expression *Ep*. For example, $M_{\tau 1//\tau 2}$ stands for *GetDescendant* $\{\tau 1, \tau 2\}$, where $\{\tau 1, \tau 2\}$ records the tag names for node test; $M_{\tau 1[./\tau 2]}$ stands for *FilterInput1byInput2* $\{\tau 1, \tau 2\}$.

The work of pipelined query primitives involves basic computation and pipelined communication, as shown in Algorithm 2. The algorithm enumerates the processing procedure of two typical primitives. One is the nonfilter primitive *GetDescendant*, which is used to find descendants, and the other is the filtering primitive *FilterInput1byInput2*, which is used for basic predicate operations.

In Algorithm 2, the unit data $\mathscr{U}^{\text{in}}$ contains an input XML node information, which is obtained from the evaluation result of the preceding pipeline stage. Lines 2–11 and 12–20 are the pipelined processes of a nonfilter query primitive *GetDescendant* and a filter query primitive *FilterInput1byInput2*, respectively. The algorithm shows that each query primitive has a similar pipelining procedure. For nonfilter primitives, the node test conditions are set first (lines 3 and 4). Then get the ID value of the current pending node u (line 5) from the input unit data. Next, according to the index of node u, all relational nodes of node u are checked one by one (lines 6–10). For nodes that meet the condition of descendant relation (line 7), if there are following query primitives, the history field of input unit data will be updated first, then the local evaluation result and the updated history field are used to create a unit data, and finally, the unit data are put into the queue in the following query primitive (line 9). If the query primitive is the last primitive in the primitive sequence, the current node information will be directly output to the global result (line 10). For filter query primitives, the node ID of the node to be filtered is obtained from the history field of the input unit data (line 13), and then all the relation nodes of u are checked one by one according to the index of node u (lines 14–19). When the filtering conditions are met, the check is terminated immediately (line 19) after completing the process similar to that of the nonfilter primitive (lines 16–18).

Input: primitive name pName, input unit data $\mathcal{U}^{\text{in}}$, blocking queue in following query primitive $\mathcal{Q}$, tag name tName, input position index i, and output position index j in history record of unit data, region encoding $\mathcal{E}$, relation index $\mathcal{I}$, and global result $\mathcal{E}^{\text{out}}$.

Output: none. The local results are processed in the algorithm: either put into the queue or output directly to the global result.

```
(1)  switch(pName)
(2)    case "GetDescendant":
(3)    if(tName = "*") nodeTest ← true;
(4)    else nodeTest ← false;
(5)    u←U^in · E.id;
(6)    foreach index s ∈ I_u
(7)      if((nodeTest = true ∨ E_s.id.tagName = tName)∧(s.r = DE ∨ s.r = CH))
(8)        if(Q ≠ ∅)
(9)        U^in · E^h_j ← E_s.id; U^out ← CreateUnit (E_s.id, U^in · E^h); Q.enqueue (U^out);
(10)       else E^out ⟶ E^out ∪ {E_s.id};
(11)   return
(12)   case "FilterInput1byInput2":
(13)   u←U^in · E^h_i.id;
(14)   foreach index s ∈ E_k
(15)     if((s.id = U^in · E.id)∧(s.r = DE ∨ s.r = CH))
(16)       if(Q ≠ ∅)
(17)         U^in · E^h_j ← E_s.id; U^out ← CreateUnit(E_s.id, U^in · E^h); Q.enqueue(U^out);
(18)       else E^out ← E^out ∪ {E_s.id};
(19)       break;
(20)   return
```

Algorithm 2: PipedPrimitive (pName, $\mathcal{U}^{\text{in}}$, $\mathcal{Q}$, tName, i, j, $\mathcal{E}$, $\mathcal{I}$, $\mathcal{E}^{\text{out}}$).

4.5. Cost Estimation in Pipeline. The query primitives in PXQ make use of the relation index to process the query efficiently. According to the characteristics of this query method, we introduce a cost model based on XML statistics. The model estimates the number of nodes in the query result of the pipelined query primitive generated by the XPath query expression and further estimates the execution time cost of each query step. To estimate the number of nodes in the query result, two definitions are introduced as follows.

Definition 3 (containing-mean). It refers to the average number of nodes that satisfy the node relation condition in all subtrees of XML document. Use the symbol V for containing-mean. For $V_{\tau 1\theta\tau 2}, \theta \in \{/, //, @\}$, it refers to the average number of nodes with relation θ and tag name $\tau 2$ (i.e., satisfying the $\tau 1\theta\tau 2$ relation condition) under each node with tag name $\tau 1$ (i.e., in the subtree whose root is the node with tag name $\tau 1$) in the XML document D. The formula is

$$V_{\tau 1\theta\tau 2} = \frac{N^D_{\tau 1\theta\tau 2}}{N^D_{\tau 1}}. \tag{1}$$

For the node with tag name τ, the containing-mean V_τ can be simplified from the definition of $V_{\tau 1\theta\tau 2}$. There is $V_\tau = V_{\tau\theta^*} = N^D_{\tau\theta^*}/N^D_\tau = N^D_{\tau^*}/N^D_\tau$. In addition, the root node is a special case, since it contains all the other nodes, and its containing-mean is $V_{\tau r} = N$.

Definition 4 (filtering-rate). It refers to the ratio of nodes that satisfy the node relation condition in all subtrees of XML document. Use the symbol P for filtering-rate. For $P_{\tau 1\theta\tau 2}, \theta \in \{/, //, @\}$, it refers to the ratio of the number of nodes with relation θ and tag name $\tau 2$ (i.e., satisfying the $\tau 1\theta\tau 2$ relation condition) among all nodes under each node with tag name $\tau 1$ (i.e., in the subtree whose root is the node with tag name $\tau 1$) in the XML document D. The formula is

$$P_{\tau 1\theta\tau 2} = \frac{N^D_{\tau 1\theta\tau 2}}{N^D_{\tau 1^*}}. \tag{2}$$

The number of evaluation result nodes of each query primitive is estimated according to the formulas given in Propositions 1 and 2. The symbol $\mathbb{N}(M)$ means to estimate the number of result nodes of query primitive M. The number of result nodes is represented by n. Specifically, n_τ represents the number of current input nodes with tag name τ. $n_0, n_1, n_2, \ldots$, represent the number of result nodes of each query step.

Proposition 1. *For nonpredicate query step expression $\tau 1\theta\tau 2, \theta \in \{/, //, @\}$, the corresponding query primitive is $M_{\tau 1\theta\tau 2}$, where $\tau 1$ and $\tau 2$ are node tag names, respectively. For this type of query primitive, the number of result nodes can be estimated by equation (3), where $n_{\tau 1}$ is the number of current input nodes with tag name $\tau 1$:*

$$\mathbb{N}\left(M_{\tau 1\theta\tau 2}\right) = n_{\tau 1} \times V_{\tau 1\theta\tau 2}. \tag{3}$$

Proof. In XML document D, the number of all nodes satisfying the condition of relation $\tau 1\theta\tau 2$ is $N^D_{\tau 1\theta\tau 2}$, and the ratio of the number of current input nodes with tag name $\tau 1$ to the number of all nodes with tag name $\tau 1$ in the whole document is $n_{\tau 1}/N^D_{\tau 1}$. From the probability of occurrence, the

product of the two is the estimated number of tag nodes satisfying the condition, i.e., $\mathbb{N}(M_{\tau 1\theta\tau 2}) = N^D_{\tau 1\theta\tau 2} \times (n_{\tau 1}/N^D_{\tau 1})$. According to Definition 3, $\mathbb{N}(M_{\tau 1\theta\tau 2}) = n_{\tau 1} \times V_{\tau 1\theta\tau 2}$ can be further deduced.

For the root node of XML document, τr is used to represent the tag name of the root node. The estimation of query expression $\tau r\theta\tau 2$ is as follows: according to equation (3), there is $\mathbb{N}(M_{\tau r\theta\tau 2}) = n_{\tau r} \times V_{\tau r\theta\tau 2} = n_{\tau r} \times N^D_{\tau r\theta T2}/N^D_{\tau r}$, where $n_{\tau r} = 1$ and $N^D_{\tau r} = 1$, so $\mathbb{N}(M_{\tau r\theta\tau 2}) = N^D_{\tau r\theta\tau 2}$. □

Proposition 2. *For predicate query step expression $\tau 1[Ep]$, the corresponding query primitive is $M_{\tau 1[Ep]}$, where $\tau 1$ is the node tag name and Ep is the path expression. For this type of query primitive, the number of result nodes can be estimated by equation (4), where $\tau 1Ep$ is a new path expression composed of tag $\tau 1$ and path expression Ep in predicate symbol.*

$$\mathbb{N}\left(M_{\tau 1[\mathrm{Ep}]}\right) = \mathbb{N}\left(M_{\tau 1\mathrm{Ep}}\right) \times \frac{N^D_{\tau 1}}{N^D_{\tau 1^*}}. \tag{4}$$

Proof. According to the semantics of predicate, expression $\tau 1[\mathrm{Ep}]$ is to find the nodes in the current input nodes with tag name $\tau 1$ that meet the filtering condition of query $\tau 1\mathrm{Ep}$. The estimation of its results can be calculated by the product of the number of nodes number $n_{\tau 1}$ and a ratio ρ satisfying the filtering condition, i.e., $\mathbb{N}(M_{\tau 1[\mathrm{Ep}]}) = n_{\tau 1} \times \rho$. When the predicate is evaluated, the filter condition $\tau 1\mathrm{Ep}$ is calculated first, and the estimation of this part can be obtained by $\mathbb{N}(M_{\tau 1\mathrm{Ep}})$. This estimate is based on the current input node, while for all $\tau 1$ nodes in the XML document, it needs to be further divided by the estimation of any case under the condition of node $\tau 1$, i.e., $\mathbb{N}(M_{\tau 1\theta^*})$. So there is $\rho = \mathbb{N}(M_{\tau 1\mathrm{Ep}})/\mathbb{N}(M_{\tau 1\theta^*})$. According to Proposition 1, there is $\mathbb{N}(M_{\tau 1\theta^*}) = n_{\tau 1} \times V_{\tau 1\theta^*}$; therefore, the following formula is derived:

$$\begin{aligned}
\mathbb{N}\left(M_{\tau 1[\mathrm{Ep}]}\right) &= n_{\tau 1} \times \frac{\mathbb{N}\left(M_{\tau 1\mathrm{Ep}}\right)}{\mathbb{N}\left(M_{\tau 1\theta^*}\right)} \\
&= n_{\tau 1} \times \frac{\mathbb{N}\left(M_{\tau 1\mathrm{Ep}}\right)}{n_{\tau 1} \times V_{\tau 1\theta^*}} \\
&= \mathbb{N}\left(M_{\tau 1\mathrm{Ep}}\right) \times \frac{N^D_{\tau 1}}{N^D_{\tau 1^*}}.
\end{aligned} \tag{5}$$

□

Corollary 1. *The result nodes number of predicate $\tau 1[\theta\tau 2]$ can be estimated by equation (6). The equation is an estimation of the simple predicate with only one path step, and it is also a special case of Proposition 2.*

$$\mathbb{N}\left(M_{\tau 1[\theta\tau 2]}\right) = n_{\tau 1} \times P_{\tau 1\theta\tau 2}. \tag{6}$$

Proof. According to Propositions 1 and 2, combined with Definitions 3 and 4, when the expression Ep is $\theta\tau 2$, the derivation is as follows:

$$\begin{aligned}
\mathbb{N}\left(M_{\tau 1[\theta\tau 2]}\right) &= \mathbb{N}\left(M_{\tau 1\theta\tau 2}\right) \times \frac{N^D_{\tau 1}}{N^D_{\tau 1^*}} \\
&= N^D_{\tau 1\theta\tau 2} \times \left(\frac{n_{\tau 1}}{N^D_{\tau 1}}\right) \times \frac{N^D_{\tau 1}}{N^D_{\tau 1^*}} \\
&= n_{\tau 1} \times \frac{N^D_{\tau 1\theta\tau 2}}{N^D_{\tau 1^*}} \\
&= n_{\tau 1} \times P_{\tau 1\theta\tau 2}.
\end{aligned} \tag{7}$$

□

According to the characteristics of pipelined query primitives, the execution time cost of primitive can be considered from the following two aspects: (1) *Evaluation Cost*. The basic evaluation of primitives is carried out through relation search and condition detection, and the processing time is related to the number of relations of nodes. (2) *IO Cost*. Each pipelined query primitive has a blocking queue to store the local evaluation results in the form of unit data. The access of queue elements reflects the IO cost, which is related to the number of result nodes. Since the cost estimation here is only related to the primitive itself, the communication cost between primitives does not need to be considered. The symbol $\mathbb{C}(M)$ is used to estimate the cost of primitive M. The general formula of cost estimation of pipelined query primitive is

$$\mathbb{C}\left(M_{\mathrm{Ep}}\right) = n_{\mathrm{in}} \times c_{\mathrm{ior}} + n_{\mathrm{out}} \times c_{\mathrm{iow}} + R_{\mathrm{in}} \times c_{\mathrm{eval}}, \tag{8}$$

where n_{in} and n_{out} are the number of input nodes and output nodes, respectively; c_{ior}, c_{iow}, and c_{eval} are the unit time of reading data, writing data, and basic evaluation; and R_{in} is the number of input relations. According to equation (8), the cost estimation formula of nonpredicate query primitives is

$$\mathbb{C}\left(M_{\tau 1\theta\tau 2}\right) = n_{\mathrm{in}} \times c_{\mathrm{ior}} + \mathbb{N}\left(M_{\tau 1\theta\tau 2}\right) \times c_{\mathrm{iow}} + n_{\mathrm{in}} \times V_{\tau 1} \times c_{\mathrm{eval}}, \tag{9}$$

which can be further deduced as

$$\mathbb{C}\left(M_{\tau 1\theta\tau 2}\right) = n_{\mathrm{in}} \times c_{\mathrm{ior}} + n_{\mathrm{in}} \times V_{\tau 1\theta\tau 2} \times c_{\mathrm{iow}} + n_{\mathrm{in}} \times V_{\tau 1} \times c_{\mathrm{eval}}. \tag{10}$$

For predicate query primitives, the cost estimation formula is

$$\mathbb{C}\left(M_{\tau 1[Ep]}\right) = n_{\mathrm{in}} \times c_{\mathrm{ior}} + \mathbb{N}\left(M_{\tau 1[Ep]}\right) \times c_{\mathrm{iow}} + \mathbb{N}\left(M_{\tau 1\mathrm{Ep}}\right) \times V_{\tau 1} \times c_{\mathrm{eval}}. \tag{11}$$

For example, in the "//S[./VBP]//PP/VP" query, the cost estimation process is analyzed as follows: according to the analysis results of XPath query expression, the filtering-rate to be calculated is $P_{S/\mathrm{VBP}}$, and the containing-mean to be calculated includes $V_{S/\mathrm{VBP}}$, $V_{S//\mathrm{PP}}$, $V_{\mathrm{PP/VP}}$, V_S, and V_{PP}. In order to calculate the required P and V values, the $N^D_{\tau 1\theta\tau 2}$ type statistical variables to be obtained include $N^D_{.//S}$, $N^D_{S/\mathrm{VBP}}$, $N^D_{S//\mathrm{PP}}$, and $N^D_{\mathrm{PP/VP}}$; the $N^D_{\tau 1}$ type statistical variables to be obtained include N^D_S and N^D_{PP}; the $N^D_{\tau 1^*}$ type statistical variables to be obtained include $N^D_{S^*}$ and $N^D_{\mathrm{PP}^*}$. Table 4 shows

Table 4: Cost estimation procedure.

	Query step	Query primitive	Estimate result nodes number and cost
1	//S	$M_{.//S}$: $GetDescendant\{., S\}$	$\mathbb{N}(M_{.//S}) = N^{D}_{.//S} = n_0$ $\mathbb{C}(M_{.//S}) = 1 \times c_{\text{ior}} + n_0 \times c_{\text{iow}} + 1 \times V_{tr} \times c_{\text{calc}}$ $\approx n_0 \times c_{\text{iow}} + N \times c_{\text{calc}}$
2	/VBP	$M_{S/\text{VBP}}$: $GetChild\{S, \text{VBP}\}$	$\mathbb{N}(M_{S/\text{VBP}}) = n_0 \times V_{S/\text{VBP}} = n_1^c$ $\mathbb{C}((M_{S/\text{VBP}})) = n_0 \times c_{\text{ior}} + n_1^c \times c_{\text{iow}} + n_0 \times V_S \times c_{\text{calc}}$
3	[ ]	$M_{S[./\text{VBP}]}$: $FilterInput1byInput2\{S, \text{VBP}\}$	$\mathbb{N}(M_{S[./\text{VBP}]}) = n_0 \times P_{S/\text{VBP}} = n_1$ $\mathbb{C}(M_{S[./\text{VBP}]}) = n_0 \times c_{\text{ior}} + n_1 \times c_{\text{iow}} + n_1^c \times V_S \times c_{\text{calc}}$
4	//PP	$M_{S//\text{PP}}$: $GetDescendant\{S, \text{PP}\}$	$\mathbb{N}(M_{S//\text{PP}}) = n_1 \times V_{S//\text{PP}} = n_2$ $\mathbb{C}(M_{S//\text{PP}}) = n_1 \times c_{\text{ior}} + n_2 \times c_{\text{iow}} + n_1 \times V_S \times c_{\text{calc}}$
5	/VP	$M_{\text{PP/VP}}$: $GetChild\{\text{PP}, \text{VP}\}$	$\mathbb{N}(M_{\text{PP/VP}}) = n_2 \times V_{\text{PP/VP}} = n_3$ $\mathbb{C}(M_{\text{PP/VP}}) = n_2 \times c_{\text{ior}} + n_3 \times c_{\text{iow}} + n_2 \times V_{\text{PP}} \times c_{\text{calc}}$

the cost estimation steps of query primitives corresponding to each query step.

4.6. Query Pipeline Construction. The construction of query pipeline is the procedure of pipeline query plan generation. It includes the following three basic steps: (1) obtaining the sequence of pipelined query primitives by parsing the XPath expression; (2) estimating the cost of each primitive in the query primitive sequence; (3) partitioning the primitive sequence according to the estimated cost, then generating each pipeline stage, and finally completing the whole pipeline construction.

Pipeline stage is the basic unit of a pipeline. In PXQ, a pipeline stage consists of one or more pipelined query primitives. Algorithm 3 describes the working process of a pipeline stage with multiple primitives.

In Algorithm 3, firstly, according to the number of query primitives, local blocking queues (lines 1 and 2) are created, and then the execution body is defined, which processes the calls of pipelined query primitives in a nested iterative manner (lines 4–15). When processing the first primitive in a stage, if the end flag of execution is detected, the flag will be passed to the following pipeline stage, and then the iteration operation in the execution body will be immediately exited (lines 6–8). When processing other primitives, it first checks whether the preceding primitive has the result output, if there is no output, it will exit the iteration processing of the primitive (line 11); otherwise, it takes an item of the result as the input unit data and then calls the pipelined query primitive (lines 9 and 13).

To get better performance, two principles are considered: one is to make full use of available worker threads and the other is to keep the load balance of pipeline stage as much as possible to avoid excessive synchronous waiting between stages. Therefore, the thread allocation strategy is as follows: when the number of query steps is less than the number of threads, each query step is allocated a thread as a pipeline stage; when the number of query steps is more than the number of threads, we need to partition the primitive sequence and merge the query steps with low cost, so that each stage can get a worker thread. The purposes of partitioning and merging are as follows: on the one hand, it can avoid the context switching overhead caused by executing multiple stages in a thread, and on the other hand, it can make the load of each stage more balanced and avoid excessive waiting between stages. When the number of query steps P is more than the number of threads T, the partition problem is converted to the evaluation of the programming problem, as shown in the following equation:

$$\begin{cases} \min\left(\sum_{i=0}^{T-1}\left(\frac{C_i^S - C_{\text{Ep}}}{T}\right)^2\right), \\ C^S = \sum_{s\in[s1,s2]} C_s, \\ C_{Ep} = \sum_{0\le p\le P-1} C_p. \end{cases} \tag{12}$$

In equation (12), T is the number of available threads, which is the same as the number of stages; C_p is the estimated cost of pipelined query primitives corresponding to each XPath query step (there are P steps); C_{Ep} is the cost of the entire query; C_s is the cost of each query primitive in the pipeline stage; and C^S is the cost of each pipeline stage. The partition result should minimize the variance of the cost of each pipeline stage.

For the structure of Figure 3(e), if it is partitioned into two pipeline stages and the result is shown in Figure 3(f), the pipeline generated is described as follows:

$\lambda 1 \leftarrow$ PipeStage({*GetChild*, *FilterInput1byInput2*, *GetDescendant*}, tName{C, ∅, D}, i{0, 0, 0}, j{1, 0, 0}, ∅, $\mathscr{E}$, $\mathscr{I}$, $\mathscr{E}^{\text{out}}$);

$\lambda 0 \leftarrow$ PipeStage({*GetDescendant*, *GetChild*, *FilterInput1byInput2*}, tName{A, B, ∅}, i{0, 0, 0}, j{0, 1, 0}, $\lambda 1.\mathscr{Q}_0$, $\mathscr{E}$, $\mathscr{I}$, $\mathscr{E}^{\text{out}}$);

If it is partitioned into four pipeline stages and the result is shown in Figure 3(g), the pipeline is described as follows:

$\lambda 3 \leftarrow$ PipeStage({*GetChild*}, tName{D}, i{0}, j{0}, ∅, $\mathscr{E}$, $\mathscr{I}$, $\mathscr{E}^{\text{out}}$);

Input: primitive name list pName, tag name list tName, input position index list i, and output position index list j in history record of unit data, blocking queue in following query primitive $\mathcal{Q}$, region encoding $\mathcal{E}$, relation index $\mathcal{I}$, and global result $\mathcal{E}^{\text{out}}$.
Output: none. The local results are processed in the algorithm: either put into the queue or output directly to the global result.

```
(1) s ← primitiveName.size;//get the number of primitives contained in stage.
(2)   Q^L ← CreateQueue(s);//create s local blocking queues.
(3) run(){//the start of the executive body
(4)     while(true)//processing of the 1st query primitive
(5)       U_0←Q_0^L.dequeue();
(6)       if(U_0 · E.id = −1∧Q ≠ ∅)
(7)         Q.enqueue(U_0);
(8)         break;
(9)       PipedPrimitive(pName[0],U_0,Q_1^L,tName[0],i[0],j[0], E, I, E^out);
(10)      while(true)//processing of the 2nd query primitive
(11)        if(Q_1^L.peek = ∅) break;
(12)        else U_1←Q_1^L.dequeue();
(13)          PipedPrimitive(pName[1],U_1,Q_2^L,tName[1],i[1],j[1], E, I, E^out);
(14)        while(true)//processing of the 3rd query primitive
(15)          ... ...//omit the remaining processing steps
(16) }//the end of the executive body
```

Algorithm 3: PipeStage(pName[], tName[], i[], j[], $\mathcal{Q}$, $\mathcal{E}$, $\mathcal{I}$, $\mathcal{E}^{\text{out}}$).

$\lambda 2 \leftarrow$ PipeStage({*GetDescendant*, *FilterInput1byInput2*}, tName{C, ∅}, i{0, 0}, j{1, 0}, $\lambda 3.\mathcal{Q}_0$, $\mathcal{E}$, $\mathcal{I}$, $\mathcal{E}^{\text{out}}$);

$\lambda 1 \leftarrow$ PipeStage({*GetChild*, *FilterInput1byInput2*}, tName{B, ∅}, i{0, 0}, j{1, 0}, $\lambda 2.\mathcal{Q}_0$, $\mathcal{E}$, $\mathcal{I}$, $\mathcal{E}^{\text{out}}$);

$\lambda 0 \leftarrow$ PipeStage({*GetDescendant*}, tName{A}, i{0}, j{0}, $\lambda 1.\mathcal{Q}_0$, $\mathcal{E}$, $\mathcal{I}$, $\mathcal{E}^{\text{out}}$);

Algorithm 4 gives the overall framework of pipeline construction.

In Algorithm 4, the query primitive sequence is extracted according to the XPath expression (line 1), and then the cost of each primitive is calculated according to the cost estimation model in Section 4.5 (line 2). Next, the pipeline is constructed according to different situations: if the number of threads is less than the number of query primitives, the query primitive sequence is partitioned by cost accumulation. Query primitives in the same partition are concatenated to create a larger pipeline stage, and then the pipeline is composed of each pipeline stage (lines 6–9); if the number of threads is more than the number of query primitives, each query primitive is created with one pipeline stage (lines 11 and 12). Finally, by modifying the parameters of the blocking queue, the connection relationship of each adjacent stage is adjusted (line 13). This algorithm only adopts a simple iterative partition according to cost accumulation and can be further optimized according to equation (12).

5. Experiments

5.1. Experimental Settings. Two different test platforms are used in this paper. One is Treebank [32] test platform. The data source of the platform is an XML data document of about 82 MB, which is a deep recursive XML dataset. The query cases from T1 to T3 in Table 5 are used to test. Among them, Case T1 is a simple path query; Case T2 is a query with multiple juxtaposed predicates and a wildcard axis operation query step; Case T3 is a query with complex nested predicates. Another test platform is XMark platform [33], which provides a tool for generating XML data documents of any size. To facilitate the comparative experiment, we generated an XML document of the same size as the Treebank dataset to test the X1 ~ X3 cases in Table 5. Among them, Case X1 is a simple path; Case X2 is a query with predicates; Case X3 query is a query with nested predicates and a query step to obtain attribute nodes.

The hardware environment of this experiment is a Dell Latitude 5290 notebook equipped with Intel Core i5-8250u CPU, 8 GB physical memory, and 240 GB SSD, which can provide 8 CPU threads. The software environment is JDK1.8 and Windows 7 (SP1) operating system.

5.2. Experimental Results. We compare the PXQ method with two typical XPath evaluation methods based on data parallelism. One is the classic navigational parallel XPath evaluation method [3, 11], which is named pNav here, and the other is the pM^2 method based on node relation matrix [4]. To facilitate comparison, the three methods adopt the result of XML parsing in the form of region encoding. The difference is only in the query process, so the performance test is limited to the query part. The execution time of PXQ includes the extra time required to obtain XML statistics. The experiments include the comparative test under different thread conditions and different data sizes.

Input: XPath query expression Ep, XML statistics $\mathcal{S}$, number of available worker threads T.
Output: Pipeline represented by pipeline stage sequence L.

(1) $M_{\text{Ep}} \leftarrow$ ExtractPrimitive(Ep);
(2) $C_{\text{Ep}} \leftarrow$ CostEstimate(M_{Ep}, $\mathcal{S}$);
(3) $P \leftarrow M_{\text{Ep}}$.size, $C_{\text{avg}} \leftarrow C_{\text{Ep}}/T$;//the number of primitives P; average cost of pipeline stage $\tau 1 \theta \tau 2$.
(4) $C^t \leftarrow 0$, $M^t_{\text{Ep}} \leftarrow n_0$;//cost accumulation C^t; primitive sequence within a stage M^t_{Ep}.
(5) **if** $(P > T)$
(6) **foreach** primitive id $i \in [0, P-1]$
(7) $C^t \leftarrow C^t + C^i_{\text{Ep}}$, $M^t_{\text{Ep}} \leftarrow M^t_{\text{Ep}} \cup \{M^i_{\text{Ep}}\}$;
(8) if $(C^t \geq C_{\text{avg}})$
(9) $\lambda_i \leftarrow$ CreatePipeStage(M^t_{Ep}), $L \leftarrow L \cup \{\lambda_i\}$;
(10) **else**
(11) **foreach** primitive id $i \in [0, P-1]$
(12) $\lambda_i \leftarrow$ CreatePipeStage(M^i_{Ep}), $L \leftarrow L \cup \{\lambda_i\}$;
(13) AdjustPipeStage(L);
(14) **return** L;

Algorithm 4: CreatePipeline (Ep, $\mathcal{S}$, T).

Table 5: Experimental cases.

Case	XPath expression	Number of results
T1	//S//NP/PP/NN	15
T2	//S[./VBP][.//NP/VP]//PP[.//IN]/*//VBN	174
T3	//EMPTY[.//VP/PP//NNP][.//S[.//PP//JJ]//VBN]//PP/NP//_NONE_	1589
X1	//open_auctions/open_auction//time	42915
X2	//regions/asia/item[./payment]//name	1440
X3	//categories[./category[./name]/@id]//description	720

5.2.1. Comparative Test under Different Thread Number Conditions. We set the experimental conditions that the number of worker threads does not exceed the number of CPU threads and then test each query case with each method to obtain the query execution time. When the number of threads is 1, each query step in pNav and pM^2 method runs in serial, while the PXQ method constructs all query steps into a single-stage pipeline and allocates a thread to execute. When the number of worker threads is greater than 1, pNav performs parallel processing according to the best manually selected query plan; however, each query step is evaluated in data parallelism according to the number of threads in the pM^2 method, and PXQ partitions the query steps to build a pipeline with the number of pipeline stages as the number of threads, and then each stage is allocated a thread to execute. The execution time comparison results under the four conditions of 1, 2, 4, and 8 worker threads are shown in Figure 6. In general, pM^2 method has better performance than pNav, while PXQ outperforms both pNav and pM^2. With the increase of the number of threads, the execution time of PXQ will decrease, showing good scalability, which is related to PXQ's ability to adaptively adjust the pipeline structure according to the thread number conditions. When the number of threads exceeds the number of query steps, PXQ creates a pipeline stage for each query step and allocates a worker thread. Therefore, if there are fewer query steps, the increase of the number of threads will not increase the running speed. For example, in the test of Cases T1 and X1, due to the same pipeline structure under 4 and 8 threads, the execution time is similar. However, in some cases, the increase of the number of threads leads to the increase of the execution time of the pM^2 method. For example, the execution time of Cases T1 and T2 under the condition of 8 threads is longer than that under the condition of 4 threads. This is because when the number of threads increases, the overhead of thread cooperation for data parallelism exceeds the parallelization benefits of increasing threads. In the case of many query steps, such as T2, T3, and X3, the PXQ method can still maintain good performance by partitioning and merging the query steps to construct the appropriate pipeline stage.

5.2.2. Comparative Test under Different Data Size Conditions. We use the XMark tool to generate XML documents with data size of 125 MB, 250 MB, 500 MB, and 1 GB, respectively, and then set the number of available threads as the number of CPU threads, that is, 8 threads,

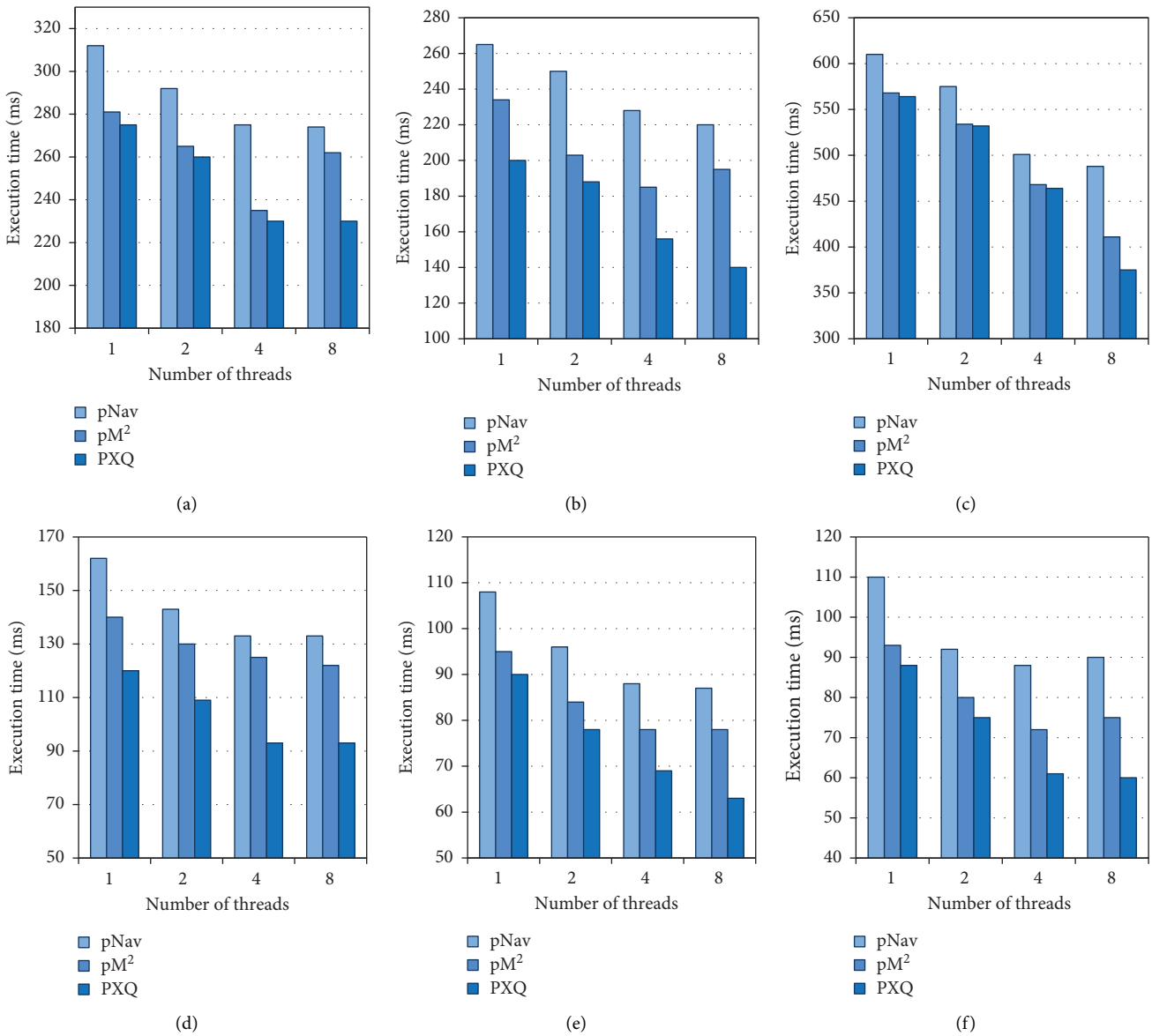

Figure 6: Comparison of execution time under different thread number conditions. (a) Case T1. (b) Case T2. (c) Case T3. (d) Case X1. (e) Case X2. (f) Case X3.

for comparative experiments. To avoid garbage collection and disk storage exchange caused by insufficient memory in the case of large size of XML data, the initial and maximum heap space of JVM are set to 6 GB. The experimental results are shown in Figure 7. As can be seen from the figure, PXQ runs faster than both pNav and pM^2 on each test case, which shows that PXQ has better scalability for data size. The formula $\rho = |D|/t$ is used to calculate the nominal query throughput of each test project, where $|D|$ represents the size of the input document and t is the execution time. It is found that, with the increase of data size, the throughputs of the three methods also increase in most cases, while the average throughput of PXQ is about 45% and 26% higher than that of pNav and pM^2, respectively, indicating that PXQ has more advantages in the scalability of data processing.

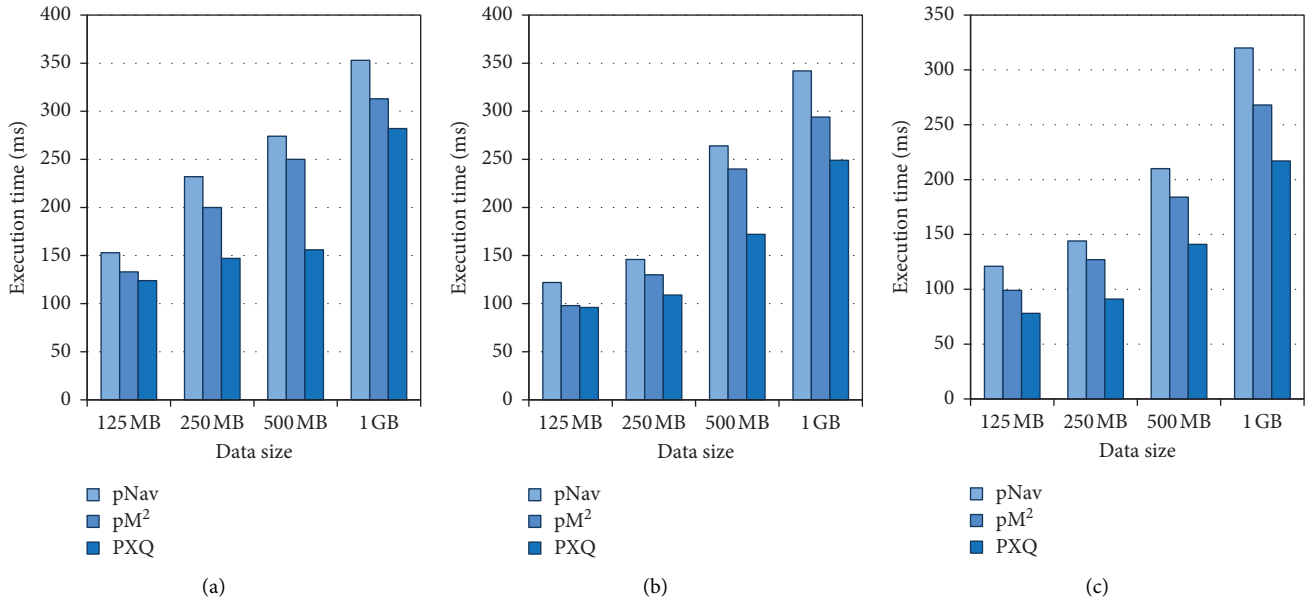

Figure 7: Comparison of execution time under different data size conditions. (a) Case X1. (b) Case X2. (c) Case X3.

6. Conclusion

With the popularity of multicore computing environment, XPath query technology for multicore parallel computing provides an important way to improve the performance of XML query. In order to make full use of multicore resources to improve query performance and adapt to the data stream processing scenarios of XPath, this paper proposes a pipeline XPath query method based on cost optimization. Our method uses pipelined query primitives as the basic query processing unit and query primitives based on relation index which can perform query step evaluation efficiently. Moreover, the primitives are easy to be concatenated to support the creation of large pipeline stages. In the aspect of load balancing in the pipeline stage, a cost model based on XML statistics is proposed to guide the generation of the pipeline stage. To fully utilize available worker threads and optimize query performance, the strategy of determining the number of pipeline stages according to the number of available threads is adopted. Compared with the existing typical XPath evaluation methods based on data parallelism, the experimental results under different thread number conditions and different data size conditions show that our method can obtain better performance. Our future work is to optimize the design of pipelined query primitives to further improve the efficiency of pipelined queries, and provide query primitives with more types to support full XPath query semantics.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was partially supported by the Natural Science Foundation of Fujian Province of China (no. 2018J01538) and Open Fund of Digital Fujian Big Data Modeling and Intelligent Computing Institute.

References

[1] P. Buneman, "Semistructured data," in *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 117–121, ACM, Tucson, AZ, USA, May 1997.

[2] J. Robie, M. Dyck, and J. Spiegel, "XML path language (XPath)," 2017, https://www.w3.org/TR/xpath/.

[3] R. Bordawekar, L. Lim, and O. Shmueli, "Parallelization of XPath queries using multi-core processors," in *International Conference on Extending Database Technology: Advances in Database Technology (EDBT2009)*, pp. 180–191, Saint-Petersburg, Russia, March 2009.

[4] R. Chen, H. Liao, and Z. Wang, "Parallel XPath evaluation based on node relation matrix," *Journal of Computational Information Systems*, vol. 9, no. 19, pp. 7583–7592, 2013.

[5] S. Sato, W. Hao, and K. Matsuzaki, "Parallelization of XPath queries using modern XQuery processors," *New Trends in Databases and Information Systems. ADBIS*, vol. 2018, 2018.

[6] X. Huang, X. Si, X. Yuan, and C. Wang, "A dynamic load-balancing scheme for XPath queries parallelization in shared memory multi-core systems," *Journal of Computers*, vol. 9, no. 6, 2014.

[7] B. Karsin, H. Casanova, and L. Lim, "Low-latency XPath query evaluation on multi-core processors," in *Hawaii International Conference on System Sciences*, pp. 6222–6231, Hilton Waikoloa, HI, USA, January 2017.

[8] X. Wu and D. Theodoratos, "A survey on XML streaming evaluation techniques," *The VLDB Journal*, vol. 22, no. 2, pp. 177–202, 2013.

[9] G. Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys (CSUR)*, vol. 25, no. 2, pp. 73–170, 1993.

[10] A. G. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, Raleigh, NC, USA, September 2009.

[11] R. Bordawekar, L. Lim, and A. Kementsietsidis, "Statistics-based parallelization of XPath queries in shared memory," in *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, ACM, Lausanne, Switzerland, March 2010.

[12] C. Grün, L. Wörteler, L. Kircher, and R. Shadura, "BaseX: the XML framework," 2018, https://basex.org/.

[13] C. Barton, P. Charles, D. G. D. Goyal, M. Fontoura, and V. Josifovski, "Streaming XPath processing with forward and backward axes," in *Proceedings of the International Conference on Data Engineering*, Bangalore, India, March 2003.

[14] J. Kwon, P. Rao, B. Moon, and S. Lee, "Value-based predicate filtering of XML documents," *Data & Knowledge Engineering*, vol. 67, no. 1, pp. 51–73, 2008.

[15] Y. D. Peter, P. Fischer, M. J. Franklin, and R. To, "YFilter: efficient and scalable filtering of XMLDocuments," in *Proceedings ICDE*, pp. 341-342, San Jose, CA, USA, March 2002.

[16] S. H. Kim, K. H. Lee, and Y. J. Lee, "Multi-query processing of XML data streams on multicore," *Journal of Supercomputing*, vol. 73, no. 6, pp. 1–30, 2016.

[17] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi, "Query processing of streamed XML data," in *Proceedings of the CIKM*, ACM, McLean,Virginia,USA, November 2002.

[18] S. Kim, Y. Lee, and J. J. Lee, "Matrix-based XML stream processing using a GPU," in *Proceedings of the IEEE International Congress on Big Data*, New York, NY, USA, June 2015.

[19] D. Zinn, S. Bowers, T. M. Mcphillips, and B. Ludäscher, "X-CSR: Dataflow optimization for distributed XML process pipelines," in *Proceedings of the 25th International Conference on Data Engineering*, ICD, Shanghai, China, March 2009.

[20] N. Walsh, A. Milowski, and H. Thompson, *XProc: An XML Pipeline Language*, W3C, Cambrige, MA, USA, 2010.

[21] R. Lopes and L. Carrico, *Automating XML Pipelines through Rules*XATA, Burnsville, MN, USA, 2007.

[22] M. Spiliopoulou, M. Hatzopoulos, and Y. Cotronis, "Parallel optimization of large join queries with set operators and aggregates in a parallel environment supporting pipeline," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 3, pp. 429–445, 1996.

[23] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton, *Elastic Pipelining in an In-Memory Database Cluster*ACM Sigmod, New York, NY, USA, 2016.

[24] Y. Li, X. Shi, and H. Jin, "Runtime-aware adaptive scheduling in stream processing," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 14, 2015.

[25] W. Jiang, H. M. Sha, Q. Zhuge, L. Yang, H. Dong, and X. Chen, "On the design of minimal-cost pipeline systems satisfying hard/soft real-time constraints," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 1, 2018.

[26] A. Aboulnaga, A. Alameldeen, and J. Naughton, "Estimating the selectivity of XML path expressions for internet scale applications," in *Proceedings of VLDB Conference*, Rome, Italy, March 2001.

[27] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On supporting containment queries in relational database management systems," *ACM SIGMOD Record*, vol. 30, no. 2, pp. 425–436, 2001.

[28] K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang, *Storing and Querying Ordered XML using a Relational Database System*ACM Sigmod, New York, NY, USA, 2002.

[29] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava, "Navigation- vs. index-based XML multi-query processing," in *Proceedings 19th International Conference on Data Engineering*, Bangalore, India, March 2004.

[30] P. Rao and B. Moon, "PRIX: indexing and querying XML using prufer sequences," in *Proceedings of the 20th International Conference on Data Engineering*, pp. 288–299, Boston, MA, USA, April 2004.

[31] D. Megginson, "Simple API for XML (SAX 2.0)," 2004, http://sax.sourceforge.net/.

[32] University of Pennsylvania, "University of Pennsylvania treebank project," 2002, http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/data/treebank/treebank_e.xml.

[33] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: a benchmark for XML data management," in *VLDB '02: Proceedings of the 28th International Conference on Very Large Databases*, pp. 974–985, VLDB Endowment, Hong Kong, China, August 2002.