NAIST-IS-DD0661209

Doctoral Dissertation

Making XML Database Systems Scalable to Computer Resources and Data Volumes

Makoto YUI

February 5, 2009

Department of Bioinformatics and Genomics
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Makoto YUI

Thesis Committee:

Professor Hirokazu Kato	(Supervisor)
Professor Hiroyuki Seki	(Co-supervisor)
Associate Professor Jun Miyazaki	(Co-supervisor)
Assistant Professor Toshiyuki Amagasa	(Committee from Tsukuba University)

Making XML Database Systems Scalable to Computer Resources and Data Volumes*

Makoto YUI

Abstract

Increasing use of XML has emphasized the need for scalable database systems that are capable of handling a large amount of XML data efficiently. This study explores effective methods for making a scalable XML database system in the following aspects: (a) scalability to data volumes, (b) scalable XML processing with a shared-nothing PC cluster, and (c) scalable database processing on shared-memory multiprocessors. In the study of (a), we propose an XQuery processing scheme in which an XML document is internally represented as a set of blocks and can directly be stored on a secondary storage. Our experimental results showed that our storage scheme is scalable to data volumes and outperforms competing schemes with respect to I/O intensive workloads. In (b), we discuss on-the-fly XML processing using shared-nothing PC clusters. We propose a scheme for distributed and parallel query processing that employs a pass-by-reference semantics by using remote proxy. Previously proposed methods that use pass-by-value semantics have often suffered from redundant communication between processor elements and limited inter-operator parallelism. To cope with these problems, we developed a distributed XML query processing scheme that leverages the benefit of lazy evaluation. Our experimental results showed that our proposed scheme obtains up to 22x speedups compared with competitive methods, and demonstrated the importance of distributed XML database systems to take pass-by-reference semantics into consideration. In (c), we explain the internal locking in the buffer management module that prevents databases from being scalable to the number of processors. We further propose a scalable buffer management scheme that employs non-blocking synchronization instead of locking-based ones. Our experimental results revealed that our

*Doctoral Dissertation, Department of Bioinformatics and Genomics, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0661209, February 5, 2009.

scheme can obtain nearly linear scalability to processors up to 64 processors, although the existing locking-based schemes do not scale beyond 16 processors. Finally, we conclude our studies with examining our XML native database system built on top of the three contributions.

Keywords:

XML Database, Distributed Query Processing, Non-blocking Synchronization, Buffer Management

Acknowledgements

My deepest gratitude goes to Professor Hirokazu Kato and Professor Shunsuke Uemura for their supervision. Professor Hirokazu Kato gave me lots of advices derived from his different perspective without the slightest hesitation. The attitude was worth imitable for me. I do not know how to express my deep gratitude to my former supervisor Shunsuke Uemura giving me a chance to start my Ph.D. program at NAIST. He encouraged me to complete my master course ahead of the schedule. Without his advices and encouragement, I could not finish my research this time. I have to learn flexible and forthcoming policies that both of my supervisors take.

I appreciate Associate Professor Jun Miyazaki for his appropriate guidance. When I started my master course, I worried that no faculty member is left in XML research group of the Database Laboratory (It was beyond my assumption!). But my worries were soon over due to his vast range of knowledge in database systems. I always enjoyed concentrated discussion with him.

I am also grateful to the members of my dissertation committee for giving comments to improve this dissertation. Professor Hiroyuki Seki suggested valuable and sharp comments throughout the master and doctoral course. I would like to thank Assistant Professor Toshiyuki Amagasa for reviewing my dissertation. He kindly gave me an approval to use his prototype system *XRel* when I was an undergraduate student at SHIBAURA Institute of Technology. It was a catalyst that decided me to start studying at NAIST.

I also thank Associate Professor Atsuyuki Morishima for his supervision of my

B.S. degree at SHIBAURA Institute of Technology. He awoke my interest in research and XML. What learned in the Data Engineering Laboratory became the basis of my current research. I have fond memories there; truly stimulating relationship with colleagues including Hayato Ito, Akiyoshi Nakamizo and Akira Kojima.

Last but not least, thanks to all current members of Interactive Media Design Laboratory and former members of Database Laboratory. They provided supportive environment all the time and tolerated me patiently. I will never forget the well-spent years at NAIST.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

After XML 1.0 [BPSM+03] became a W3C recommendation on February 10, 1998, XML data has been increasing and spreading over computer networks. Since then, database community has been actively working on diverse research issues on XML data management for over 10 years.

This dissertation describes effective methods for making a scalable XML database system. Making a scalable XML database system is worth trying theme, not simply because scalability of database is a primary concern for database systems, but because even state-of-the-art XML database systems are not scalable enough for web-scale computing where large-scale distributed computing is mandatory.

1.2 Approach

The goal of this dissertation is, in short, making XML database systems scalable. We discuss scalability issues in XML databases in the following aspects:

(a) scalability to data volumes,

(b) scalable XML processing with a shared-nothing PC cluster, and

(c) scalable database processing on shared-memory multiprocessors.

While distributed XML processing (b) is apparently important, local XML processing (a and c) is important as well since a distributed XML processor is built on top of local XML processors.

In the study of (a), we propose an XQuery [W3Cd] processing scheme in which an XML document is internally represented as a set of blocks and can directly be stored on secondary storage. The results of our experiments clearly show that the proposed scheme can often obtain almost linear scalability in performance as the data size increases.

In studying (b), we discuss on-the-fly XML processing using shared-nothing PC clusters. We believe that on-the-fly processing of XML increases developers' attention because dynamic XML documents on the web — web feeds in RSS [RSS00] or ATOM [NS] format, search results in XML published by information systems, and scientific data that are updated on an ongoing basis from individual laboratories around the world — are increasing. However current XML query processor technologies provide no such facilities that make on-the-fly processing for thousands of XML data realistic. Responding to the latent demand, we introduce a divide-and-conquer approach that divides a query into multiple-queries to XML query processing. Our enhanced XML query processor parallelizes the execution of divided queries on multiple computation nodes. We further address outstanding issues that such a hierarchical distributed system must deal with.

In studying (c), we address processor scalability issues for database systems in a setting when processor manufacturers are increasing the number of CPU cores per chip. In particular, we explain how internal locking in buffer management module prevents databases from being scalable to the number of processors. The issued problems had become a reality in the development of our XML database system. To deal with them, we propose a scalable buffer management scheme that employs non-blocking synchronization instead of locking-based ones. Our experimental results revealed that our scheme can obtain nearly linear scalability to processors up to 64 processors, although the existing locking-based schemes do not scale beyond 16 processors.

Finally, we conclude our studies with examining our XML native database system built on top of the three contributions and show accumulated knowledge about making

an XML database system scalable.

1.3 Thesis Organization

The remainder of this dissertation is organized as follows. Before going to the main topics, Chapter **2** describes XML-related technologies. Chapter **3** gives a survey on XML database systems. As well as introducing influential researches to my dissertation, research opportunities of equal importance in the discussion of current XML database systems are explored. In Chapter **4**, we propose an XQuery processing scheme in which an XML document is internally represented as a set of blocks and can directly be stored on secondary storage. In Chapter **5**, we discuss on-the-fly XML processing using shared-nothing PC clusters. In particular, we focus on an aspect of distributed XQuery processing that involves data exchanges between processor elements. In Chapter **6**, we explain the internal locking in the buffer management module that prevents databases from being scalable to the number of processors. And then, we propose a scalable buffer management scheme that employs non-blocking synchronization instead of locking-based ones. Chapter **7** concludes this dissertation with examining our XML native database system built on top of the three contributions. We further give a future direction of my research.

2 XML and Related Technologies

This chapter introduces XML and related technologies that are required in discussions of this dissertation. We assume the reader moderately familiar with XML; they at least know the basic syntax of XML and the tree-structured nature of XML data.

2.1 XML Data Model

XML itself is just a document format, in which some structural irregularities are permitted, and not a data model. XML documents must satisfy syntactic constraints defined in the specification [BPSM+03]. XML has two correctness levels:

- *Well-formed*. An XML document is well-formed if it obeys the syntax of XML.
- *Valid*. A well-formed XML document is called valid if it is conformed (validated) with semantic rules described in a particular schema [W3Cc, CM01] or DTD [BPSM+03].

We treat well-formed XML documents in the rest of this dissertation.

2.1.1 Modeling XML-encoded Information

XML Information Set (XML Infoset) [CT04] describes an abstract data model of a well-formed XML document. XML Infoset forms the basis for other specifications that need to refer to the information in a well-formed XML document and does not restrict XML data model to tree.

On the other hand, XML is often modeled as a tree [HHN+00, W3Ce]. Both of *XPath 1.0 data model* [W3Ca] and *XQuery 1.0 and XPath 2.0 Data Model* (XDM) [W3Ce] model an XML document as a tree of nodes. The tree, often referred to as *XML tree* in the literature, contains nodes. There are seven types of node: document, element, attribute, text, namespace, processing instruction, and comment.

Both of XPath 1.0 and XPath 2.0/XQuery 1.0 expressions operate over XML trees. However, significant differences appear in the underlying data models between XPath 1.0 [W3Ca] an XPath 2.0 [W3Cb] as follows:

- Every value in XPath 2.0 is a sequence of typed items instead of node sets.

  Sequences are ordered and may have duplicates in contrast to node sets are unordered and have no duplicates. The items in XPath 2.0 may be nodes or atomic values aside from XPath 1.0 consists of simple four types — node sets, number, string, and boolean. Atomic values may belong to any of the 19 primitive types defined in the XML Schema [BM04], e.g., string, boolean, double, float, decimal, dateTime, QName, and so on.

- XPath 2.0 data model allows trees to be rooted at other kinds of node sequences instead of node sets in XPath 1.0.

- Nodes may be typed or untyped in XPath 2.0. A node can acquire a type as a result of validation against an XML Schema.

2.2 XML Query Language

As increasing amounts of information are stored, exchanged, and presented using XML, the ability to intelligently query XML data sources becomes important.

Creating a new language is a serious business. Lots of time/persons have been spent in defining a standard query language for XML. XQuery 1.0 [W3Cd] eventually became a W3C Recommendation on January 23, 2007. XQuery is derived from an XML query language called Quilt [CRF01], which in turn borrowed features from several other languages, including XPath 1.0 [W3Ca], XQL [RLS98], XML-QL [DFF+98], SQL, and OQL [CA95].

2.3 XQuery Fundamentals

2.3.1 FLWOR Expression

XQuery expression is often characterized by FLWOR expression. The following shows each role of the clauses.

- *for* creates a sequence of tuples.
- *let* binds a sequence to a variable.
- *where* filters the tuples on a boolean expression.
- *order by* sorts the tuples.
- *return* gets evaluated once for every tuple.

An XPath expression in Table 2.1 can be expressed in XQuery as in Table 2.2.

The expression below will select all the title elements under the book elements that are under the bookstore element that have a price element with a value that is higher than 30.

```
doc("books.xml")/bookstore/book[price>30]/title
```

Table 2.1. An example of XPath expression.

```
for $store in doc("books.xml")/bookstore
let $x := $store/book
where $x/price>30
return $x/title
```

Table 2.2. An example of FLWOR (FLWR) expression.

2.3.2 Functional Aspects

XQuery is a functional programming language that does not contain side effects. The basic evaluation order is defined in XQuery formal semantics [W3Cf].

Aside from that, some *evaluation order* of expression, e.g., function calls, are not defined and implementation-dependant. The table below shows an example of the function call semantics.

> The order of function argument evaluation is implementation-dependent and a function need not evaluate an argument if the function can evaluate its body without evaluating that argument.
>
> http://www.w3.org/TR/xquery/#id-function-calls

Lazy Evaluation

Our XQuery implementation, namely XBird, takes a call-by-name evaluation strategy and most expressions are lazily evaluated. Next, we briefly illustrate the semantics using examples.

When evaluating Table 2.3, XBird returns ”lazy” as the result. On the other hand, Saxon [Sax], a famous XQuery implementation, causes stack-overflow because Saxon eagerly evaluates function parameters. Table 2.4 can also be evaluated normally in XBird. The query is also executable with Saxon.

Recursive Call

Recursion is a powerful feature in function definitions. Recursive functions are useful that are defined over a hierarchical data model such as XML.

As an illustration of a recursive function, the depth function in Table 2.5 can be invoked on an element and returns the depth of the element hierarchy beginning with its argument. Table 2.6 is the example that returns the maximum depth of elements in the first *author* element.

```
declare function local:params() {
    local:params()
};
declare function local:test($x) {
    "lazy"
};
declare function local:f() {
    let $x := local:params()
    return local:test($x)
};
local:f()
```

Table 2.3. An example query in which function parameters are never evaluated.

```
declare function local:endlessOnes() as xs:integer*
{
  (1, local:endlessOnes())
};
some $x in local:endlessOnes() satisfies $x eq 1
```

Table 2.4. A lazy list in XQuery.

```
declare function local:depth($root as element()) as xs:integer
{
  if(empty($root/*)) then 1
  else 1 + max
    (for $c in $root/* return local:depth($c))
};
local:depth($input)
```

Table 2.5. A recursive query that returns the maximum level of elements.

```
let $input :=
 <authors>
   <author>
      <fName>William</fName>
      <lName>Shakespeare</lName>
   </author>
   <author>
      <fName>Ernest</fName>
      <lName>Hemingway</lName>
   </author>
 </authors>
return local:depth($input/author[1])
```

Table 2.6. A query that returns the maximum depth using recursion.

2.3.3 Research Opportunities

While XQuery 1.0 already became a W3C Recommendation, several opportunities for XQuery 1.1 [Eng07] and fundamental challenges in XML query language exist:

- *Updating*: The XQuery Update Facility [W3Cg] appeared as a draft specification to allow XQuery expressions to perform database update. Implementations of it are starting to appear. MonetDB/XQuery [BGvK+06] proposed an efficient updating scheme using a *shadow paging* technique.

- *Fulltext search*: W3C has been developing a full-text search facility for use with XPath 2.0 and XQuery [AYBB+08]. Supporting full-text (IR-style) queries over XML is becoming most hot research topic [YS05].

- *Stream processing*: Processing XML path queries over steaming XML data has been actively studied, e.g., in [IHW02, PC03]. Recently it is adapted to XQuery [BKF+07] and XQuery 1.1 [Eng07] suggested to support *windowing*.

- *Functional features*: Supporting higher-order functions and lambda expressions is suggested in [Eng07].

- *Procedural features*: The impedance mismatch problem has existed in the relational database world for many years. One historically successful approach to this problem was to extend SQL with control logic, resulting in stored procedure languages such as PL/SQL in Oracle and Transact-SQL in SQLServer. Supporting procedural features in XQuery is challenging because procedural languages, in general, reduce optimization opportunities that are available in declarative/functional languages. XQueryP [CCF+06] is the first attempt for that.

- *Distributed query processing features*: Supporting distributed XML query is essentially important because XML data has been spread over computer networks [ABC+03, Suc02]. Several researches addressed this problem [ABC+03] and proposed extending XQuery for distributed XML processing [RBHS04, ZB07b, FJM+07a]. We also address this problem in Chapter **5**.

3

Survey on XML Database Systems

This chapter makes a survey on XML database systems with introducing influential researches to this dissertation. Research opportunities of equal importance in the discussion of current XML database technologies are also explored.

The free-form nature of XML has provided various new opportunities for database community [Wid99] as partially listed below:

- modeling XML-encoded information as a true data model [HHN+00, GMW99],
- an appropriate query language for querying XML data [CRF01, W3Cd],
- efficient physical layouts [KM00, ZKO04] and indexing mechanisms for XML data [JLWO03],
- storing and querying XML data on traditional DBMSs [FK99, TVB+02, YASU01],
- join processing tailored for XML tree structures [AKJP+02, JLWO03, BKS02],
- publishing XML documents from traditional data sources (e.g., relational tables) [FKS+02, CKS+00],
- information retrieval (IR)-style searches on XML documents [YS05], and
- benchmarking XML data processing [SWK+01, YÖK04].

In the following sections, we introduce some key elements for our research. In Section **3.1**, we introduce semistructured data model referring to one of XML. Section **3.2** gives an overview of XML database systems. In Section **3.2.1**, XML native database systems are independently introduced. Finally, in Section **3.3**, we introduce previous research on distributed XML processing. We furture refer to parallel database systems in Section **3.3.1**.

3.1 Semistructured Data and XML

XML is often said to be semistructured because structural irregularity of XML can be caused by its self-describing and schema-free nature. The XML data model is similar to that for semistructured data in the sense that both represent data as a directed graph. An important difference is that XML data model has order while semistructured data is usually unordered.

Research work on XML shared earlier challenges on semistructured data management because XML documents are essentially semi-structured [Bun97, Abi97]. Semistructured databases, unlike traditional rigid databases, do not have a fixed schema known in advance. It posed new challenges in large areas such as index structures [GW97, MS99] and data model [GMW99].

Let us consider query evaluation for instance. Navigation over a semistructured graph is fundamental part of query evaluation. Due to the lack of information about the schema, a naive evaluation involves unnecessary scans of the database in search of those paths that satisfy a given query. In addition, since navigating the graph is essentially a pointer traversal and the graph objects may be scattered across the disk, some queries may require numbers of disk accesses and cause significant performance degradation.

Index structures for semistructured data have been developed in order to address these problems. *DataGuide* [GW97] generates a path index that summarizes all paths in the database that start from the root. In a sense, the path index serves as dynamic schema, generated from semistructured databases. This schema information as well as the index facilities are useful for query optimization and reduce the portion of database to be scanned in query evaluation.

Structural summaries are also effective for XML path processing with associating node paths in XML trees with the node-set that paths reach. Several variations of structural summaries have been proposed and a detailed survey can be found in [CMV05]. Fundamental research opportunities remained to develop an efficient storage scheme for path traversals [MBB+06]. In Chapter **4**, we introduce an efficient storage scheme optimized for frequent access patterns in XML query processing.

3.1.1 DataGuide and Strong DataGuide

As shown in Figure 3.1, DataGuides guarantee that each label path in the data graph reaches one node in the DataGuide but do not prevent multiple label paths from reaching the same DataGuide node.

One type of DataGuide which guarantees that all label paths that reach the same node in the DataGuide as in Figure 3.2 is called *Strong DataGuide* [GW97]. Strong DataGuide summarizes all path information in G into G_I. G_I can be viewed as a DFA converted from G, which, in turn, can be viewed as a NFA. Strong DataGuides might be exponential in the size of G. However, [MS99] showed that when G is a tree, Strong DataGuide is reduced to *1-index* [MS99] whose size does not exceed the size of G.

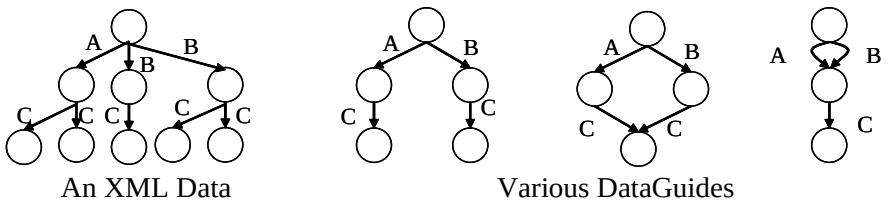


Figure 3.1. Various DataGuides of an XML tree.

3.1.2 Data Model for Semistructured Data

Semistructured data is often modeled as some form of labeled, directed graph [Abi97, Bun97]. There are two major approaches in modeling semistructured data: *node-labeled graph* and *edge-labeled graph*. Node-labeled graphs have lables on nodes and edge-labeled graphs have labels on edges as shown in Figure 3.3. As long as treating trees,

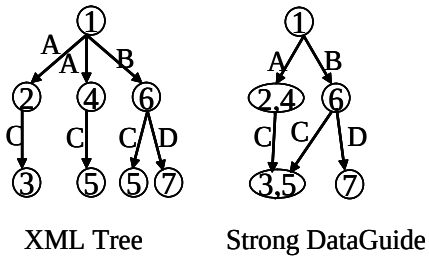
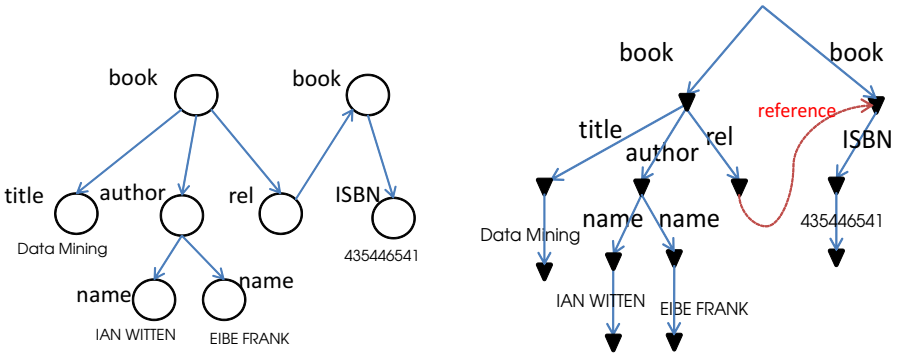


Figure 3.2. A Strong DataGuide of an XML tree.

the distinction between *node-labeled* and *edge-labeled* is minor [ABS99]. When treating graphs, the distinction between the two models becomes important. An intuitive difference is that edge-labeled graphs have unique identifiers for each edge while node-labeled graphs have one for each node. This property of edge-labeled graphs becomes advantages when a node is referenced by multiple edges; every label is identifiable with the edge.

XML defines two particular attributes associated to unique identifiers for linking elements, called *ID* and *IDREF*. They used to link elements beyond the relationship given by the tree structure of XML documents. This mechanism allows us to define an XML document that has a graph structure in addition to a tree structure.



(a) node-labeled graph (b) edge-labeled graph

Figure 3.3. An example of node and edge labeled graphs.

3.1.3 Lore and Object Exchange Model

The Lore project [MAG+97, GMW99], developed at Stanford Database Group, had produced numerous and influence ideas, as partly listed below, to XML database research derived from their earlier efforts on semistructured database.

- *DataGuide* [GW97] for the structural summaries/indices for semistructured data.
- *Query formulation* techniques (and its user interface) that can assist users to compose a query without structural information of database in advance.
- *Object Exchange Model (OEM)* [PAM96] is self-describing, nested object model that can intuitively be thought of as a labeled, directed graph. The data in OEM can be exported and imported among other mediators.
- *Lorel*, Lore's query language for querying semistructured data [AQM+97], has familiar *select-from-where* syntax. Lorel influenced the design of XQuery.

3.2 XML Database System

XML database systems fall into three main categories as seen in Figure 3.4:

- *XML-enabled database* uses an existing DBMS to provide interfaces for XML handling and decomposes XML data into the internal data model, e.g., relational tables or object store.
- *Native XML database (NXDB)* that uses a special mechanism for storing and querying XML data [JAKC+02, KM00].
- *Hybrid database* is a variant of XML-enabled databases while it also provides similar facilities to NXDBs, i.e., XML-specific data store and XML-specific evaluation techniques. It essentially can be considered either XML-enable databases or NXDBs [BCJ+05, BGvK+06].

In XML-enabled databases, two approaches can be considered in designing the underlying database schemes [YASU01]:

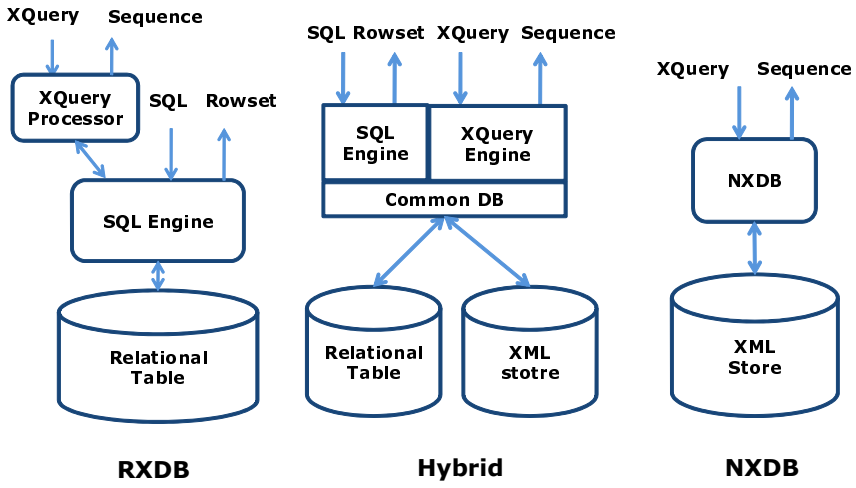


Figure 3.4. Implementation strategies of XML databases.

- *structure-mapping approach*: Database schemas represent the logical structure (or DTDs if they are available) of target XML documents. In the structure-mapping approach, a database schema is defined for each XML document structure or DTD.

- *model-mapping*: Database schemas represent constructs of the XML document model. In this approach, a fixed database schema is used to store the structure of all XML documents.

Because of the wide availability, robustness, manageability of RDBMSs, and ease-of-development of research prototypes, the shredding solutions have received a lot of attention. On the other hand, a hybrid solution [BCJ+05] showed industrial-strength and draw unprecedented attention recently.

Our system can be classified to native XML databases because we (at least the author) believe that future XML-enabled DBMSs take hybrid approaches as in [BCJ+05]. Our techniques, proposed in Chapter **4**, can also be applied to hybrid databases.

3.2.1 XML Native Database System

As defined by the XML:DB initiative [The01], an XML native database (NXDB) uses an XML document as its fundamental unit of logical storage. It stores and retrieves documents according to that model.

Data Organization

Data organization schemes characterize XML native databases. Physical composition unit of an XML tree can be classified to three models as seen in Figure 3.5: *document-level*, *node-level*, *block-level*. A block-level composition relies on the page (or a disk block) allocation scheme of XML nodes.

Natix [KM00] uses a subtree for the basic storing and accessing unit. NoK [ZKO04] uses a succinct string representation for an XML tree, namely *subject tree*. The string representation is allocated to disk pages in a depth-first search manner. Both of Timber [JAKC+02] and eXist [Mei06] take a node object as the storing unit. Xindice [Apaa] uses a document-level physical composition. OrientStore [MLLA03] uses explicit schema information for designing a physical allocation.

On the other hand, we propose an alternative storage scheme, namely pDTM, in Chapter **4**.

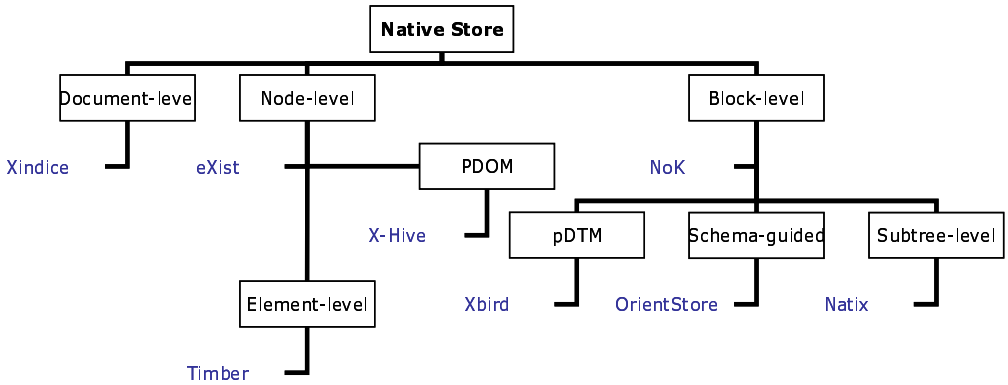


Figure 3.5. XML-native data storing schemes.

3.3 Distributed XML Processing

Distributed databases share workloads among network-connected computation nodes to reduce the amount of computation for each database server. The design of distributed XML database systems involves making decisions on the placement of XML data and computation as depicted in Figure 3.6. In particular, data partitioning is important in a sense that determines a good part of later computation.

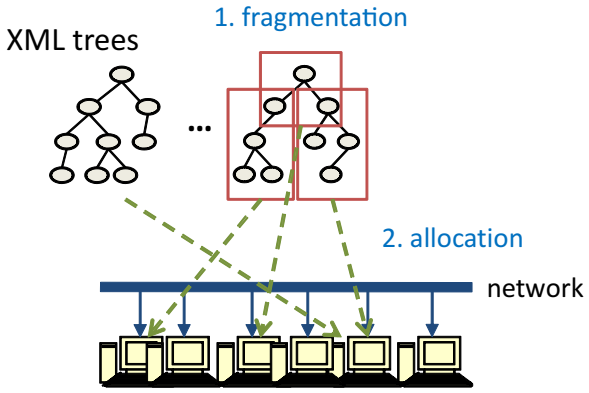


Figure 3.6. Data partitioning of XML data.

When building distributed XML processors, the following techniques should be considered:

- *Data distribution* (*partitioning*):

    1. *Fragmentation*: With respect to fragmentation, the important issue is the appropriate unit of distribution. XML data should be fragmentized considering data size as well as tree structure of each fragment. Since XML has tree structure, XML data have to be fragmentized preserving the integrity of the tree structure. On the other hand, no fragmentation scheme is a considerable option that uses document-level distribution instead of subtree-level fragmentation. We, in Chapter **5**, follow the former scheme mainly because the disadvantages of fragmentation [OV99] (e.g., difficulty in supporting full-fledged XQuery [W3Cd]).

2. *Allocation*: The allocation of resources across the nodes of a computer network is a classical problem though few are studied for distributed XML data management. Allocation problem is formalized [OV99] as follows.
   Assume that there are a set of fragments $F = F_1, F_2, \ldots, F_n$ and a network consisting of sites $S = S_1, S_2, \ldots, S_m$ on which a set of applications $Q = q_1, q_2, \ldots, q_q$ is running. The allocation problem involves finding the "optimal" [DF82] distribution of F to S.

- *Distributed query processing*:

  1. *Query optimization*: Parallel query optimization should take the maximum parallelisms. We explain the parallelism in Section **3.3.2**.
     Using index structures for distributed query processing is worth consideration. Bremer et al. [BG03] used a summary index structure, which is similar to *Strong DataGuide*, and took *index shipping* into consideration.
  2. *Evaluation strategies*: Distributed XML processing in P2P environment has been actively studied as described in [KP05]. Most of them provide limited expressive power (i.e., simple path matching using *distributed hash tables* (DHTs)) for XML query processing.
     MonetDB/XRPC [ZB07a] proposed *bulkRPC* in which *set-at-a-time* processing is exploited to reduce the number of RPCs.

- *Load balancing*: Kurita et al. [KHMU07] proposed adaptive data placement scheme that adaptively relocates partitioned XML fragments based on CPU load.

We propose, in Chapter **5**, an efficient distributed XML query evaluation strategy using *remote proxy*. The proposed scheme introduces a *lazy evaluation* technique and a load balancing mechanism to distributed XML query processing.

3.3.1 Parallel Database Systems

Parallel database systems (database systems on parallel computers [DG92]) had been intensively studied between the late 80's and 90's when multiprocessor computers displaced traditional mainframe computers. For example, in the 90's, Wal-Mart famously utilized parallel databases to gain radical efficiencies in supply chain management via

item-level inventory and historical sales information. This change evokes the current situation that increasing number of processors per chip has raised critical challenges in software engineering [Sut05].

Recently, excitement about *MapReduce* [DG04] has spread quickly in the computing industry. Some database leaders have argued publicly that the MapReduce phenomenon is not a technical revolution at all [DS08]. Actually, what MapReduce made is, at least partly, a reinvention of parallel database techniques.

These facts imply that parallel database techniques are now worth getting reconsidered. In Chapter **5**, we introduce *pipeline parallelism* to XML query processing. Distributed query processing of XML is different from one of parallel databases in terms of treating contents. Traditional parallel databases, in general, consider neither semi-structured nor unstructured data.

The following sections briefly explore database parallelism originating from past studies of parallel databases.

Architectures of Parallel Database Systems

The followings are commonly mentioned multiprocessor architectures for parallel systems:

- *Shared memory* (*shared everything*): multiple processors share a common central memory.
- *Shared disk*: multiple processors each with private memory share a common collection of disks.
- *Shared nothing*: each node is independent and self-sufficient. Communication among processors is achieved by message passing.

Currently succeeding architecture includes shared nothing (MySQL NDB cluster and MapReduce [DG04]) and shared disk (Oracle RAC). An influential discussion can be found in [Sto86] that describes pros and cons of each architecture. Their conclusion is that shared nothing is adequate to address the most case.

We are considering to apply shared nothing to our distributed XML query processor in Chapter **5** though our scheme is adaptable to other architectures.

3.3.2 Query Parallelism

Several levels of parallelism can be found in (parallel) query processing [OV99].

- *Inter-query parallelism* enables the parallel execution of multiple queries generated by concurrent transactions in order to increase throughput.
- *Intra-query parallelism* consists of *inter-operator parallelism* and/or *intra-operator parallelism* and used to decrease response time. Figure 3.7 depicts the two kinds of operator parallelisms.
    1. *Intra-operator parallelism* executes the same operator by multiple processors, each one working on a subset of the data.
    2. *Inter-operator parallelism* executes several operators of the query tree on multiple processors.

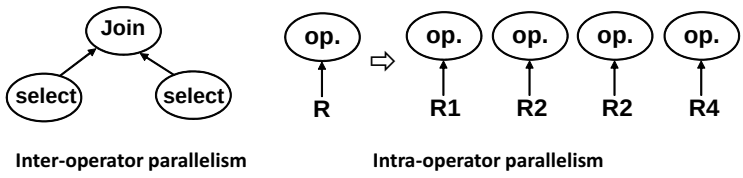


Figure 3.7. Query parallelism.

Two forms of inter-operator parallelism as shown in Figure 3.8 can be exploited:

- Within *pipeline parallelism* (also known as dependent parallelism), several operators with a producer-consumer link are executed in parallel.
- *Independent parallelism* is achieved when there is no dependency between the operators executed in parallel.

For instance, the operator OP_1 and the subsequent operators OP_4 and OP_5 in Figure 3.8 are concurrently exectuted with pipeline parallelism. On the other hand, OP_2, OP_3 and OP_4 are executed in parallel when independent parallelism is achieved.

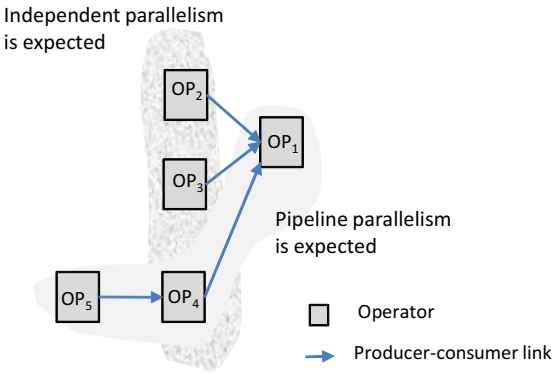


Figure 3.8. Inter-operator parallelism.

Independent operator parallelism is attractive because there is no interference between the processors. However, it is only possible for bushy execution and consumes more resources than pipelined processing [SYT93]. State-of-the-art solutions [LR05, SD90, CLYY92] of parallel databases tend to apply maximal pipelined parallelism in processing multi-join queries over multiple machines.

In Chapter **5**, we introduce the pipeline parallelism into distributed XML query processing. We show some reason why pipeline parallelism can be more effective for XML database systems than traditional parallel databases.

4

Scalable Storage System for Large XML Data

Scalability to data volumes is clearly important for database systems and also for XML databases. There is no doubt that database users prefer such a database system that is capable of handling as much volumes as possible.

To attain this, we propose an XML storage scheme based on Document Table Model (DTM) which expresses an XML document as a table form. When performing query processing on large scale XML data, XML storage schemes on secondary storage and their access methods greatly affect the entire performance. For this reason, we developed an XQuery processing scheme in which an XML document is internally represented as a set of DTM blocks and can be directly stored on secondary storage. We explore an efficient XML storing scheme — assuming a given situation that certain number of pages are accessed. It means that we put XML indices behind for the time being (in the discussion) for the fair assessment of XML storage schemes themselves.

There is a well-known rule thumb in database systems that it is better to scan than to use an index when more than 10% of the data is accessed; otherwise using indices should be considered. Not surprisingly, this principle also applied to XML database systems. May et al. confirmed that in [MBB+06]. We follow this principle and explore a non-oblivious problem that how XML fragments should be placed on a secondary storage for the situation that indices are not usable. Such situations are mandatory for distributed query processing due to difficulties in distributed optimization, and even for local query processing of complex XML queries.

Designing an XML storing scheme requires design decisions. We made, in this dissertation, our scheme tailored for read-oriented workloads; an XML document is

stored on disks as arrays of nodes. We analyzed the actual data access patterns to DTM that appeared in processing XML queries, and employed the combination of informed prefetching and scan-resistant buffer management based on the analysis. Our experimental results showed that our storage scheme outperforms competing schemes with respect to I/O-intensive workloads, and our sophisticated prefetching and caching increase overall throughput without significant drawbacks.

4.1 Background

Recently increasing use of XML has heightened the need for storing and querying large amount of XML data efficiently. Previous researches have mainly been focused on indexing paths and optimizing XML queries. On the other hand, an underlying storage representation significantly impacts on XML query processing, and thus, it is important to explore storage schemes for XML documents.

Several storage schemes have been proposed for XML documents [TDCZ02, MLLA03, KM00, ZKO04]. However, to the best of our knowledge, there has been no careful study on the actual data access patterns of XML query processing (e.g., XQuery). It has still been an open issue as to which strategy is suitable for XQuery processing.

In designing our storage scheme, we made the following architectural decisions.

- Our scheme aims to be tailored for read-only and/or read-most workloads. This is based on the fact that XML databases are often required for managing existing XML documents received from other organizations (e.g., Electric Data Interchanges). In such situation, node-level updates (i.e., DML operations) are not always required. However, document-level updates (i.e., bulk insertions/deletions and document replacement) are considered to be required. Read-optimized database design has been suggested for relational database systems, such as [HLAM06], but not well studied for XML database systems. Therefore, we propose an efficient XML database system optimized for read-oriented workloads.

- We focus on iterative XQuery processing in which an operator tree consists of iterators. This is because many XQuery processors [FHK+04, Sax], including our implementation, employ an iterator model and a *tuple-at-a-time* semantics instead of a *set-at-a-time* semantics.

Considering the above aspects, we propose an XML storage scheme based on Document Table Model (DTM) which expresses an XML document as a table form. A DTM table is internally represented as an array of DTM blocks, so that it can be directly stored on secondary storage. This straightforward approach enables effective prefetching of DTM blocks. However, it is known that there are interactions between prefetching and caching, and traditional cache replacement policies like LRU do not work well with prefetching [CFKL95]. It is because prefetched disk blocks need to be stored on the cache, and prefetched entries can potentially compete for (hot) cache entries. On the other hand, the benefit of prefetching and caching can coexist by using a scan-resistant cache replacement policy in certain situations citebutt05. To deal with this problem, we conducted the combination of informed (i.e., directed) prefetching and scan-resistant caching.

We have implemented a native XML database system, named *XBird*, using the proposed storage scheme. While XBird supports indices, in this chapter, we focus on XML storage schemes and their access methods without indices to evaluate the performance. Since XML queries require them for string-value calculation and serialization, the performance often depends on the basic access methods.

Our experimental results showed that our storage scheme outperforms competing schemes under I/O-intensive workloads, and our sophisticated prefetching and caching increase overall throughput without significant drawbacks.

The rest of the paper is organized as follows: in Section **4.2**, we introduce a logical design of DTM. In Section **4.2.3**, we give an analysis of data access patterns in XQuery processing. Section **4.3** presents our physical storage scheme and its access methods. In Section **4.4**, we provide experimental results and their evaluation. We introduce related work in Section **4.5** and conclude in Section **4.6**.

4.2 Logical Data Structure

4.2.1 Document Table Model

An XML document is represented as a variant of Document Table Model (DTM) in our proposed scheme. DTM was originally used in Apache Xalan XSLT processor [Apac]. It expresses an XML document as a table form, while previous Document Object Model

(DOM) regards an XML tree as an object tree.

Since DTM table consists of primitive data types, DTM can avoid footprints of objects, such as object instantiation and memory consumption, which are mandatory to DOM. Therefore, popular XQuery/XPath processors [Sax, Apac] adopt either DTM or a similar internal data structure to DTM. However, these processors do not consider use of secondary storage to manage large scale XML data. To deal with this issue, we aim at a natural extension of DTM by taking account of secondary storage.

4.2.2 Internal Organization

Figure 4.2 shows an overview of our system organization. Figure 4.1 depicts an example of an XML tree labeled in depth-first search manner. In Figure 4.1, the symbols *E* and *T* indicate element nodes and text nodes, respectively. The upper half of Figure 4.2 represents the DTM associated with the XML tree illustrated in Figure 4.1.

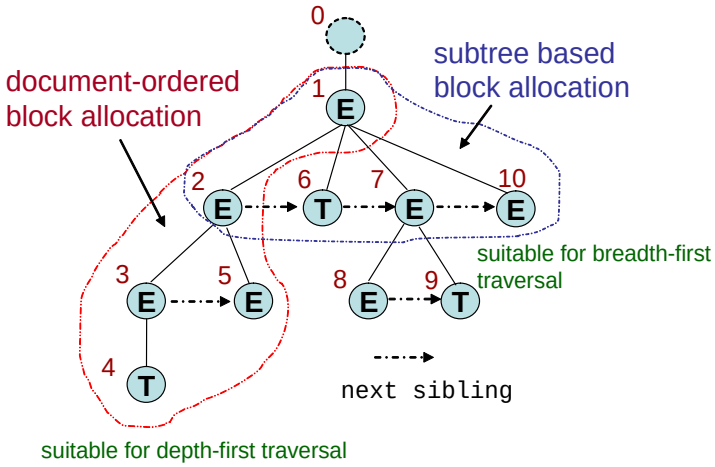


Figure 4.1. An XML tree labeled in depth-first order and its fragmentation examples.

A DTM table consists of four integer arrays (see Figure 4.2). An index of these arrays indicates a node handle, and thus, XBird requires 16 bytes per an XML node when an integer is 4 bytes long.

The first *TYPE* row represents a node type, i.e., either *E* (abbreviation of element) or *T* (abbreviation of text), and the following attributes: FIRST_ENTRY flag,

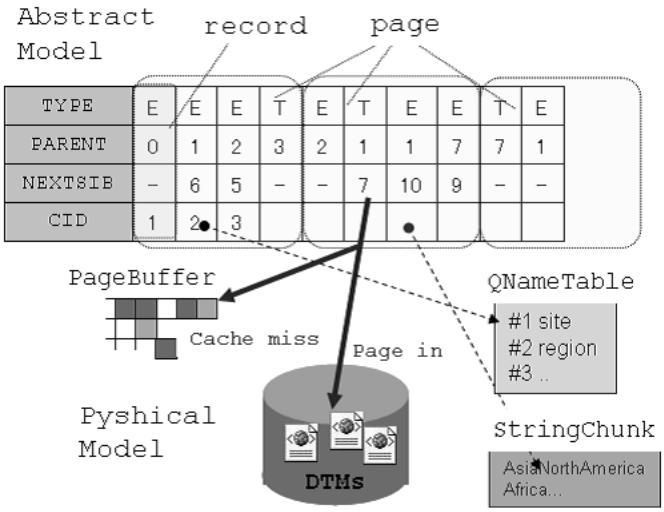


Figure 4.2. Internal organization of XBird.

LAST_ENTRY flag, HAS_CHILD flag, the number of namespace declarations, and the number of child elements. The FIRST_ENTRY and LAST_ENTRY flags are used to determine whether a sibling node exists on the left or right. The HAS_CHILD flag denotes whether the node has child node(s) or not. All these data are compacted and stored into an integer in a bitwise manner, while node types are only shown in Figure 4.2 for sake of simplicity, though.

The second *PARENT* and the third *NEXTSIB* rows represent the index to parent nodes and that to the next sibling nodes, respectively. The fourth row keeps a content ID (*CID*) which indicates a unique identifier for QName and character string.

Character strings in XML data are converted into chunks. A CID is attached to each string, and then the strings are managed by the string management module, named *StringChunk*. If the length of a string exceeds the system threshold (512 bytes by default in our current implementation), the string is compressed by LZF algorithm [Leh] so that memory consumption can be suppressed.

QNames are managed in a unit of a collection which corresponds to the directory of file system to improve space efficiency. We call the QName management module *QNameTable*. A QNameTable generally fits in memory. Thus, it is mapped to memory while processing the corresponding collection.

QNames are managed in a unit of a collection which corresponds to the directory of file system to improve space efficiency. We call the QName management module *QNameTable*. A QNameTable generally fits in memory. Thus, it is mapped to memory while processing the corresponding collection.

4.2.3 Access to DTM

The access to a DTM table is based on operations to acquire a numerical value related to a specified node, such as a CID or a parent value. For example, QName and character string are acquired from QNameTable and StringChunk, respectively, by using a CID as a key. Axis processing is based on offset calculation by referring to attributes such as parent and next-sibling values. Thus, the logical data structure, i.e., DTM, and its operations are equipped in the query processor.

All axis operations can easily be implemented by the combinations of the following five core functions in our DTM variant. The functions *firstChild*, *lastChild*, *nextSibling*, *parent*, and *previousSibling* are used to obtain youngest child, eldest child, next elder brother, parent, and younger brother, respectively. The algorithms of frequently used functions *firstChild*, *nextSibling*, and *parent* are shown in Figure 4.3. Using these functions, axis processing, such as parent, child, next-sibling, etc, can be implemented as a simple offset calculation.

Note that, in Figure 4.3, the function *getCol* acquires an array element specified by a node handle. The function *hasChild* judges whether the node has a child or not by using the bit flag. The functions *getNamespaceCount* and *getAttributeCount* acquire the number of namespace declarations and attributes, respectively. The constant PER_NODE shows the number of array elements consumed by each node. The remaining constants PARENT_OFFSET and NEXTSIB_OFFSET represent the offsets to identify the location of the parent and next-sibling values, respectively.

Analyzing Page Access Pattern

In order to design an efficient XML storage and their access methods, we preliminarily analysed access patterns to DTM. The key issues to design the physical layout for effective XML query processing are:

- Mapping DTM to secondary storage for iterative XQuery processing, and

Algorithm Algorithm of primary axis accesses

```
const PER_NODE = 4;
const PARENT_OFFSET = 1; const NEXTSIB_OFFSET = 2;
```

(a) Algorithm of *firstChild* function

IN: context node identifier OUT: identifier of the first child node

```
function firstChild(curnode) {
  code := getCol(curnode);
  if(!hasChild(code))
    return nil;
  namespaces := getNamespaceCount(code);
  attributes := getAttributeCount(code);
  addr := curnode + ((namespaces + attributes) + 1) * PER_NODE;
  return addr;
}
```

(b) Algorithm of *nextSibling* function

IN: context node identifier OUT: identifier of the next-sibling node

```
function nextSibling(curnode) {
  return getCol(curnode + NEXTSIB_OFFSET);
}
```

(c) Algorithm of *parent* function

IN: context node identifier OUT: identifier of the parent node

```
function parent(curnode) {
  return getCol(curnode + PARENT_OFFSET);
}
```

Figure 4.3. Algorithm of primary axis accesses.

- Paging strategy between main memory and secondary storage.

Our solution to the former issue is to store nodes in document order, and for the second issue is to employ aggressive prefetching I/Os. Here after, we discuss these key issues by using the experimental results of 20 XQuery queries in XMark benchmark [SWK+01]. The complete list of XMark queries are shown in Appendix A.

In the experiment, as we set page size to 2KB, a page has 512 DTM rows (see Figure 4.2). As for the experimental data, we used 113MB of the XML document generated by XMark where the scale factor (SF) is 1.

Figure 4.4 shows an overview of page access patterns of XMark queries. The inset figure within Figure 4.4 depicts the data access patterns of Q9 and Q10 both of which show peculiar page access patterns.

The vertical and horizontal axes of the figure denote the page ID being accessed and time where accessing one page is a unit of time. Since page IDs are numbered in document order, the page with the lowest ID contains the document root and its neighbors, and the last node is allocated in the page with the highest ID. In other words, the document order is reflected on the vertical axis. Seeing in Figure 4.4, access patterns from lower left to upper right are a common tendency among 18 queries, which means that pages are requested in document order. Note that this sort of access pattern is not unique to XMark queries, but XBench [YÖK04] queries (both TC/SD and DC/SD) had also indicated similar access patterns (see Figure 4.5).

4.2.4 Page Replacement Policy

This section describes how page replacement policy affects XML query processing with giving an example, Q9. Q9 contains a triple nested-loop, and thus, its locality of reference tends to be low. Therefore, we could not make the best use of the locality of reference, and it took long time to process Q9 due to paging overhead. A natural question is what algorithm is suitable for the buffer management of XML query processing. LRU is widely used as a buffer replacement policy even in XML databases [MLLA03, KM00]. However, it is known that LRU does not work well for sequential scans and large Inter-Reference Gaps (IRGs), while such sequential scans often appear in serialization and string-value calculations in XML query processing. Moreover, standard LRU is uncongenial to prefetching [CFKL95].

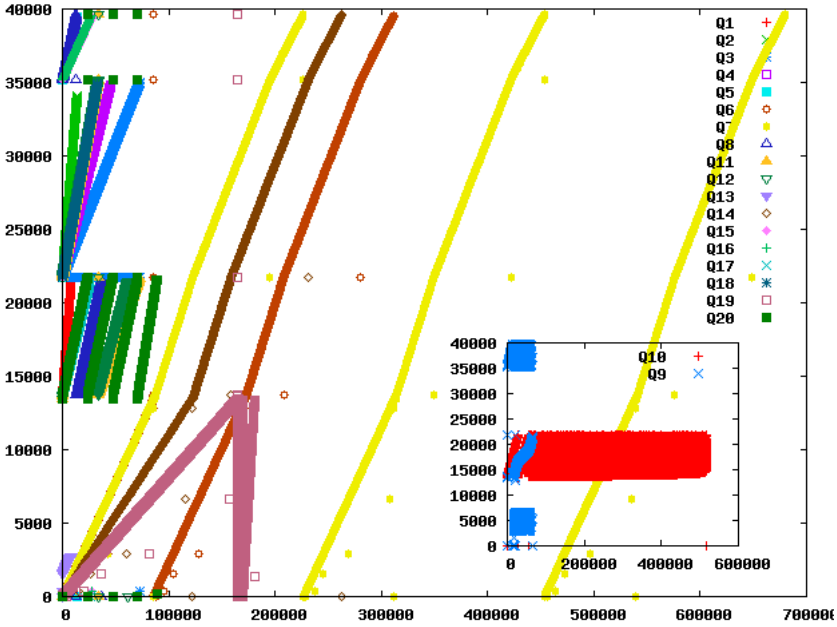
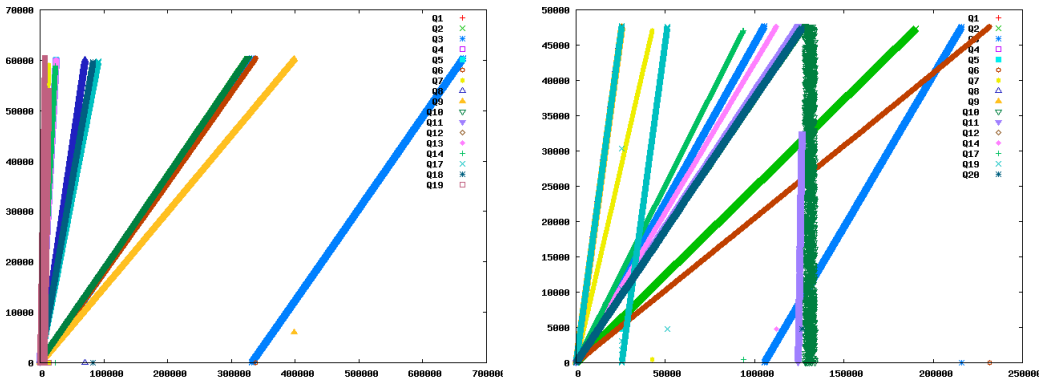


Figure 4.4. Page access patterns of XMark queries.



(a) TC/SD (b) DC/SD

Figure 4.5. An overview of page access patterns of XBench queries.

We compared the efficiency of LRU with that of *scan-resistant* 2Q [JS94] with prefetching. The scan-resistant property prevents large sequential data scans from flooding the buffer pool, which would otherwise victimize pages that are more frequently referenced, e.g., index root pages. Ideally, the index root pages should be kept and victimize the scan pages instead, even if the latter is more recently referenced. 2Q achieves a scan-resistance by using one FIFO queue $A1_{in}$ and two LRU lists, $A1_{out}$ and A_m. When first accessed, a block is placed in $A1_{in}$; when a block is evicted from $A1_{in}$, its identifier is inserted into $A1_{out}$. An access to a block in $A1_{out}$ promotes this block to A_m. The result comparing LRU and 2Q for the buffer management algorithm of our XML database showed that 2Q brought 10% on average and a maximum of 23.5% better performance than LRU where SF is 10 for the XMark benchmark. Therefore, we use 2Q in XBird as a page replacement policy.

4.3 Physical Storage

4.3.1 Storage Scheme

XBird internally treats XML documents as DTM. If DTM is applied to the fundamental data structure of an XML database, it needs to be persistent. In order to have DTM persistent, it needs to be extended to secondary storage. A simple solution is to decompose the DTM into multiple blocks. Then, paging these blocks has only to be performed between main memory and secondary storage, as shown in the lower half of Figure 4.2.

In our proposed system, each block of DTM is stored on the secondary storage in document order. Here after, we call our proposed scheme *persistent-DTM*, *pDTM* for short, which follows the well-known persistent-DOM (PDOM).

Since the block allocation policy of pDTM is based on document order, it does not always place adjacent nodes to the same or nearby block. A certain parent and its second or higher children might be taken apart on secondary storage. The reasons why we chose this allocation policy are as follows:

- According to the analysis in Section **4.2.3**, access patterns in document order appear frequently.

- Node allocation in document order is suitable for string-value calculation and serialization which are mandatory for XML query processing.

Our XQuery processor employs an iterative query processing model based on an iterator tree, which is similar to BEA/XQRL XQuery processor [FHK+04] and Saxon [Sax] in which pipelined processing operators deal with loops, axis accesses, etc. Since queries are processed in operators in a tuple-at-a-time fashion, linear accesses in document order are found in Figure 4.4.

For example, during a query evaluation of 'site/regions' pipelined XQuery processor accesses to nodes are made in the following order: 'site[]/regions[1]', 'site[1]/regions[2]', .., 'site[last()]/regions[last()]'. Thus, document-ordered storage model, i.e., pDTM model, fits for iterative query processing more efficiently than other strategies for most queries. In contrast, subtree-based storage model is not suitable for depth-first traversal (see Figure 4.1).

4.3.2 Physical Layout

The DTM table explained in Section **4.2.2** is internally formed as a two-dimensional array (more specifically, Iliffe vector [Ili61]). The arrays of the second dimension consist of pages. Each element of the first dimension holds a pointer to each page. Since the pages that are not paged-in to main memory are expressed as *null* values, the skeleton, i.e., the first dimension, does not waste memory space.

The physical structure of DTM is divided into three layers as shown in Figure 4.6. A DTM row stores a record which is identical to information of a node. A page is the minimum accessible data-unit of an I/O operation, which corresponds to a disk block.

4.3.3 Physical Access

A page-in operation is performed only when a requested DTM page does not exist in the page buffer (see Figure 4.2). For a simple implementation of paging, a hook is inserted just in front of *getCol* function, shown in Figure 4.3. The algorithm of the hook for paging is shown in Figure 4.7.

The following four paging profiles are currently defined. FORWARD profile is the default strategy that looks-ahead the forward direction. REVERSE profile is used

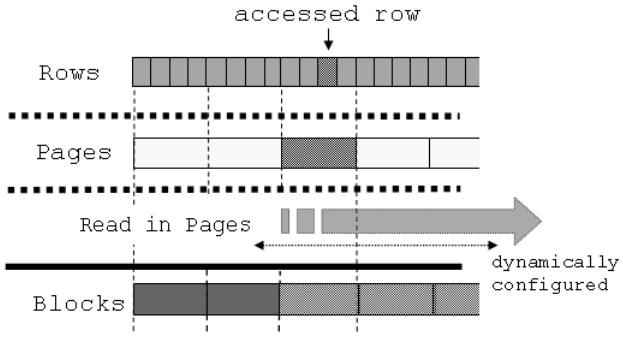


Figure 4.6. Physical structure of pDTM and its prefetching.

for reverse-axis traversals which request nodes in reverse document order (e.g., preceding). INDEX profile is for index lookups. SERIALIZE profile is for retrievals of subtrees at serialization.

4.4 Experimental Evaluation

We implemented XBird in Java. In order to reveal the potential performance of XBird and compare it to competing schemes, we used the XMark benchmark suite [SWK+01] for evaluation. The experimental setting commonly used in this chapter is as shown in Table 4.1.

CPU	Intel Pentium D 2.8GHz
OS	SuSE Linux 10.2 (Kernel 2.6.18)
RAM	2GB
Hard Disk	SATA 7200rpm
Java	Sun JDK 1.6
JVM option	-server -Xms1400m -Xmx1400m
Buffer size for DTM paging: 128MB	
Cache size used by StringChunk module: 32MB	
DTM page size	2KB
FORWARD	readForwards: 32, readBackwards: 0
SERIALIZE	readForwards: 64, readBackwards: 0

Table 4.1. Experimental setting.

Algorithm Page-in Algorithm

IN: rowId OUT: page

```
1. const PAGE_SHIFT := 9;
# following items of paging profile are dynamically configured
# by giving hints from the query processor.
2. readForwards := 32, readBackwards := 0;
# The hook to the getCol function.
3. pageAddress := rowId >> PAGE_SHIFT;(a)
4. page := PAGE_BUFFER.get(pageAddress);(b)
5. if(page == nil) page := readInPages(pageAddress);(c)
6. return page;
7. function readInPages(pageAddress)(d) {
8.    fromPage := pageAddress - readBackwards;
9.    toPage := pageAddress + readForwards;
10.   for(k:=fromPage; k<=toPage; k++) {
11.      page := readIn(k);
12.      if(i == pageAddress) requiredPage := page;
13.      PAGE_BUFFER.putIfNotFound(k, page);
14.   }
15.   return requiredPage;
16. }
```

a) Calculate a page address from the requested row ID.

b) Retrieve the requested page from the page buffer.

c) If 'page' value is *nil*, then call the function *readInPages* to retrieve pages.

d) Retrieve pages from disk based on a paging profile. All pages that are read from disk are placed to the PAGING_BUFFER using a page address as a key.

Figure 4.7. Page-in Algorithm.

4.4.1 Comparison to Subtree-based scheme

In order to evaluate storage techniques themselves, we compare XBird with Natix [KM00](version 2.1.1) by using queries shown in Table 4.2 (These queries are introduced in XPathmark [Fra05]). Natix is a native XML database system implemented in C++, and supports XPath 1.0. The unique feature of Natix is that it adopts a subtree-based storage block allocation strategy. XBird was configured not to use indices as to make it a fair comparison of XML storage methods, because Natix 2.1.1 does not have indices.

The summarized results where the SF is 5 and 10 are shown in Figure 4.8 and Figure 4.9, respectively. Now, we focus on Q2-Q14 and Q17 for discussion.

An important difference appears in the results of Q6, Q7 and Q14 which contain '//'. Because '//' requests a lot of blocks in a depth-first manner, the efficiency of handling blocks, such as paging and buffer management, tends to appear remarkably. For Q2, Q5, Q14 and Q17 whose outputs are relatively large, pDTM achieves better performance than Natix because of the efficiency of serialization. Our storage scheme is efficient for sterilization because serialization generally requires depth-first assess of XML trees.

Natix shows a slightly good performance for Q4 which contains following-sibling axis, due to its subtree-based physical layout. Since child nodes are brought together in Natix, the following-sibling axis can be processed efficiently. Recall that subtree-based layout is suitable for breadth-first traversal of XML trees, but not for depth-first traversal.

Both of Natix and our scheme takes block-level storing scheme according to page size (typically 4K/8K bytes) and both of them do navigational accesses of XML trees. We thus consider that the difference between Natix and our scheme mainly due to page allocation scheme on secondary disks.

4.4.2 Performance Comparison with respect to Data Volumes

We verify, in this section, our XML storage scheme with respect to its scalability to data volumes. Table 4.3 is the results of the performance evaluation using XMark benchmark dataset [SWK+01]. We used four scale factors from SF=1 to SF=10 for evaluating scalability to data volumes. The results are also depicted in Figure 4.10. The notation $\times N$, e.g., $\times 3$, $\times 5$ and $\times 10$ in Figure 4.10, represents N fold to the value where

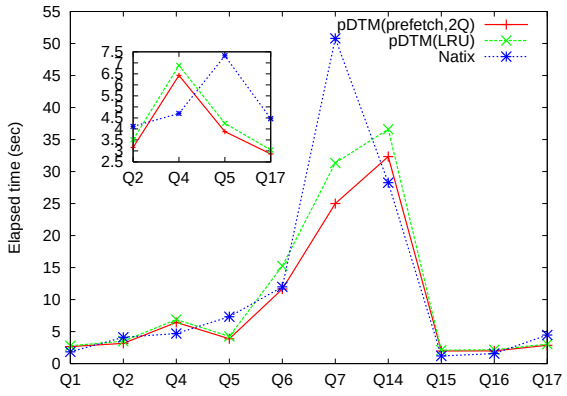


Figure 4.8. Performance of XMark SF=5.

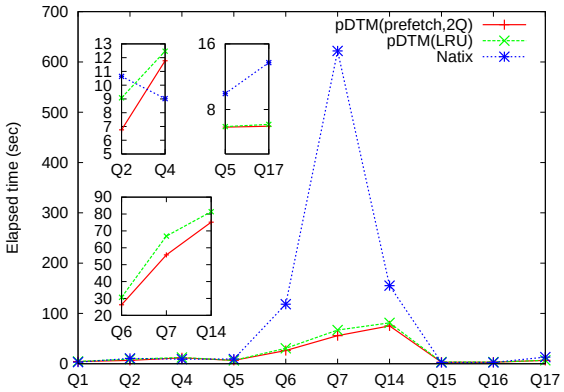


Figure 4.9. Performance of XMark SF=10.

Query Id	Output size SF5 / SF10	Query expression
Q1	15 / 12	/site/people/person[@id = ”person0”]/name/text()
Q2	241K / 482K	/site/open_auctions/open_auction/bidder[1]/increase/text()
Q4	0 / 0	/site/open_auctions/open_auction[bidder[personref/@person = ”person20”]/following-sibling::bidder [personref/@person = ”person51”]]/reserve/text()
Q5	666K / 1.3M	/site/closed_auctions/closed_auction[price/text() >= 40]/price
Q6	6 / 6	count(//site/regions//item)
Q7	6 / 6	count(/site//description | /site//annotation | /site//emailaddress)
Q14	149K / 297K	/site//item[contains(description, ”gold”)]/name/text()
Q15	42K / 80K	/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text/emph/ keyword/text()
Q16	9.3K / 20K	/site/closed_auctions/closed_auction[annotation/description/parlist/listitem/parlist/listitem/text/emph/ keyword/text()]/seller/@person
Q17	897K / 1.8M	/site/people/person[homepage/text()]/name/text()

Table 4.2. XPath queries converted from XMark queries.

SF=1.

Let us focus on the overall tendency appears in Figure 4.10. Most of queries excepting Q11 and Q12 showed an apparently good scalability property. The reason why Q11 and Q12 does not scale is because they involve θ-joins. Opposite to the fact that equijoins can be optimized efficiently using hash-joins, θ-joins tends to be slow in general. Our current implementation of θ-joins remains optimization opportunities while a non-equality condition of θ-join often makes a query CPU-intensive.

	113MB (SF=1)	340MB (SF=3)	568MB (SF=5)	1.1GB (SF=10)
Q1	1.235	2.11	2.922	4.828
Q2	1.657	3.141	5.47	8.907
Q3	2.62	4.281	5.344	11.375
Q4	2.531	5.703	9.406	17.25
Q5	1.703	2.89	5	7.813
Q6	3.969	10.797	17.375	52
Q7	8.78	23.172	39.344	102.547
Q8	2.5	5.578	8.672	20.484
Q9	5.875	18.672	39.344	90.703
Q10	19.328	68.672	142.562	333.703
Q11	8.485	50.734	135.391	522.703
Q12	6.625	35.453	90.641	332.94
Q13	1.656	2.828	5.844	8.75
Q14	7.172	17.625	44.734	72.515
Q15	1	1.547	2.63	9.61
Q16	1.109	1.781	2.5	4.359
Q17	1.515	2.797	4.78	7.343
Q18	2.62	4.47	6.313	12.391
Q19	6.31	23.125	44.172	101.312
Q20	2.547	6.688	10.891	24.609

Table 4.3. Performance of pDTM on different data volumes (in sec).

4.5 Related Work

A few native XML storages have been studied. OrientStore [MLLA03] proposed a schema-guided storage method. Their strategy called Logical Partition-Based Clustering utilizes schema information, which clusters XML data into schema blocks to reduce I/Os required for path processing of XML queries. Though this storage technique is effective for path processing, it is not efficient for string-value calculation and serialization which require document ordering.

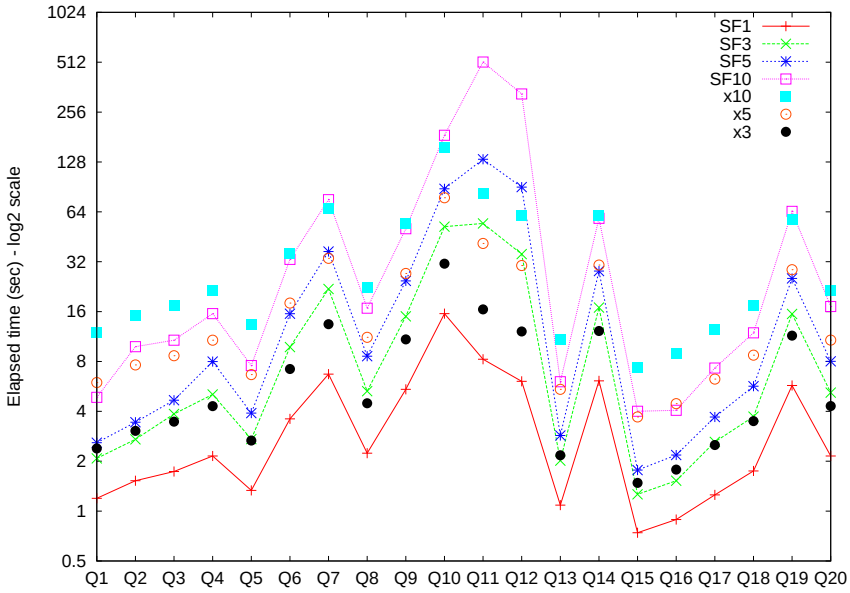


Figure 4.10. Scalability of pDTM.

Natix [KM00] is a well-known native XML database which employs a subtree-based storage scheme. It divides an XML tree into subtrees based on the physical page size, so that each subtree fits into a page. Each page keeps the pre-order property of the subtrees on secondary storage.

Zhang et al. proposed a fast tree pattern matching algorithm, called *next-of-kin* (NoK) pattern matching, and a succinct XML string representation scheme, called *subject tree* [ZKO04]. In NoK, each page of subject trees is stored on secondary storage in the pre-order of XML trees. Though NoK supports simple parent-child queries, they have mentioned neither prefetching I/Os nor sophisticated buffer management.

On the other hand, non-native XML storages have been studied well, for example, in [TDCZ02].

4.6 Summary

In this chapter, we proposed an efficient XML storage scheme based on DTM for iterative XQuery processing. We also analyzed access patterns that frequently appear for

XML queries. Our experimental results showed that our storage scheme outperforms competing schemes in the certain situation where lots of pages are required such as queries contains '//' or when serialized results are relatively large. Furthermore, our enhancements (i.e., prefetching and scan-resistant buffer management) improved the performance of query processing by 10% on average and by 23.5% at maximum in our experiments. These results demonstrate the importance for XML database systems to take informed prefetching and scan-resistant caching into consideration.

Issues to be explored include realization of automatic database tunings such as buffer replacement policy and prefetching strategy based on the online analysis of access patterns.

5

Scalable XML Processing with a Shared-nothing Cluster

As described in Chapter **3**, shared-nothing systems are promising approach in most settings. We assume this architecture as the primary target of our distributed XML database. However, there is an obstacle that shared-nothing involves cost-ful message passing.

To overcome this obstacle, in this chapter, we focus on an aspect of distributed XQuery processing that involves data exchanges between processor elements. Needless to say, efficiency of data exchange between processor elements becomes extremely important in shared-nothing systems.

We first address problems of distributed XML query processing and explain how the problems differ from traditional database problems. Then, in order to achieve efficient and transparent data exchange, we adopt the use of *remote proxy*, in which each shipped data is wrapped in a proxy sequence, and the proxy sequence is returned to the remote peer. When accessing the proxy sequence, actual results (possibly partial results) are fetched from a data provider, and then the data provider evaluates its entity sequence in a *call-by-name* fashion. Our scheme allows parallel query execution and reduces network traffic and redundant buffer utilization by exchanging required data directly between a consumer and a provider.

Terminology

Evaluation strategy

- *call-by-name*. Our XQuery processor takes call-by-name for the base evaluation strategy. When calling a function, the argument expressions are passed unevaluated and evaluated lazily.

Parameter passing mechanism of Remote Procedure Call (RPC)

We, in this chapter, use the term *pass-by-xxx* for the parameter passing mechanism of RPC in sharing the context with distributed object technologies. The terminology is different from one of evaluation strategy [BLS97].

- *pass-by-value*. We use this term when all parameters of a remote procedure are eagerly evaluated.
- *pass-by-reference*. We use this term when all parameters are passed as remote objects. The remote objects hold references to their entity objects. The entity objects are lazily evaluated though procedure calls of the corresponding remote objects.

5.1 Background

As a result of wide adoption of XML, XML data has been spread over computer networks. It has produced the need to integrate distributed and dynamic XML documents. For example, users might want XML feed-readers to show more fresh and/or on-the-fly information (e.g., recent disaster information or latest results of sport games) through their thousands of RSS/ATOM subscriptions. However, current feed-readers do not aggregate their subscriptions in real-time but at hourly intervals (Bloglines [IAC] currently checks subscriptions once an hour). Another example is found in integration of biological databases. Most biological databases today have the ability of publishing XML to support integrations among heterogeneous data-sources, and each of them receives frequent updates/corrections from individual laboratories around the world. Therefore,

integration systems of biological databases must take data freshness into account.

Considering such situations, it is doubtful that on-the-fly processing for thousands of XML data is realistic. Because, as far as we know, there is no XML query processor tackling both inter-operator parallelism [OV99] and distributed query processing, both of which are indispensable to solving the underlying problems as explained below.

To achieve on-the-fly XML query processing of thousands of XML data that cannot be processed by an XQuery processor on a single computation node, we apply a divide-and-conquer approach that divides a query into sub-queries, and then, executes these sub-queries on multiple computation nodes as in Figure 5.1. However, such a hierarchical distributed system must deal with the following technical issues:

- First, partitioned computation requires that query processors exchange intermediate results in order to produce its final result. The time for exchanging intermediate results cannot be ignored. In particular, data exchanging of XQuery Data Model (XDM) [W3Ce] instances requires long CPU time because, in contrast to relational databases, XML databases must deal with non-scalar data types such as XML trees, which basically treat scalar data types. Therefore, encoding and decoding of XDM instances, in general, spend more CPU time than those for a relational model. According to the findings in [NJ03], parsing an XML document typically takes 175K CPU instructions per kilobyte, which is as of the same order as inserting a row into a relational table (30K to 200K instructions). Hence, each edge of two operators can potentially be a bottleneck.

- The second issue is that CPU-utilization of current query processors cannot exploit *inter-operator parallelism* in addition to *divide-and-conquer parallelism*. *Inter-operator parallelism* consists of *pipeline parallelism* (a.k.a. *dependent parallelism*) and *independent-operator parallelism*. With pipeline parallelism, several operators in a producer-consumer relationship are executed in parallel. Taking Figure 5.1 as an example, q_g that follows q_y, $\cdots$ and q_z are executed concurrently with a pipeline processing. Pipeline processing enable that q_g proceeds its execution before the termination of the child execution (q_y, $\cdots$ and q_z). On the other hand, *independent-operator parallelism* is achieved when there is no dependency among operators executed in parallel. This form of parallelism is attractive because there is no interference between processors. However, its

adoption is only possible for bushy execution and may consume more resource bursty [SYT93, OV99].

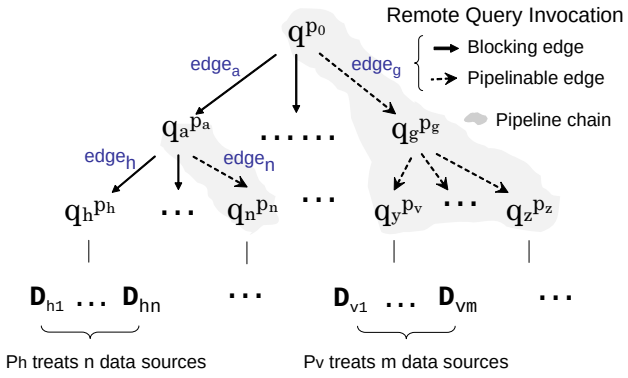
Regarding the divide-and-conquer strategy, a sub-query q_a can execute its sub-queries ($q_h \cdots q_n$) in parallel and even out-of-order (see Figure 5.1). Here, we use the notations $T(q_a)$, $LT(q_a)$ and $T(edge_a)$ for elapsed time of a query q_a at peer p_a, that of local query processing of q_a, and that at an edge $edge_a$, respectively. When we do not consider pipeline parallelism, $T(q_a)$ is recursively defined as follows:

$$T(q_a) = \underbrace{max((T(q_h) + T(edge_h)), \cdots, (T(q_n) + T(edge_n)))}_{\text{elapsed time of the most time-consuming edge}} + LT(q_a)$$

According to this formula, computation time of a node depends on the most time-consuming edge. Taking Figure 5.1 as an example, local query processing of q^{p_0} is blocked and its CPU resource tends to be idle until the last intermediate result is returned. Moreover, the non-parallelized part of queries, e.g., $LT(q_a)$ in the above example, restricts the theoretical maximum speedups. According to Amdahl's law [Amd00], the expected speedups by parallel query execution are often limited by the non-parallelized part. Therefore, to leverage the computation power of current multi-processors including multi-core processors, pipelining is indispensable.

Considering the above aspects, in this chapter, we focus on the data exchanges between processor elements in distributed XQuery processing, which so far have not been carefully discussed in the literature. As an alternative to previous methods, we propose an efficient data exchange method using *remote proxy*, in which each entity or result sequence is wrapped in a proxy sequence and the proxy is returned to the remote peer. When accessing the proxy sequence, actual results (possibly partial results) are fetched from the data provider. The use of *remote proxy* brings the following three advantages:

- Our scheme allows parallel query execution, which supports several kinds of parallelisms, e.g., independent-operator and pipeline parallelism.



A user query q is divided into sub-queries $q_a, \cdots, q_z$ recursively. The symbols $p_0, p_a, \cdots, p_h, \cdots, p_z$ denote the peers in which a query is executed. The symbol D such as D_{hn} represents a data-source.

Figure 5.1. Divide-and-conquer and pipeline parallelism.

- Entity sequences are computed in a demand-driven manner. A new FIFO entry of the entity sequence [1] is requested when the entry is consumed and dropped to the low watermark level. Its sophisticated and built-to-order mechanism utilizes the server's resources such as memory and CPUs.

- Our method can reduce network traffic and redundant buffer occupation by directly exchanging required data between a consumer and a provider, which are mandatory for the previous *pass-by-value* data exchanges [RBHS04, ZB07b, FJM+07b]. Avoiding intermediary trades reduces both network traffic and network latency as well as redundant computations such as encoding and decoding in mediator nodes.

We have implemented the proposed method on the top of our practical implementation of a native XML database system. Our experimental results show up to 22x speedups compared with competing methods. We demonstrate the importance for distributed XML database systems to take *pass-by-reference* semantics into (re-)consideration.

[1] In XDM [W3Ce], a result is a sequence containing zero-or-more items. The items are consumed by an operator (iterator) in a FIFO manner. We call the remaining items "FIFO entry."

The rest of this chapter is organized as follows: Section **5.2** introduces related works and identifies open problems of distributed XML query processing, and then, we briefly mention our solutions for the problems. In Section **5.3**, we describe details of our implementation including our language extension to XQuery and distributed query optimizations. In Section **5.4**, we provide experimental results and their evaluation, and conclude in Section **5.5**.

5.2 Open Problems and Our Solution

In this section, we address open problems in taking *pass-by-value* strategy for the parameter passing of distributed XML query processing, and propose our solution for each problem. We also refer to related work.

Figure 5.2 depicts a typical remote query execution flow with *pass-by-value* semantics. *Pass-by-value* semantics has been used for the evaluation strategy of previous distributed XML query processors [RBHS04, ZB07b, FJM+07b]. The problems underlying these strategies are as follows:

Limited inter-operator parallelism

With a *pass-by-value* evaluation strategy, a remote query is executed sequentially as in Figure 5.2. Therefore, during processing on Server A, Server B becomes idle. In contrast, Server A tends to be idle while Server B processes a query. In addition, as we mentioned with Figure 5.1, current XML query processors using *pass-by-value* strategy cannot exploit pipeline parallelism, and thus, inter-operator parallelism is limited for nested query executions.

We examined how long of a query processing time it takes to execute the following query at a remote peer where $doc locates an XML document of scale factor (SF) 5 or 10 of XMark [SWK+01]. The transferred data size of the resulting XDM instances were 2164 KB (SF=5) and 4307 KB (SF=10).

```
for $a in $doc/site/closed_auctions/closed_auction
where $a/price/text() <= 40 return $a/price
```

The result, as in Figure 5.3, shows that, at least in our experience, the latency

including encoding and decoding is the same as the query execution time at a remote peer. This supports our claim that each edge of Figure 5.1 can be a potential bottleneck.

To solve this problem, we take, in Section **5.3.2**, an advantage of pipeline parallelism and inter-operator parallelism into our *remote proxy*. Due to the *remote proxy*, our systems can make each edge of Figure 5.1 pipelinable, and thus, all computation nodes can theoretically run in parallel. Moreover, we ensure much *inter-operator parallelism* into our scheme by introducing an *asynchronous pipeline* processing in Section **5.3.2.1**.

Poor resource utilization

In a multi-user environment, multiple concurrent queries consume lots of system resources. Thus, it is important to allocate adequate CPU and memory resources especially under current multi-processor or multi-core architectures. For example, selecting low degrees of an operator parallelism can lead to under-utilization of the system and reduce throughput. On the other hand, high degrees of parallelism can spend "too many" resources on one query and lead to high resource contention. With the current *pass-by-value* strategy, such resource contentions frequently occur, for instance, at A and B in Figure 5.2. Suppose that encoding entire parametric sequences at A. Server A consumes lots of CPU cycles and memory space, and then remote query execution at Server B is blocked until the encoding finishes. On decoding receiving parameters at Server B, Server B may suffer from less available memory. Such a situation actually happened in our later experiment in reality (see Section **5.4**). To deal with the problem, we propose an efficient resource utilization scheme using *remote blocking-queue* with which processing rates of operators are managed.

Encoding and decoding overhead

Previous research used XML formats for data-exchanges between processor nodes [RBHS04, ZB07b, FJM+07b]. However, as mentioned in [NJ03], decoding XML inputs consumes lots of CPU cycles. In [BGJM04], Bayardo et al. asserted that binary encoding of XML would appear to provide performance benefits to most applications without any significant drawbacks other than compromising a view-source principle. On the contrary, a naive (blocking) binary encoding may prevent pipelined XML stream processing.

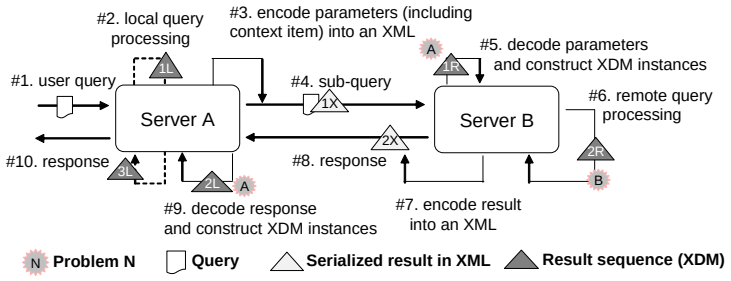


Figure 5.2. A typical remote query execution flow with pass-by-value semantics.

We thus take an incremental encoding/de-coding scheme that incrementally converts an XDM instance to a SAX-like event (binary) stream. In addition, we propose an efficient *direct result forwarding* mechanism for the *pass-by-reference* in Section **5.3**.

5.3 Implementation of XBird/D

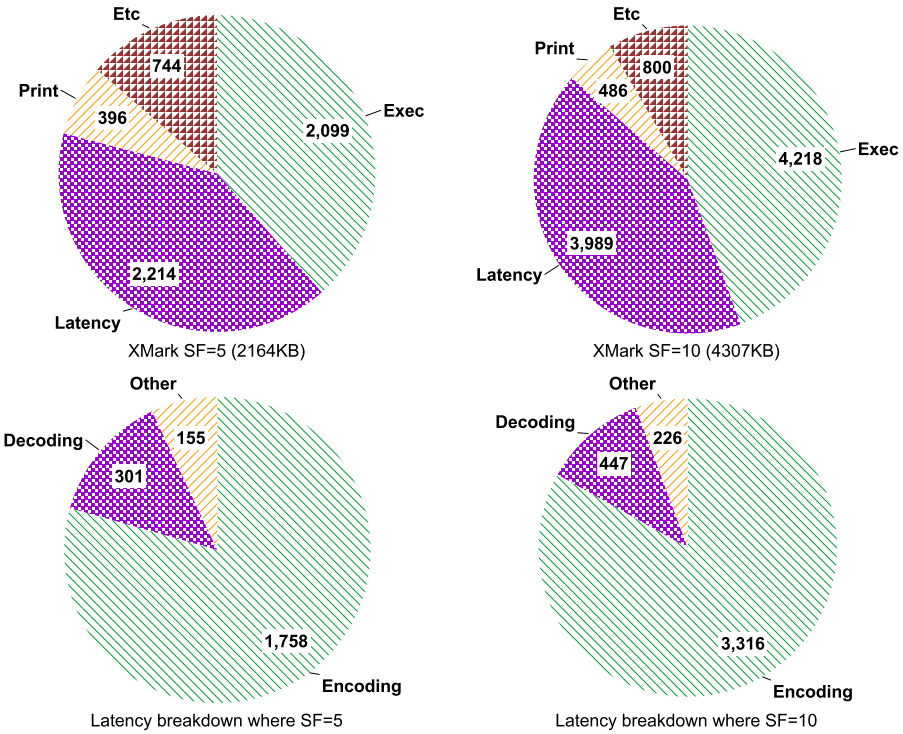
5.3.1 The Language Extension: BDQ

XBird/D extends the XQuery language [W3Cd] to support remote query execution. We call the extension for *XBird Distributed Query*, *BDQ* for short. Figure 5.4 describes the grammatical extension to *PrimaryExpr* of XQuery. *BDQExpr* means that $Expr_2$ is to be executed at a remote peer $\mathcal{P}$, where the endpoint of $\mathcal{P}$ is *fn:string*(*fn:exactly-one*($Expr_1$)). The endpoint takes a URL format of the form:
//*host*:*port*/*name* where *name* is the binding name of remote service that is bounded at the service registry identified with the pair of *host* and *port*.

5.3.2 Remote Proxy

As mentioned in Section **5.2**, our distributed XQuery processor *XBird/D* employs *remote proxy* to achieve an inter-operator parallelism.

The base (single) XQuery processor, which is noted as *XBird*, is based on the *Volcano* iterator model and employs an iterative query processing model based on an iterator tree, which is similar to the BEA/XQRL XQuery processor [FHK+04] in which pipelined processing operators deal with loops, axis accesses, etc. The iterator model



Exec: Elapsed time for remote query execution at a remote peer.

Latency: Latency of remote query execution including encoding/decoding and network latency.

Print: Elapsed time of serializing an XDM instance to a file.

ETC: Elapsed time including compilation and other (local) query execution footprints.

Figure 5.3. Breakdown of remote query processing time (in msec).

allows lazy evaluation of expressions, and also plays an important role in *XBird/D* architecture.

We describe how the *pass-by-reference* evaluation semantics is achieved by using *remote proxy*. *Remote proxy* is not a unique feature for *XBird/D*. It is an extension of the well-known proxy design pattern for distributed object communications [Roh95], and was not developed in the context of distributed XML query processing. It is impossible to accomplish inter-operator parallelism only with *remote proxy*. Therefore, we try to use a combination of *remote proxy* and an asynchronous entity production of intermediate results, as is described next.

$$BDQExpr ::= \text{“}execute\ at\text{”}\ Expr_1\ \text{“\{”}\ Expr_2\ \text{“\}”}$$
$$PrimaryExpr ::= Literal \mid .. \mid Constructor \mid BDQExpr$$

Figure 5.4. BDQ grammar extensions.

5.3.2.1 Asynchronous Production and Queue Management

We assume that items in a sequence are stored in FIFO queue in which each item is consumed by another operator processed by a consumer.

Figure 5.5 gives an overview of our *remote proxy* implementation. When executing a remote query, the intermediate result (*result entity sequence*) is wrapped with a proxy and the proxy object (*result proxy sequence*) is returned to the caller ($Peer_1$). Then, the remote operator asynchronously produces the items of the *result entity sequence* until the queue becomes full. When the queue becomes full, the producer thread is blocked until space becomes available in the queue$^{(a)}$. On the other hand, retrieving items is blocked until the queue becomes non-empty$^{(c)}$. The *remote proxy* fetches remote entries$^{(c)}$ when the local queue is empty at timing (*b*). The size of fetched items can be configured for each query by specifying initial fetch size and its growth factor. The fetched size automatically grows according to the parameters up to a specified threshold. This feature aims at reducing client/server communications. Due to the advantages of this simple but effective queue management, relying on *remote blocking-queue*, our system can utilize system resources by avoiding oversupply and undersupply.

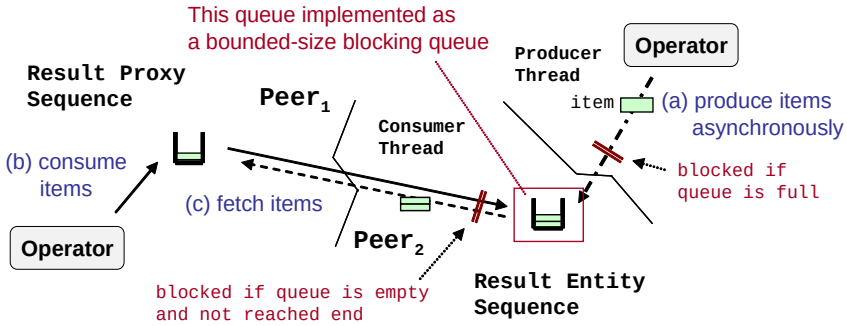


Figure 5.5. Server/Client interaction between processor elements using a remote proxy.

5.3.2.2 Direct Result Forwarding

As we have already mentioned in Section **5.2**, *XBird/D* has a *direct result forwarding* feature to reduce latency, e.g., encoding/decoding and network latency, in the distributed query execution. We explain how a query is processed in distributed and nested query execution in detail using Figure 5.6. In Figure 5.6, *filter*, *reduce*, *select1* and *select2* functions are executed at PE1, PE2, PE3 and PE4, respectively. The left half of Figure 5.7 expresses the nested operation tree. The *reduce* function collects closed_auction and open_auction and returns the first 1000 items for each. Since this *reduce* function does not cause local resource access (*fn:doc* and/or *fn:collection*), the execution is relocatable. This optimization is performed not at the compilation time but at execution time [2]. Since our XQuery processor is implemented based on the *Volcano* iterator model, the result iterators are calculated by lazy evaluation in call-by-name fashion. We do not implicitly attempt *memoization* techniques because intermediate results in XML processing are relatively larger than those of programming languages. For the relocation of execution, we just forward the iterators P1 and P2 to the upper operator (on PE1, in this case), and evaluate them on PE1. The intermediate results are fetched from PE3 and PE4 directly. This optimization is effective since encoding and decoding on PE2 can be avoided. Recall that it takes the majority of total elapsed time in remote query execution for encoding and decoding (see Figure 5.3).

A previously proposed system [FJM+07b] can use an *intensional* expression that was originally proposed in [MAA+05]. The intensional answer can make a server shift its work to a client by mutating the result expression with the intermediate results. For example, the intensional expression transfers the computation of the *reduce* function by mutating the result expression as follows:

```
fn:sequence( (<closed_auction> ... </closed_auction>, <closed_auction> ... </closed_auction>), 1, 1000),
          | (<open_auction> ... </open_auction>, <open_auction> ... </open_auction>), 1, 1000) )
```

However, according to our experience, the benefit of an intensional answer is limited to the case that the size of the result is small, since it requires additional (and expensive) encoding and decoding of the intensional results. The encoding and decoding often waste more server resources than evaluation of a query. In contrast, our

[2] The advantage of this dynamic relocation is that it can take execution time information into consideration.

iterator forwarding can receive the benefits of lazy evaluation without such a significant drawback.

```
declare function bdq:select1() {
  execute at $PE3 {
    fn:collection($col)/site/closed_auctions/closed_auction
  }
};
declare function bdq:select2() {
  execute at $PE4 {
    fn:collection($col)/site/open_auctions/open_auction
  }
};
declare function bdq:reduce() {
  execute at $PE2 {
    (  fn:subsequence(bdq:select1(), 1, 1000)
     | fn:subsequence(bdq:select2(), 1, 1000) )
  }
};
declare function local:filter() {
  for $a in bdq:reduce()
  where $a/seller/@person >= "person10000"
     or $a/buyer/@person >= "person10000"
  return $a
};
local:filter()  (: execute at PE1 :)
```

Figure 5.6. Nested remote query execution example.

5.4 Experimental Evaluation

In order to evaluate the effectiveness of our enhancements, i.e., *remote proxy* and *direct result forwarding*, we conducted performance comparisons to MonetDB/XRPC [ZB07b] version 4.18.1, which is one of the state-of-the-art distributed XQuery processors.

As the experimental environment, we used four PCs. We denote the XQuery processor running on each PC as PE1 ... PE4. Each PC consists of Pentium D 2.8GHz CPU, 2GB of memory, SuSE Linux 10.2 as an OS, and JDK 1.6 as a runtime system,

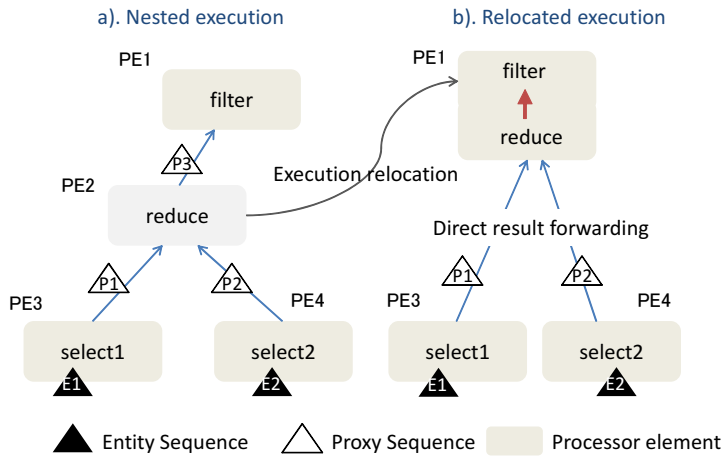


Figure 5.7. Execution relocation and direct result forwarding.

connected on 1Gb/s Ethernet, except for PE2, which is equipped with an Athlon 64 X2 2.4GHz CPU. We used a query in Figure 5.6 for the evaluation of *XBird/D* and the equivalent query for *MonetDB/XRPC*. As for the data set, we used a 1.1GB XML document generated by the data-generator of XMark [SWK+01] where the scale factor was set to 10. We bound the generated document to variable $col in Figure 5.6. The documents were loaded to each database instance in advance on PE3 and PE4 where both *MonetDB* and *XBird* were running.

The summarized results are shown in Figure 5.8. We compared four evaluation strategies including *remote proxy* (Proxy), *remote proxy* with *direct result forwarding* (Proxy+Forward), our implementation of *pass-by-value* semantics (Value), and *MonetDB/XRPC* by *pass-by-value* semantics [ZB07b] (XRPC).

As in Figure 5.8, our *pass-by-reference* implementation using *remote proxy* shows a significant improvement on the elapsed time. This is because our system could eliminate unnecessary computation at PE3 and PE4, as a result of applying lazy evaluation to distributed XQuery processing. The remote query evaluation by *pass-by-value* semantics computed and produced the entire results at one time, while not all of them are used in the later computation. It is clearly inefficient, and explains why our *remote proxy* evaluation strategy (Proxy) executed about 9 times faster than our *pass-by-value* evaluation strategy. Moreover, *direct result forwarding* eliminated the redundant en-

coding/decoding on PE2 and the overhead of mediated communications. As a result, our system could finally obtain about 22 times better performance than the competitive method (XRPC).

In addition, only our system using *remote proxy* can process 100 concurrent query requests using 30 threads in 160 seconds where the maximum and average elapsed times are 53.76 and 36.75 seconds, respectively. Both of the *pass-by-value* semantics implementations (Value and XRPC) suffer from a frequent swap-in/out [3] due to poor resource utilization. We, thus, confirm the advantage of our methods in a certain situation.

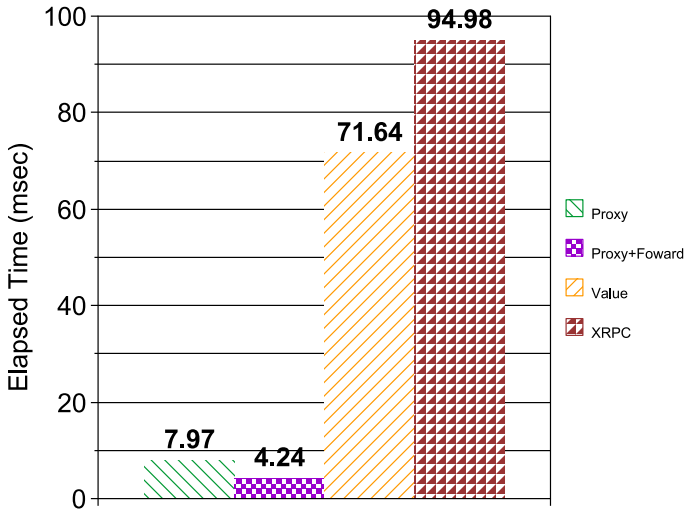


Figure 5.8. Comparison of four evaluation strategies.

5.5 Summary

In this chapter, we proposed an efficient distributed XML query execution strategy using *remote proxy*. Our experimental results show up to 22x speedups compared with competitive methods, and demonstrated the importance for distributed XML database

[3] XRPC did not return the first response of 5 concurrent queries in 10 minutes.

systems to take *pass-by-reference* semantics into consideration. Furthermore, our enhancements (asynchronous production managed by *remote blocking-queue*) can utilize system resources efficiently with supporting inter-operator parallelisms.

Issues to be explored include dynamic execution dispatching of remote query processors taking system resources and utilizations (e.g., CPU utilizations, free memory space, and specs) of the participating nodes into account, and development of a selection model of execution strategies.

6 Scalable Database System on Shared-memory Multiprocessors

Though our approaches in previous chapters are promising in the shown conditions, we noticed the possibility of improvement in many-core settings. A many-core processor, UltraSPARC T2 [Sun], revealed that when processing concurrent XML queries as examined in Section **5.4**.

In this chapter, we propose a lock-free variant of the GCLOCK page replacement algorithm, named Nb-GCLOCK. Concurrent access to the buffer management module is a major factor that prevents database scalability to processors. Therefore, we propose a non-blocking scheme for *bufferfix* operations that fix buffer frames for requested pages without locks by combining Nb-GCLOCK and a wait-free hash table. Our experimental results revealed that our scheme can obtain nearly linear scalability to processors up to 64 processors, while the existing locking-based schemes do not scale beyond 16 processors.

6.1 Background

Recent hardware trends toward multithreading for improved performance, including multi-core and multithreaded chip design, have raised critical challenges in software engineering [Sut05]. It has also presented issues to the database community both in research [CBHR06, CRG07] and open source database development. Open source DBMSs, such as PostgreSQL, MySQL and Apache Derby [Apab], have had to face scalability

problems with the increases in the number of processors. The open source DBMSs did not scale beyond four processors before revising their synchronization mechanisms in the buffer management modules.

In general, there are basically three approaches to coping with concurrency issues of synchronization:

(a) Do not acquire locks, and use a data structure that does not require locking [Val96]. The synchronization mechanism that avoids acquiring locks is called *non-blocking synchronization*.

(b) Reduce lock granularity. Fine lock granularity reduces lock contentions, although it may increase the overhead of locks themselves, i.e., the total time for acquiring and releasing locks.

(c) Use a more lightweight lock mechanism. Spinlock is efficient if threads are only likely to be blocked for a short period of time, as it avoids overhead from process re-scheduling or context switching in operating systems.

Both of the PostgreSQL and MySQL communities dealt with the scalability issues by making improvements using (b) and (c). However, several empirical studies have shown that they have scalability limits of around 16 processors [JPA08, Tol07, Inf].

Database systems now demand CPU scalability beyond 16 processors because the number of CPU cores per chip is doubling for each CPU manufacturing process in about two-year cycles. In addition, massively multithreaded processors, e.g., Sun's UltraSPARC T2 (64 processors) [Sun] and Azul System's Vega-2 7200 Series (768 processors) [Azu], have already been released as industrial products.

Most of the past research efforts on database buffer management have focused on improving their efficiency with respect to buffer hit rates on various workloads. Consequently, the literature contains very little research focusing on the concurrency of buffer management, and most of the difficulties remain to be handled by individual developers' empirical knowledge. In this chapter, we propose a scalable buffer management scheme that employs non-blocking synchronization instead of acquiring locks. To the best of our knowledge, this chapter is the first attempt to adopt non-blocking synchronization in buffer management.

One reason why concurrency in buffer management has not been discussed is that large-scale multiprocessors have not been widespread, and also the main concerns were

improving buffer hit rates and minimizing I/Os. However, the *bufferfix* operation that fixes a buffer frame for a required page [GR92] is not necessarily a CPU-bound job. Although disk I/Os in a *bufferfix* operation certainly take place when synchronization to disk is required for a replacement victim (i.e., the replaced page keeps its dirty flag on), modern DBMSs reduce such disk I/Os by *preflushing* dirty pages and preemptively selecting non-dirty pages for the replacement victim [GR92]. This means that the number of page replacements due to the *bufferfix* operation can be minimized if a large amount of memory is available and a large buffer pool can be used. In this case, the *bufferfix* operation becomes a CPU-bound task and, therefore, the CPU scalability issue in buffer management becomes particularly problematic in multiprocessor systems. Actually, *fix* and *unfix* operations to a buffer frame are the basic operations most frequently called in DBMSs. Thus, the efficiency of *fix* and *unfix* operations become extremely important because they lead to frequent contentions in the critical sections [BGMP79].

We need to clarify how much our proposed technique improves the performance for CPU-bound and I/O-bound jobs. One of the main factors in determining whether a job is CPU-bound or I/O-bound is the number of disk I/Os, which depends on the buffer hit rates as well as the ratio of dirty pages in replacement victims. Therefore, we focus on buffer hit rates and give considerations to the scalability to the number of processors when buffer hit rates change.

Several non-blocking algorithms for hash tables have already been proposed [SS06, PH05]. In this chapter, we focus on concurrency of page replacement algorithms and utilize an existing *wait-free* hash table for searching buffer frames. Then, we propose our Nb-GCLOCK page replacement algorithm, which is a non-blocking variant of the *generalized CLOCK* (GCLOCK) page replacement algorithm [Smi78]. We also verify the effectiveness of our non-blocking page replacement algorithm with respect to its concurrency and throughput using Sun UltraSPARC T2 [Sun]. The experimental results revealed that our scheme can obtain nearly linear scalability to processors up to 64 processors, while the existing locking-based schemes do not scale beyond 16 processors.

The rest of this chapter is organized as follows. Section **6.2** introduces the background of the need for non-blocking page replacement. We explain why existing buffer management schemes cause scalability bottlenecks to processors. In Section **6.3**, we describe the details of our non-blocking page replacement algorithm. In Section **6.4**, we evaluate our proposed scheme through experiments. We refer to related work in

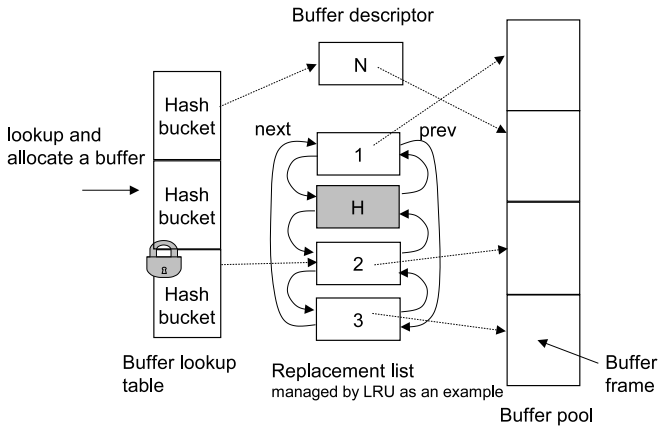


Figure 6.1. Typical organization of a buffer manager.

Section **6.5** and conclude the paper in Section **6.6**.

6.2 Problem Description

In this section, we explain the background of our research to address open and known problems in buffer management by giving examples. A buffer manager typically consists of a *buffer lookup table* for searching buffer frames, *buffer descriptors* to manage a page replacement policy, and a *buffer pool* as shown in Figure 6.1. The *buffer lookup table* is usually constructed as a hash table [EH84, GR92]. As for the page replacement policy, LRU, CLOCK, and their refinements [JS94, JCZ05] are widely used. We refer to the module that manages a page replacement policy the *replacement list*.

6.2.1 Internal Locking in Buffer Manager

Access to a shared buffer cache has a significant scalability problem, particularly on multiprocessor systems. When accessing a buffer management module, the operations to the critical sections must acquire their mutual exclusions. For example in Figure 6.1, to look up a buffer in the buffer pool, a *shared lock* is obtained in the *buffer lookup table*. To alter the page assignment of a buffer, an *exclusive lock* is acquired on the buffer manager. This lock must be held while adjusting the *replacement list* and changing

the *buffer lookup table*. This is because the reference in *buffer lookup table* still has a different page identifier immediately after changing the page allocation of a buffer frame.

To avoid the lock during the operations on the *buffer lookup table* and the *replacement list* and to reduce the lock granularity, the following strategies can be taken:

- Store a page identifier for each buffer frame, compare the page identifier of the referred buffer frame to validate whether the page is evicted, and read a page from disk if the existing page on the frame is evicted. Then, it is necessary to pay an extra CPU cost to guarantee the consistency when a cache miss occurs.

- Alternatively, delay the changes to the replacement list until the corresponding *bufferunfix* operation is implemented and replace the exclusive lock for the replacement list with a shared lock in the *bufferfix* operation. In this case, an exclusive lock on the *replacement* list in bounce mode, in which a lock is immediately denied if not available, is granted after acquiring an exclusive lock on the *buffer lookup table*. Even though the timing to select a replacement victim is delayed to a *bufferunfix* operation, empirical studies have shown that buffer hit rates do not change [GR92].

Suppose then that concurrent requests from multiple users are given. If one paging request causes a page fault and holds an exclusive lock, the exclusive lock prevents the others from holding either a shared or exclusive lock. Since system-wide *mutexes* tend to appear for each scan of pages, it would cause "*mutex* ping-pong" in multiprocessor and multithreaded environments. Moreover, high traffic access to a lock may causes the *convoy phenomenon* [BGMP79].

PostgreSQL (version 8.2), MySQL (version 5.0.30) and Apache Derby coped with the lock contention problems in their buffer pools by adopting finer-grained locking schemes. They took a conventional and conservative approach [GR92] for refining the concurrency of a hash table, called *lock-striping*, which is a technique that divides a giant lock into clusters so as to reduce contentions. On the other hand, we attempt a more aggressive approach to synchronization toward massively multithreaded environments, rather than using the conservative one.

6.2.2 Revising Concurrency in Page Replacement Algorithms

We address here concurrency issues of page replacement algorithms by taking three practical examples, *Least Recently Used* (LRU), 2Q [JS94], and *Generalized CLOCK* (GCLOCK) [Smi78], for the following discussion.

LRU is typically arranged as a double-linked list to keep the LRU chains as shown in Figure 6.1: adding new items to the head, removing items from the tail, and moving any existing items to the head when referenced (touched). When using LRU, the *replacement list* always needs to be locked for each access to the linked list. Thus, the LRU algorithm is effective for single-threaded applications but becomes very slow in a multithreaded environment. CLOCK [Cor69], which has an approximately equivalent performance to LRU, is often used as a substitution [RG02]. CLOCK does not require a giant lock when an entry is touched. It needs only one atomic operation, e.g., setting a reference bit on or incrementing a reference counter, on the touched entry.

GCLOCK is an efficient variation of CLOCK and uses a reference counter instead of the use-bit of a buffer page, as used in the conventional CLOCK. An example of GCLOCK organization is shown in Figure 6.2. In GCLOCK, the references to a page P_i increment the corresponding counter $RC(i)$. In the basic GCLOCK, $RC(i)$ is initialized to 1 upon the first fetch of P_i and incremented by one every time P_i is touched. When a buffer fault occurs, a circular search is initiated, decrementing stepwise the reference counters until the first entry with a value of 0 is found.

Furthermore, LRU is known to be inefficient for sequential scans and large *Inter-Reference Gaps* (IRGs) [RG02]. A burst of references to infrequently used pages, such as sequential scans, may cause replacement of commonly referenced pages in the cache. In [JS94], the authors present the scan-resistant 2Q algorithm, which divides cache items into hot and cold ones. The full version of the 2Q algorithm uses three FIFOs for managing items. The buffer manager in Oracle universal server employed a variation of LRU that uses two separated hot/cold queues for the LRU chain management [BJK+97]. 2Q and similar algorithms as well as plain LRU management have a global contention point on their replacement list, which degrades the scalability of buffer managers on multiprocessor systems.

To cope with sequential scans expected by database workloads, PostgreSQL 8.0 and our native XML database system [YMUK07] moved from LRU to the 2Q algorithm. However, as did the PostgreSQL community, we finally realized that 2Q has an un-

avoidable synchronization penalty on multithreaded systems. Therefore, PostgreSQL has shifted to CLOCK mostly due to the contention penalty. Similarly, we shifted to a GCLOCK refinement that employs a novel non-blocking scheme instead of a lock-based one. These facts imply that minimizing paging I/Os is not only a unique requirement on current hardware but also that CPU-bound operations affect the overall performance.

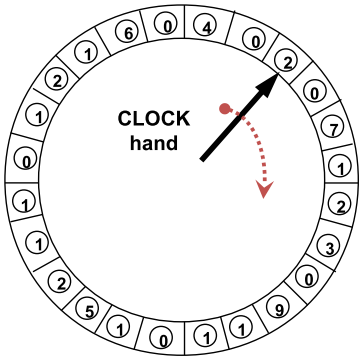


Figure 6.2. An example of GCLOCK organization.

6.2.3 Spinlock on SMT Environment

Conventional multiprocessor systems widely use spinlocks to guard *critical sections*. Spinlock is effective if threads are only likely to be blocked for a short period of time, since the cost of acquiring and releasing a lock is smaller than a sleep lock, though a *spin-wait loop* consumes one processor resource.

There are several variations of the spinlock. Past studies have shown that the Test-and-Test-and-Set (TTAS) lock with exponential backoff (or queue lock) is one of the most promising spinlock protocols [And90]. Both MySQL and PostgreSQL use *spin-wait loops* with backoff as their spinlock algorithms for most hardware architectures.

However, a spin loop can be especially wasteful where logical processors share execution resources. When such loops are executed on a processor supporting Intel *Hyper-Threading* technology, they can induce an additional performance penalty due to memory-order violations and consequent pipeline flushes caused upon their exit. To ensure the proper order of outstanding memory operations, the processor incurs

a severe penalty. In order to overcome this issue, Intel recommended embedding a PAUSE instruction in a spin loop [Int01]. PAUSE instruction introduces a slight delay in the loop and de-pipelines its execution to prevent it from aggressively consuming valuable processor resources.

In summary, a spinlock requires a special care (i.e., special instructions) on each hardware architecture when database operations are executed on SMT processors. Again, our non-blocking buffer management scheme does not acquire any locks for searching and allocating buffer pages, and thus it is free from such difficulties in spinlocks.

6.3 Non-blocking GCLOCK Page Replacement Algorithm

Here, we explain our non-blocking buffer management scheme and our lock-free variant of the GCLOCK page replacement algorithm, named *Nb-GCLOCK*. Our Nb-GCLOCK algorithm basically follows the properties of GCLOCK [Smi78] except that it allows non-blocking accesses.

The reasons why we selected GCLOCK as the baseline algorithm of our non-blocking page replacement are as follows:

1. CLOCK variants are widely used due to their advantages, i.e., low overhead and high concurrency.

2. The properties and performance of GCLOCK are well analyzed and established [Smi78, NDD92]. While the simple CLOCK respects only the *recency* of buffer references, GCLOCK takes *frequency* as well as *recency* into account.

3. The probability of contentions generated by concurrent accesses to shared variables is low. The contention indicates such a state that two or more processes concurrently access the same memory location. A typical CLOCK uses a single bitmap or few bitmaps to manage reference frequency. When a bitmap is frequently updated, CLOCK becomes inefficient on cache-coherent shared memory multiprocessors due to *false sharing*. On the other hand, GCLOCK keeps a reference counter for each buffer frame, and thus contentions rarely occur.

Our Nb-GCLOCK adopts a lock-free page replacement. Non-blocking data structures have two important properties [HS08]. If some operations are guaranteed to complete within finite time, the algorithm is defined as *lock-free*. A *lock-free* algorithm guarantees that at least one process keeps its role progressing. If all operations are guaranteed to complete within a finite time, the algorithm is defined as *wait-free*. The first and second definitions guarantee the *liveness* and *fairness* properties, respectively. From this viewpoint, our proposed buffer management scheme guarantees *lock-free* operation.

Our scheme takes a strategy that keeps trying its non-blocking operation after temporarily abandoning its execution and allows other threads to be executed when all buffer frames in the buffer pool are pinned. This decision comes from our expectation of adopting this scheme for general caching. Due to this decision, it is impossible to guarantee that all processing will complete in a finite time when we consider the case where all pages in the buffer pool are pinned, although this is an extremely rare case. The behavior depends on whether applications that uses the cache allow failures at the buffer allocation. In a typical buffer management, a transaction is aborted when all pages are pinned [RG02, GR92]. This abnormal condition had never occurred through our experiments.

We implemented our proposed scheme using atomic operations provided in Java 5.0 [AGH05]. Because the Java platform has a platform-independent memory model (*JMM*) [MPA05] as well as integrated threading and multiprocessing into the language, it is convenient for describing algorithms for multithreaded applications.

As mentioned in Section **6.2**, we use a wait-free hash table for a *buffer lookup table* to achieve non-blocking synchronization on the buffer management. The non-blocking hash table has been actively developed in the literature [SS06, PH05]. We used an open source implementation of a non-blocking hash table released in [Cli] for the wait-free hash table. The non-blocking hash table insists on *wait-freedom* — every operation has a bound on the number of steps it will take before completing. Our requirements for the wait-free hash table are to provide the following operations: *get*, *pushIfAbsent*, *remove*, and *replace*.

- *get* returns the value to which the specified key is mapped in the hash table.
- *pushIfAbsent* associates the specified key with a specified value in the hash table when the specified key is not already associated with a value. It returns the

previous value associated with the specified key, or null if there has been no mapping for the key.

- *remove* removes an entry for a key only if it is currently mapped to the given value. It returns true if the value has been removed.
- *replace* replaces an entry for a key only if the given key is currently mapped to the given value. It returns true if the value has been replaced.

These actions must be performed atomically. The implementation of the wait-free hash table can be replaced if the alternative provides the above operations.

6.3.1 Nb-GCLOCK Algorithm

We describe our Nb-GCLOCK algorithm in Figure 6.7 and Figure 6.8 with a usecase shown in Figure 6.9, using pseudo codes based on Java 5.0 language.

All operations to AtomicInteger and AtomicBoolean are atomically executed by using synchronization primitives such as *compare-and-swap* (CAS) and *Load-Link/Store-Conditional* (LL/SC). In SPARC V9, such an atomic operation is achieved by a native CAS instruction.

6.3.1.1 Organization of the Buffer Frame

The left half of Figure 6.7 describes the *Frame* class that defines cached entries. A *Frame* instance is associated with a single key, a single value, and two other control variables. In buffer management, K represents a page identifier and V represents a page itself. A *refcount* instance keeps a reference count of the entry, and a *pinning* instance is responsible for judging whether the frame is currently in use. Playing a vital role, a *pinning* instance represents an *evicted* condition when the value is -1.

The reason why we represent *evicted* and *pinned* states with a single *pinning* instance as shown in Figure 6.3 is to achieve an atomic update on these states using a synchronization primitive without acquiring a lock. We introduce the state for the Frame instance whose pinning value is -1 *evicted* (see Figure 6.3).

The roles of methods in the *Frame* class are as shown in Table 6.1. The *pin* and *unpin* operations follow FIX and UNFIX operations of the FIX-USE-UNFIX protocol, which is generally used in buffer management [GR92].

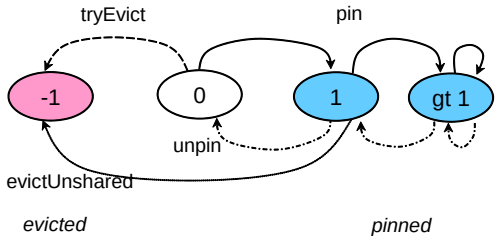


Figure 6.3. State machine of a pinning instance.

volatileGetValue	returns the associated value after interleaving memory barrier for volatile load. The memory barrier is reduced to no-op in x86 or SPARC [Lea].
CASValue	atomically sets the field value *V* to the given updated value if the current value is identical to expected value.
incrRC	atomically increments the reference count by one and returns the updated value.
decrRC	atomically decrements the reference count by one and returns the updated value.
tryEvict	atomically sets the frame evicted if the frame is not evicted, and returns true if successfully evicted and otherwise returns false.
evictUnshared	atomically sets the frame evicted.
pinCount	returns the pinning value.
pin	atomically sets the frame in use for the current thread.
unpin	atomically sets the frame not in use for the current thread.

Table 6.1. Role of each method in the Frame class.

The *pin*/*unpin* and *tryEvict*/*evictUnshared* methods atomically change a pinning value. The state of the pinning value changes by these four methods as shown in Figure 6.3. The pinning value *P* increases only when *P* is greater than or equal to 0. "gt 1" in Figure 6.3 represents the state where the pinning value is greater than 1. As seen in the transition, the pinning state does not change to the other states when once evicted. The pinning value is always one or more when an *unpin* operation is carried out. *tryEvict* succeeds only when the pinning value is 0.

6.3.1.2 Bufferfix Algorithm

The operation to fix a buffer frame for a requested page is called *bufferfix* in the literature. According to the definition in [GR92], a *bufferfix* operation acts as depicted in

1. Search a buffer frame in the buffer.
   Examine whether the requested page is in the buffer and return the reference to the corresponding frame to the caller when such a frame exists.

2. Find an empty frame.
   Find an empty frame in which a page is not fixed when the requested page is not in the buffer.

3. Select a replacement victim.
   Select a page to remove from the buffer according to the replacement policy.

4. Write a dirty page to disk.
   Write the change log to a WAL page if the victim has a dirty flag on.

5. Decide a frame to fix a page.
   In this time, the frame to fix a page is decided by step 2 to 4.

6. Pin a page to the frame.
   Read-in the requested page and fix in the selected frame. This is an optional operation depending on the requirements of the caller.

7. Return to the caller.
   Return the reference to the frame in which the requested page is fixed or to be fixed.

Figure 6.4. Definition of a bufferfix operation.

Figure 6.4.

The corresponding procedure to the bufferfix operation in our Nb-GCLOCK is the *fixEntry* method, except that a caller is responsible for the page fixing operation as in line 6 of Figure 6.9.

The right half of Figure 6.7 describes the *BufferCache* class and its algorithm. *BufferCache* contains a wait-free hash table instance HASHTBL and a replacement list CLOCKBUF as its member variables. For readability, the algorithm of CLOCKBUF is separately described in Figure 6.8. The *BufferCache* contains two methods: *fixEntry* for retrieving a page slot (i.e., buffer frame) and *addEntry* for allocating a page slot for the given key.

The *addEntry* method is called when the condition in line 49 of Figure 6.7 becomes false, when a non-evicted page associated with the specified page identifier does not exist in HASHTBL.

Buffer flushing is required when an evicted page has a dirty flag on in the purge operation of *addEntry*. This I/O in the purge operation is minimized in modern DBMSs as mentioned in Section **4.1**.

The invocation of *fixEntry* fixes a frame for the specified key and increments the reference count of the fixed frame by one.

THEOREM 1. *Every time an existing Frame instance F is returned by the* fixEntry *method, the* pinning *value of F is incremented by one.*

Proof of Theorem1. An existing Frame instance F is returned by the fixEntry invocation only when the condition in line 49 or 66 of Figure 6.7 becomes true or false, respectively. Whenever these conditions are met, it is clear that the pinning value of F was incremented by one according to the *pin* specification. □

COROLLARY 1. *From Theorem1, a Frame instance successfully evicted by* evictUnshared *is never used outside the Frame class.*

6.3.1.3 CLOCK-sweep Algorithm

Selecting and swapping a replacement victim in the buffer pool of CLOCK is called the *clock-sweep* operation.

Figure 6.8 describes the *ClockBuffer* class which manages the Nb-GCLOCK page replacement policy. It contains four member variables: an atomic array "POOL" as the buffer pool, an atomic "Free" counter responsible for managing the number of free-slots in the buffer pool, an atomic "CLOCKHAND" representing a circulating clock hand, and a "SIZE" field representing the capacity of the buffer pool.

AtomicArray class provides support for atomic operations to an array. A method invocation *CAS(index, expect, update)* on AtomicArray atomically sets the existing value of a specified index to an updated value if the current value is identical to the expected one.

The ClockBuffer class has a single entry point on the *add* method. The *add* method fixes the given frame to the buffer pool. The *swap* method is invoked when the *add* method replaces an existing frame with the new frame according to GCLOCK page replacement policy. The *moveClockHand* method moves the clock hand in a style of atomic add instructions. We used this "add" scheme because "set" instructions to a clock hand are not robust for multithreaded accesses. Therefore, we add a "delta" (i.e., an absolute difference).

We now provide theorems to give consistency to the *add* algorithm.

THEOREM 2. *A given entry is always fixed to a free space in the buffer pool whenever decrementing the FREE instance succeeds in line 15.*

Proof of Theorem2. It is clear that at least one free space is ensured at the instant when decrementing the FREE instance succeeds in line 15. However, another thread may seize free space upon entering *swap* in line 14 immediately after the success. To cope with this case, the *swap* method avoids using any free space and gives free space to other threads in line 28. Thus, the *add* method fixes the given *entry* to free space in the buffer pool in line 19 whenever decrementing the FREE instance succeeds. □

THEOREM 3. *On* add *method call, the same Frame instance will never be returned to a different invocation.*

Proof of Theorem3. When *add* method call returns a non-null value, the *swap* method is called to return an evicted Frame instance. The evicted instance will never be managed in the *replacement list* (see line 54 to 58 of Figure 6.7).

Between lines 27 and 47 in the for-loop of *swap* method, the state of a Frame instance *e* changes as shown in Figure 6.5. When *e* is returned through the state 32/44, a *compare-and-swap* operation removes *e* from the buffer pool at the transition (c). Therefore, the same Frame instance will never be returned to a different invocation. □

6.3.2 Correctness Proof

We prove that our Nb-GCLOCK algorithm is linearizable [HW90]. According to [HW87], the definition of linearizability is equivalent to the following:

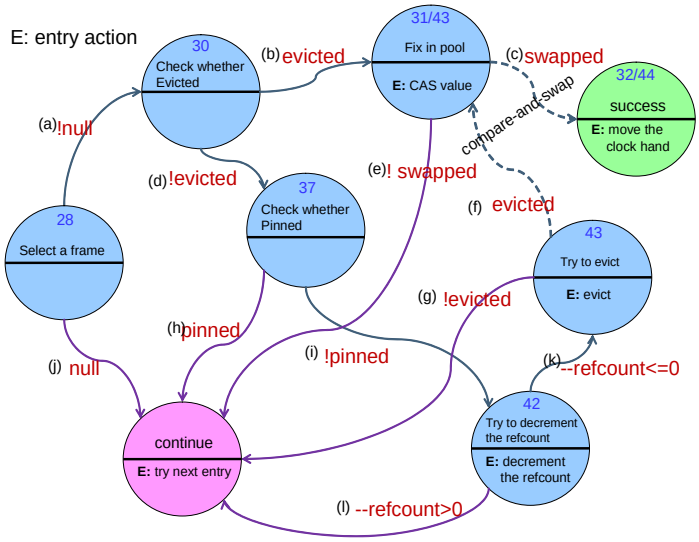


Figure 6.5. State transitions in clock-sweep.

- All function calls have a linearization point at some instant between their invocation and response.
- All functions appear to occur instantly at their linearization points, behaving as specified by the sequential definition.

Every execution path of *fixEntry* has at least one linearization point. We chose the following linearizable points for each execution path returning through lines 51, 69, 76, and 78, respectively:

- The *pin* operation for the returned Frame instance at line 49,
- The *replace* operation at line 67,
- The *pin* operation for the returned Frame instance at line 66, and
- The *putIfAbsent* operation at line 64.

LEMMA 1. *If entry is null or already evicted in line 49, then* pin *fails; otherwise,* pin *succeeds and the execution steps into line 50.*

```
1  AtomicInteger cnt[];
2  int get() {
3   int sum = 0;
4   for(AtomicInteger i: cnt)
5    sum += i.get();
6   return sum;
7  }
8  void add(int x) {
9   int idx = hash value of current thread
10  cnt[idx].add(x);
11 }
12 void increment() {
13  add(1);
14 }
```

Figure 6.6. Internal design of AtomicCounter class.

LEMMA 2. *If prevEntry does not exist in HASHTBL at line 67, then* replace *fails; otherwise,* replace *succeeds and the execution steps into line 68.*

LEMMA 3. *If prevEntry is not evicted in line 66, then* pin *succeeds; otherwise,* pin *fails and the execution steps into line 74.*

LEMMA 4. *If and only if an entry associated with the key does not exist in line 64,* putIfAbsent *returns null and the execution steps into line 78.*

Lemma1, Lemma2, Lemma3, and Lemma4 derive the following theorem:

THEOREM 4. *The* fixEntry *algorithm in Figure 6.7 is linearizable.*

6.4 Experimental Evaluation

In order to evaluate the effectiveness of Nb-GCLOCK, we compared Nb-GCLOCK with LRU, GCLOCK [Smi78], and the full version of 2Q [JS94]. As for the parameters for 2Q, we used 20% and 30% of the buffer spaces for *K1in* and *K1out*, respectively. We used TTAS lock with exponential backoff for the synchronization of blocking algorithms.

```
class Frame {
 1 K key; V value;
 2 AtomicInteger refcount=new AtomicInteger(1);
 3 AtomicInteger pinning=new AtomicInteger(1);
 4 Frame(K key, V value) {
 5  this.key = key;
 6  this.value = value;
 7 }
 8 V volatileGetValue() {
 9  memory fence for volatile load
10  return value;
11 }
12 boolean CASValue(V expect,V update) {
13  return CAS(value, expect, update);
14 }
15 void incrRC() {
16  refcount.increment();
17 }
18 boolean decrRC() {
19  return refcount.decrement();
20 }
21 boolean tryEvict() {
22  return pinning.CAS(0, -1);
23 }
24 void evictUnshared() {
25  pinning.CAS(1,-1);
26 }
27 int pinCount() {
28  return pinning.get();
29 }
30 boolean pin() {
31  int current;
32  do {
33    current = trg.get();
34    if(current < 0)
35      return false;
36  return true;
37 }
38 void unpin() {
39  pinning.decrement();
40 }
 } //end Frame
```

```
class BufferCache {
41 HashTable HASHTBL;
42 ClockBuffer CLOCKBUF;
43 BufferCache(int size) {
44  HASHTBL = new HashTable(size);
45  CLOCKBUF = new ClockBuffer(size);
46 }
47 Frame fixEntry(K key) {
48  Frame entry = HASHTBL.get(key);
49  if(entry != null && entry.pin()) {
50   entry.incrRC();
51   return entry;
52  } else {
53   return addEntry(key, null);
54  }
55 }
56 Frame addEntry(K key, V value) {
57  for(;;) {
58   Frame newEntry = new Frame(key, value);
59   Frame removed = CLOCKBUF.add(newEntry);
60   if(removed != null) {
61    if(HASHTBL.remove(removed.key, removed))
62     purge the removed page
63   }
64   Frame prevEntry =
      HASHTBL.putIfAbsent(key, newEntry);
65   if(prevEntry != null) {
66    if(!prevEntry.pin()) {
67     if(HASHTBL.replace(key,prevEntry,newEntry)) {
68      newEntry.setValue(prevEntry.getValue());
69      return newEntry;
70     }
71     newEntry.evictUnshared();
72     continue; //jump to line 58
73    }
74    newEntry.evictUnshared();
75    prevEntry.incrRC();
76    return prevEntry;
77   }
78   return newEntry;
79  }
80 }
} //end BufferCache
```

Figure 6.7. Pseudo code of the buffer cache.

```
class ClockBuffer {
 1 AtomicArray POOL;
 2 AtomicInteger FREE;
 3 AtomicCounter CLOCKHAND=new AtomicCounter(0);
 4 int SIZE;
 5 ClockBuffer(int size) {
 6  this.POOL = new AtomicArray(size);
 7  this.FREE = new AtomicInteger(size);
 8  this.SIZE = size;
 9 }
10 Frame add(Frame entry) {
11  do {
12   int free = FREE.get();
13   if(free == 0)
14     return swap(entry);
15   if(FREE.CAS(free, free - 1))
16     break;
17  } while(true);
18  int idx = CLOCKHAND.get();
19  while(!POOL.CAS(idx%SIZE, null, entry))
20    idx++;
21  CLOCKHAND.increment();
22  return null;
23 }
23 Frame swap(Frame entry) {
24  int numpinning = 0;
25  int start = CLOCKHAND.get();
26  for(int i=start%SIZE;;i=(i+1)%SIZE) {
27   Frame e = POOL.get(i);
28   if(e == null) continue;
29   int pincount = e.pinCount();
30   if(pincount == -1) { // evicted?
31     if(POOL.CAS(i,e,entry)) {
32       moveClockHand(i, start);
33       return e;
34     }
35     continue;
36   }
37   if(pincount > 0) { // pinned?
38    if(++numpinning>=size)
39      yield this thread and allow others to execute
40    continue;
41   }
42   if(e.decrRC() <= 0) {
43    if(e.tryEvict() && POOL.CAS(i,e,entry)) {
44      moveClockHand(i, start);
45      return e;
46    }
47   }
48  } //end for
49 } //end swap
50 void moveClockHand(int curr, int start) {
51  int delta;
52  if(curr < start)
53    delta = curr + size - start + 1;
54  else
55    delta = curr - start + 1;
56  CLOCKHAND.add(delta);
57 }
} //end ClockBuffer
```

Figure 6.8. Pseudo code of the ClockBuffer.

```
1  Frame slot =
     PAGE_CACHE.fixEntry(pageId);
2  try {
3   V page = slot.volatileGetValue();
4   if(page == null) {
5     page = read-in a page of the pageId from disk
6     slot.CASValue(page);
7   }
8   do application logic for the page
9  } finally {
10   slot.unpin();
11 }
```

Figure 6.9. Usage of a buffer in our scheme.

As for the workloads, we followed the example provided in [JS94] in which the authors tried a mixed workload containing both random accesses with *Zipfan* distributions and scans because database workloads generally contain scans.

The comparison is performed with respect to buffer hit rates and throughputs. Since our Nb-GCLOCK basically follows GCLOCK, the hit rate shows similar tendency to that of GCLOCK. We thus have put more evaluations on throughputs than hit rates in this chapter.

We performed experiments on a real hardware with a Sun UltraSPARC T2 processor (Sun SPARC Enterprise T5120 box). The detailed specification is shown in Table 6.2. The processor has eight CPU cores, and each core is able to handle eight threads concurrently. Thus, the processor is capable of processing up to 64 concurrent threads. We used a 64-bit version of Sun JDK 1.6 for the runtime environment in all of the experiments.

6.4.1 Experiments on Mixed/Zipfan Distributions

The experiments in this section used artificially generated workloads using a *Zipfan* input distribution [Knu98] with parameters $\alpha = 0.5$ and $\alpha = 0.86$. That is, if there are N pages, the probability of accessing a page numbered i or less is $(i/N)^{\alpha}$. A setting of $\alpha = 0.86$ gives an 80/20 distribution, while a setting of $\alpha = 0.5$ give a less skewed distribution (about 45/20). When running the Zipf simulator, we modified the workloads

Operating System	Solaris 10 8/07
Core (Threads/Core)	8 (8)
Processor frequency	1.2 GHz
Main memory	16 GB
Disk	SAS (10000 rpm)
L2 cache per core	4M

Table 6.2. Specifications of Sun SPARC Enterprise T5120.

so that it would occasionally start scans. We used mixed workload of Zipf with 20% scans of 100 pages, and the page size commonly used throughout our experiments is 8 KB. To emulate a multi-user scenario, the workloads are concurrently issued from multiple threads in which each thread uses its own simulator and workload.

We simulated a database consisting of 50,000 pages and the buffer capacity ranging from 4,096 (at least 8.2% is buffered when the pool is filled) through 16,384 page slots (at least 32.7% is buffered when the pool is filled). As for the buffer capacities used in experiments, we used from 8K to 32K.

6.4.1.1 Relation to Buffer Hit Rate

Figure 6.10 shows the relationship between buffer capacities and the buffer hit rates when 64 threads concurrently ran the above mentioned 80/20 workload on UltraSPARC T2. With decreases in the buffer capacity, 2Q shows better buffer hit rates. This result is natural because 2Q is effective for sequential scans and is expected to provide better buffer hit rates than LRU (and CLOCK) [JS94]. Of course, enough buffer capacity minimizes the differences as the result shows in Figure 6.10; however, highly concurrent accesses cause almost random access to buffers.

Figure 6.11 shows throughputs of the experiment varying the buffer capacity and *Zipf* distributions. These results imply that, at least in our experiments, throughputs depend almost completely on buffer hit rates, while the workloads weakly correlate with throughputs as seen at 75.4 and 75.5% of hit rates in Figure 6.11. We thus focus on buffer hit rates in the following experiments.

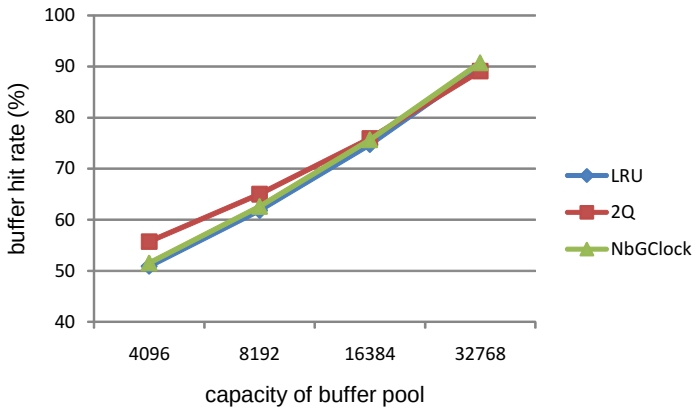


Figure 6.10. Relationship between buffer capacity and buffer hit rate (64 threads).

6.4.1.2 I/O in Progress and Concurrent I/Os

In certain scenarios, there are race conditions in which multiple threads attempt the same I/O operation on a fixed frame concurrently. Therefore, in conventional buffer management, a process waits until the *io_in_progress* lock on a fixed frame is released when someone else has already started I/O on the buffer [GR92]. Though our Nb-GCLOCK makes *bufferfix* operations non-blocking, the *page-in* operation to a fixed frame seems to remain an open problem.

To make this *page-in* operation non-blocking, our non-blocking scheme acts optimistically as shown in Figure 6.9. The existing scheme acquires a lock before reading a page and releases the lock after associating the page to a frame. Taking a lock before reading a page from disk may be reasonable, since *lseek* and *read* system calls also require mutex exclusion. On the other hand, our scheme does not delay the concurrent I/O by using *pread*. The *pread*/*pwrite* system calls enable efficient I/O to the same file descriptor from multiple threads. Then, it needs to be proven that

- to what extent contentions are expected on the critical section (i.e., lines 3 to 7 in Figure 6.9), and
- which strategy is effective for (massively) parallel workloads.

Figure 6.12 shows an experimental result on the 80/20 workload on the Ultra-

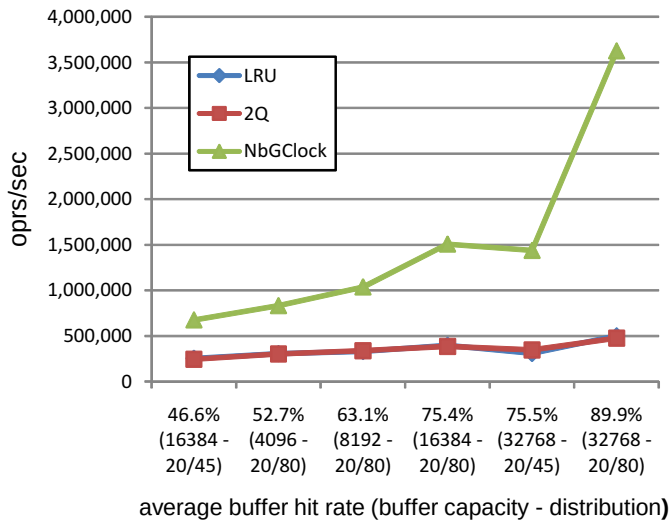


Figure 6.11. Throughputs obtained when varying buffer capacity and workload distributions.

SPARC T2 comparing the proposed scheme with the existing lock-based schemes. Each method is denoted as "*pread*" or "*lseek+read*" in Figure 6.12. To answer the first question, we counted CAS failures generated at line 6 of Figure 6.9, and the results are shown in Table 6.3. From Table 6.3 and Figure 6.12, our non-blocking scheme is effective even when contentions occur at a probability of 1/10, which lead us to expect that the proposed technique becomes more effective as the buffer capacity increases while a certain threshold exists.

Note that we used a single disk in the experiments, and thus this approach could be more effective on a high-throughput disk configuration such as RAID 0.

6.4.1.3 Scalability to Processors

We ran a series of experiments that varies the number of processors. In the experiments, we disabled/enabled processors by using the *psradm* command provided in Solaris.

The first experiment measured the scalability to processors where all pages are

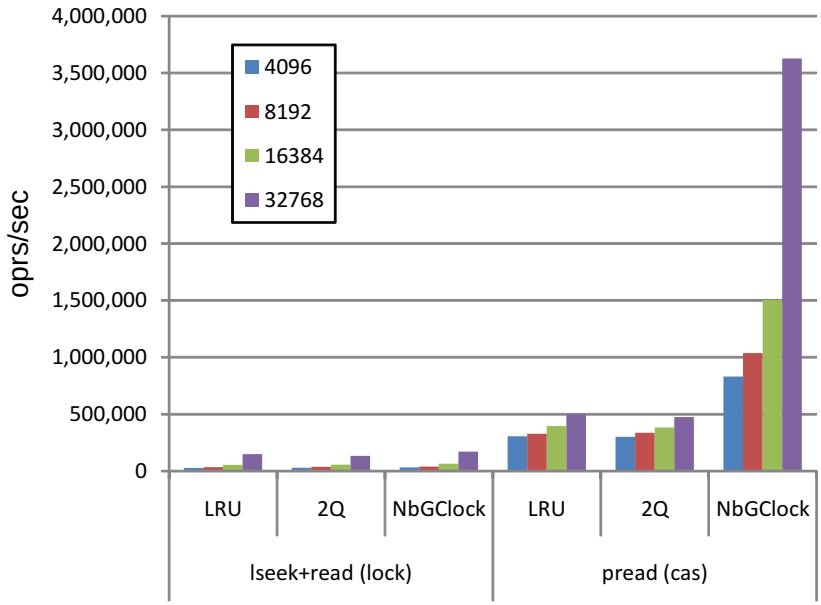


Figure 6.12. Comparison between "lseek+read" and "pread".

resident in memory. This experiment intended to see the scalability limit expected by each algorithm in light of adopting the non-blocking scheme to high I/O throughput configurations.

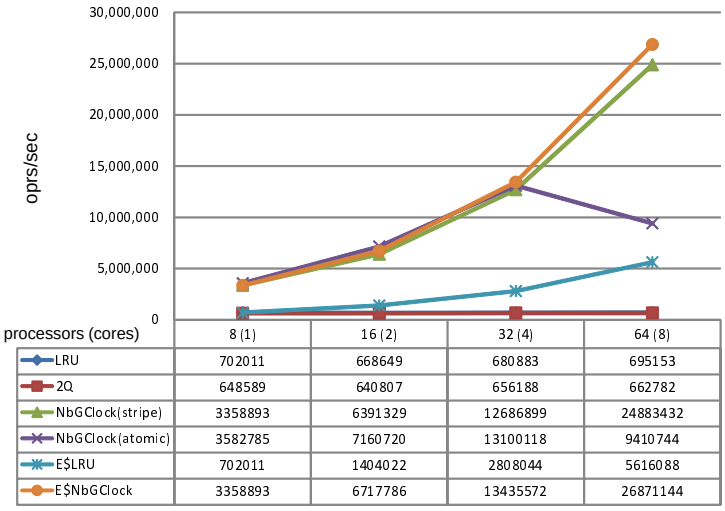
The experimental results in Figure 6.13 show that our non-blocking scheme denoted as "NbGClock(stripe)" is nearly scalable up to 64 processors. E$LRU and E$NbGClock show the expected scalability to processors according to the result on 8 processors for LRU and Nb-GCLOCK, respectively. The reasons why the scalability of "NbGClock(atomic)", which uses an AtomicInteger class for the CLOCKHAND in Figure 6.8, declined between 33 and 64 processors are as follows:

- The naive implementation of our NbGClock(atomic) has a global contention point on the CLOCKHAND. The AtomicInteger uses *compare-and-swap* operations for each decrement/increment operation, and thus the bus lock decreases its scalability. On the other hand, NbGClock(stripe) employs a striped counter as shown in Figure 6.6, which scales beyond 32 processors.

- A shared counter tends to be contended on multiprocessors since increment/decrement

buffer capacity	LRU count	LRU %	2Q count	2Q %	Nb-GClock count	Nb-GClock %
4096	6655	2.2	3590	1.2	88304	10.6
8192	3473	1.1	2391	0.7	64956	6.3
16384	2053	0.5	1541	0.4	53280	3.5
32768	698	0.1	779	0.2	67070	1.8

Table 6.3. Contentions generated by pread.



processors (cores)	8 (1)	16 (2)	32 (4)	64 (8)
LRU	702011	668649	680883	695153
2Q	648589	640807	656188	662782
NbGClock(stripe)	3358893	6391329	12686899	24883432
NbGClock(atomic)	3582785	7160720	13100118	9410744
E$LRU	702011	1404022	2808044	5616088
E$NbGClock	3358893	6717786	13435572	26871144

Figure 6.13. Scalability to processors when pages are resident in memory.

involves costful write operations.

Notably, it can be concluded that the existing locking-based schemes did not scale more than 16 processors according to the results in Figure 6.13.

We also conducted a performance measurement on varying the number of processors when disk I/Os were performed by using *pread*. The results in Figure 6.14 showed that only the proposed scheme can obtain at least log-linear performance relative to the number of processors up to 64 processors.

Based on the above experimental results, we conclude that our non-blocking scheme is much more scalable than existing schemes in certain situations and has a significant

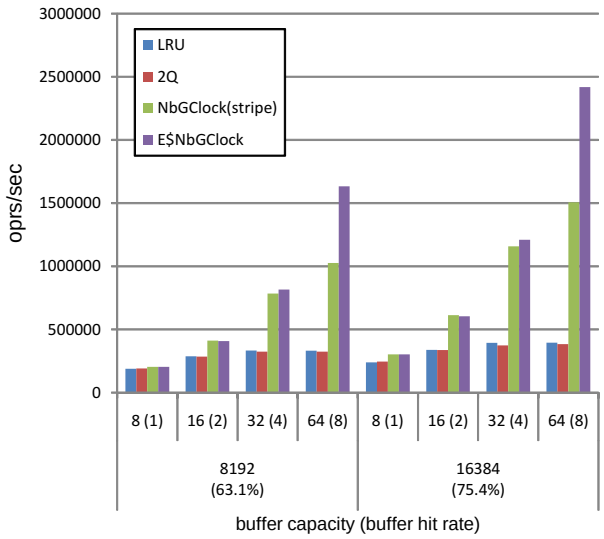


Figure 6.14. Scalability to processors when using pread for disk I/O.

advantage over the existing blocking schemes, as confirmed with Figure 6.13 where all pages are resident in memory and Figure 6.14 where disk I/Os are performed by pread under adequate hit rates.

6.4.2 Experiments on x86-64 Architecture

In Section **6.4.1**, our non-blocking scheme showed significant performance improvement on a SUN UltraSPARC T2. However, it still needs to prove its efficiency on other architectures. We have thus conducted an experiment on two different x86-64 architectures as listed in Table 6.4.

We compared our non-blocking buffer management scheme using Nb-GCLOCK to the existing locking-based schemes with respect to throughputs where all pages are resident in memory and we assume eight concurrent accesses (i.e., the same as the number of processors) to the module. We used LRU and 2Q for the page replacement algorithms, as with the existing locking-based schemes, and performed the 80/20 workloads for each algorithm.

Figure 6.15 shows the results of the experiment. Our schemes (all variations of Nb-GCLOCK) outperform the existing locking-based ones by more than 5 and 4 times on

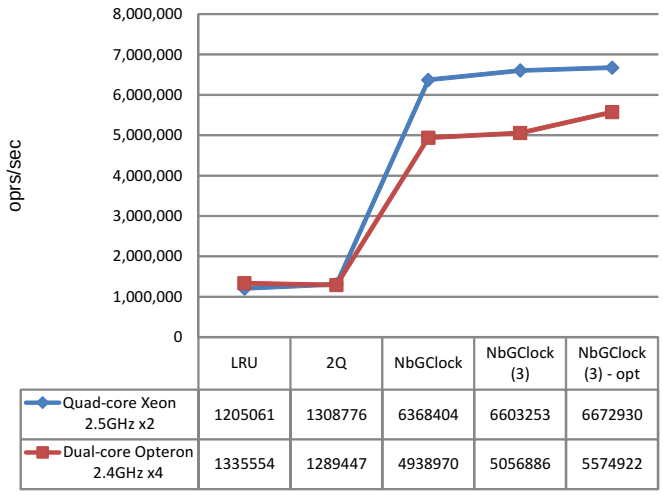
Operating System	Linux 2.6.22 OpenSUSE 10.3	Linux 2.6.5 SuSE (SLES) 9
CPU model	Quad core Xeon E5420	Dual Core Opteron 880
Architecture	SMP	ccNUMA
Core (Chips)	8 (2)	8 (4)
Processor frequency	2.5 GHz	2.4 GHz
Main memory	8 GB	32 GB
Disk	SATA 2 (7200 rpm, NCQ)	Ultra320 SCSI (10000 rpm)
L2 cache per core	6 MB	1 MB

Table 6.4. Specifications of each X86-64 machine.

the Xeon SMP architecture and the Opteron ccNUMA architecture, respectively. Note that this performance gain for 8 concurrent accesses is similar to the one expected on the SUN UltraSPARC T2 (at most 4.78 times in Figure 6.13).

The clear differences appearing on Nb-GCLOCK between the two architecture can be attributed to the greater number of contentions among chips performed on the Opteron configuration as the throughput increases. Of course, CAS incurs more latency on a four-chip configuration than a two-chip configuration because CAS (i.e., cmpxchg) causes bus locks. To reduce *false sharing* in the Opteron configuration, we striped the memory location of array elements used for the buffer pool as depicted in Figure 6.16. The "NbGClock(3) - opt" in Figure 6.15 reduces *false sharing* effects on multi-chip configurations. The "NbGClock(3)" means that the maximum value of GCLOCK's reference counter is restricted to 3. This effort is introduced to reduce CAS instructions because CMPXCHG is very costly on Intel x86 multiprocessor systems while an UltraSPARC T2 processor has a very cheap CAS instruction. UltraSPARC T2 and Opteron multiprocessor systems show better CAS and CMPXCHG performance [RD06, Izv].

Based on the above results, we conclude that even under medium multithreaded environments of x86 architectures, our proposed non-blocking scheme can provide better performance than the existing lock-based schemes and is thus the algorithm of choice.



	LRU	2Q	NbGClock	NbGClock (3)	NbGClock (3) - opt
Quad-core Xeon 2.5GHz x2	1205061	1308776	6368404	6603253	6672930
Dual-core Opteron 2.4GHz x4	1335554	1289447	4938970	5056886	5574922

Figure 6.15. Experiment on X86-64 architecture (8 threads).

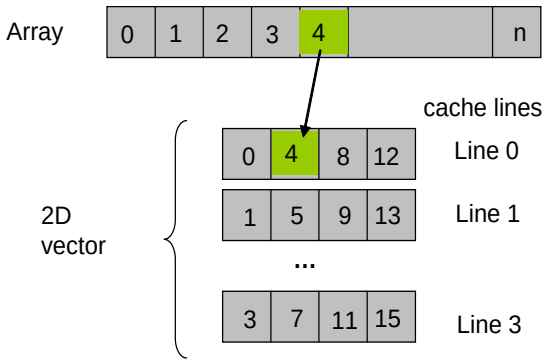


Figure 6.16. Expressing an array as a two-dimensional vector.

6.4.3 Experiment on an XML database

This section gives our initially motivated problem that massively concurrent access to an XML database is given. To simulate the situation, we used an XML data set of XMark [SWK+01] SF=1 and their 17 XQuery queries excluding Q10, Q11, and Q12 (see Appendix A). We excluded the three queries simply because including those slowest queries (see Table 4.3 for detail) makes the evaluation inadequate. We compare Nb-GCLOCK and 2Q [JS94] with respect to the turnaround time of a workload. We preparedly loaded the XML data set into our XML database instance running on Sun UltraSPARC T2 as in Table 6.2. We used a machine in Table 6.4 that equips quad core Xeon E5420 as a client workload simulator. The client workload consists of N sets of the 17 queries and the workload simulator randomly executes, with 64 threads, the queries on Sun UltraSPARC T2.

Figure 6.5 shows the results using 6 and 10 for N. Unexpectedly large difference is caused by contentions generated on slowest queries. The contention made the standard deviation of 2Q experiments large; 156.141 seconds and 444.304 seconds when N=6 and N=10 respectively. From this result, we conclude that improving concurrency in a buffer management module is important not only for RDBMSs but also for a certain XML database system.

	2Q	GCLOCK
N=6 (102 queries)	718.59	118.64
N=10 (170 queries)	1833.82	136.76

Table 6.5. Comparison of turnaround time between 2Q and GLOCK on XBird (in sec).

6.5 Related Work

An analytical and empirical study of GCLOCK is performed in [NDD92]. Our Nb-GCLOCK algorithm basically follows the GCLOCK properties.

Tsuei et al. [TPK97] designed experiments to investigate how the database size, the buffer size, and the number of CPUs impact database performance, in particular, throughputs and buffer hit rates on Symmetric Multiprocessor (SMP) systems.

They investigated the impact of buffer size on performance using the TPC-C workload and observed that an adequate memory buffer size is relatively small compared to the database size. They also suggested a rule-of-thumb of 10-15% of the database size to achieve more than an 80% buffer hit rate. As shown in our experiment in Figure 6.12, Nb-GCLOCK becomes very effective when the buffer hit rate is about 80% or more. These requirements for buffer capacity can realistically be considered acceptable because 64-bit processors, which have a huge address space, have become widespread and, moreover, DRAM has become dense and cheap.

Zhou et al. [ZCRS05] investigated thread-based techniques to exploit database operations of memory-resident data for simultaneous multithreading architectures. They used, in the spin-loop waiting, *PAUSE* instructions on Pentium 4.

To avoid lock contentions on the LRU chain, ADABAS [Sch98] splits the buffer pool into several physical regions, where each region has its own LRU chain. However, this approach can reduce buffer hit rates, especially when the distribution of hash values has skew. In addition, they did not discuss how buffer hit rates change by dividing the LRU chain.

Shared counting on shared memory multiprocessors has been studied, for example in [HLS95, MTY92].

6.6 Summary

In this chapter, we proposed a lock-free variant of the GCLOCK page replacement algorithm, named Nb-GCLOCK. We introduced a non-blocking scheme for *buffer-fix* operations that fix buffer frames for requested pages without locks by combining Nb-GCLOCK and a wait-free hash table. Our experimental results revealed that our scheme can obtain nearly linear scalability to processors up to 64 processors, while the existing locking-based schemes do not scale beyond 16 processors.

Gray et al. suggested in [GR92] that a future database system might introduce its page replacement at random since it could have a huge buffer pool. This can be a strategy of choice in a certain situation, though the behavior of the worst case cannot meet the needs of critical systems. Ensuring lock-freedom operation, as in our scheme, is considered preferable for such practical requirements.

The proposed scheme is effective not only for database buffer management but

also for general purpose caching that requires synchronization. One example of such an application would be scalable query result caching for web applications, where the requests come from over 1000 clients simultaneously [Keg]; another key use would be certain real-time systems that have (soft) real-time constraints, such as real-time operating systems.

7 Conclusion

Contributions

This dissertation has proposed three distinct (but associative with few exceptions) methods for making a scalable XML database system.

Let us recollect our research goal defined for this dissertation in Chapter **1**:

#1 building an XML database system scalable to data volumes,

#2 building a scalable XML processor with a shared-nothing PC cluster, and

#3 making database processing scalable on shared-memory multiprocessors.

The contributions of this dissertation are summarized as follows.

Relevant to research goal #1, we proposed, in Chapter **4**, an XQuery processing scheme in which an XML document is internally represented as a set of blocks and can directly be stored on secondary storage. The results of our experiments clearly showed that the proposed scheme can often obtain almost linear scalability in performance as the data size increases, and thus achieves the goal of #1.

In Chapter **5**, we studied on-the-fly XML processing using shared-nothing PC clusters. We propose a scheme for distributed and parallel query processing that employs a pass-by-reference semantics by using remote proxy. Our experimental results showed up to 22x speedups compared with competitive methods, and demonstrated the importance for distributed XML database systems to take *pass-by-reference* semantics into

consideration. We thus conclude that our attempt represents a major step forward for #2.

In Chapter **6**, we proposed scalable buffer management scheme that employs non-blocking synchronization instead of locking-based ones. Our experimental results revealed that our scheme can obtain nearly linear scalability to processors up to 64 processors. The results clearly achieve the goal of #3. The tendency of increasing the number of CPU cores drives databases to multithreaded implementation and concurrency is an issue they have to deal with; and our proposal is a good attempt to address this problem.

We compiled all proposed methods into one system, namely XBird. From the above achievements, we conclude that our initial goal was achieved through parallel efforts on scalability improvements on XBird. More detailed contributions are summarized below.

Scalable XML Storage Scheme

In Chapter **4**, we analyzed access patterns that frequently appear for XML queries. We consider that this is a strong point of this study because such a practical analysis was not provided before.

We also demonstrate the importance for XML database systems to take informed prefetching and scan-resistant caching into consideration. Our scheme based on DTM is adoptable to popular XQuery/XPath processors [Sax, Apac] because they use either DTM or a similar internal data structure to DTM.

Efficient XML Data Exchange between Processor Elements

The first strong point, in Chapter **5**, is that we addressed the scalability limit on hierarchical distributed XML processing which is mandatory for divide-and-conquer approaches. We addressed problems of distributed XML query processing in detail and explain how the problems differ from traditional database problems.

We demonstrated that considering pipeline parallelism is clearly important for distributed XML query processing which so far is not enough discussed in the context of XML query processing. Moreover, we introduced parallel database techniques with ap-

propriate modifications, e.g., independent-operator parallelism and inter-operator load balancing, to XML query processing.

Non-blocking Buffer Management

In Chapter **6**, we explained the internal locking in buffer management. We introduced how concurrency in the page replacement mechanism is addressed for each LRU, 2Q and GCLOCK algorithms. Furthermore we considered spin lock techniques for handling the concurrency and discussed cons for each of the methods which Nb-GCLOCK tried to address.

We further provided semi-formal correctness (*linearizability*) proof of our lock-free algorithm.

Future Work

As a further goal of #2, we are engaged on optimizing our system on larger PC-cluster environment. When running a system on a large PC-cluster, the MTBF tends to be lower. For the dependable computing, we have to make our system robust to node failures. Issues to be explored for #3 include evaluating our system with the TPC-C benchmark [Cou07] and porting our enhancement to public domain databases.

We noticed through this study that “Rare chance is coming to revise classic algorithms in database core modules”. Since the coming many-core era that computers have multiple processors for granted and processing power tends to vary dramatically, database systems thus should change the behavior dynamically (or statically) by introducing algorithm selection mechanism. We advance this study to make *XBird* the next generation database management system.

References

[ABC+03] Serge Abiteboul, Angela Bonifati, Grégory Cobéna, Ioana Manolescu, and Tova Milo. Dynamic XML Documents with Distribution and Replication. In *Proc. SIGMOD*, pages 527–538, 2003.

[Abi97] Serge Abiteboul. Querying Semi-Structured Data. In *Proc. ICDT*, pages 1–18, 1997.

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, October 1999.

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Fourth Edition*. Addison-Wesley Professional, August 2005.

[AKJP+02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE*, pages 141–152, 2002.

[Amd00] Gene M. Amdahl. *The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. Morgan Kaufmann Publishers Inc., 2000.

[And90] Thomas. E. Anderson. The Performance of Spin Lock Alternatives for

Shared-Money Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990.

[Apaa] Apache Software Foundation. Apache Xindice. http://xml.apache.org/xindice/.

[Apab] Apache Software Foundation. The Apache Derby Project. http://db.apache.org/derby/.

[Apac] Apache Software Foundation. The Apache Xalan Project. http://xml.apache.org/xalan-j/.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason Mchugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[AYBB+08] Sihem Amer-Yahia, Chavdar Botev, Stephen Buxton, Pat Case, Jochen Doerre, Mary Holstege, Jim Melton, Michael Rys, and Jayavel Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text. http://www.w3.org/TR/xpath-full-text-10/, 2008.

[Azu] Azul Systems, Inc. http://www.azulsystems.com/.

[BCJ+05] Kevin Beyer, Roberta J. Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy Lohman, Bob Lyle, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Tuong Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System RX: One Part Relational, One Part XML. In *Proc. SIGMOD*, pages 347–358, 2005.

[BG03] Jan-Marco Bremer and Michael Gertz. On Distributing XML Repositories. In *Proc. WebDB*, pages 73–78, 2003.

[BGJM04] Roberto. J. Bayardo, Daniel Gruhl, Vanja Josifovski, and Jussi Myllymaki. An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing. In *Proc. WWW*, pages 345–354, 2004.

[BGMP79] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. The Convoy Phenomenon. *SIGOPS Oper. Syst. Rev.*, 13(2):20–25, April 1979.

[BGvK+06] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. SIGMOD*, pages 479–490, 2006.

[BJK+97] William Bridge, Ashok Joshi, M. Keihl, Tirthankar Lahiri, Juan Loaiza, and N. Macnaughton. The Oracle Universal Server Buffer. In *Proc. VLDB*, pages 590–594, 1997.

[BKF+07] Irina Botan, Donald Kossmann, Peter M. Fischer, Tim Kraska, Dana Florescu, and Rokas Tamosevicius. Extending XQuery with Window Functions. In *Proc. VLDB*, pages 75–86, 2007.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. SIGMOD*, pages 310–321, 2002.

[BLS97] Gerald Brose, Klaus-Peter Löhr, and André Spiegel. Java does not distribute. In *Proc. Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 144–152, 1997.

[BM04] Paul V. Biron and Ashok Malhotra, editors. *XML Schema Part 2: Datatypes*. W3C Recommendation. W3C, second edition, October 2004.

[BPSM+03] Tim Bray, Jean Paoli, C. M. Sperberg-Mcqueen, Eve, and François Yergeau, editors. *Extensible Markup Language (XML) 1.0*. W3C Recommendation. W3C, fourth edition, August 2003.

[Bun97] Peter Buneman. Semistructured Data. In *Proc. PODS*, pages 117–121, 1997.

[CA95] R.G.G. Cattell and Tom Atwood. *Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann Publishers Inc., 1995.

[CBHR06] John Cieslewicz, Jonathan Berry, Bruce Hendrickson, and Kenneth A. Ross. Realizing Parallelism in Database Operations: Insights from a Massively Multithreaded Architecture. In *Proc. DaMoN*, 2006.

[CCF+06] Don Chamberlin, Michael Carey, Daniela Florescu, Donald Kossmann, and Jonathan Robie. XQueryP: Programming with XQuery. In *Proc. XIME-P*, 2006.

[CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. SIGMETRICS*, pages 188–197, 1995.

[CKS+00] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *Proc. VLDB*, pages 646–648, 2000.

[Cli] Cliff Click. high-scale-lib. http://sourceforge.net/projects/high-scale-lib.

[CLYY92] Ming S. Chen, Ming L. Lo, Philip S. Yu, and Honesty C. Young. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *Proc. VLDB*, pages 15–26, 1992.

[CM01] James Clark and MURATA Makoto. RELAX NG Specification (Committee Specification). *http://www.oasis-open.org/committees/relax-ng*, 12(03), 2001.

[CMV05] Barbara Catania, Anna Maddalena, and Athena Vakali. XML Document Indexes: A Classification. *IEEE Internet Computing*, 9(5):64–71, 2005.

[Cor69] Fernando. J. Corbató. A Paging Experiment with the Multics System. In Feshbach and Ingard, editors, *In Honor of Philip M. Morse*, page 217. MIT Press, Cambridge, Mass, 1969.

[Cou07] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification, Revision 5.9. http://www.tpc.org/tpcc/, June 2007.

[CRF01] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proc. WebDB*, pages 1–25. 2001.

[CRG07] John Cieslewicz, Kenneth A. Ross, and Ioannis Giannakakis. Parallel Buffers for Chip Multiprocessors. In *Proc. DaMoN*, 2007.

[CT04] John Cowan and Richard Tobin. XML Information Set (Second Edition). W3C Recommendation, Feb 2004.

[DF82] Lawrence W. Dowdy and Derrell V. Foster. Comparative Models of the File Assignment Problem. *ACM Comput. Surv.*, 14(2):287–313, 1982.

[DFF+98] Alin Deutsch, Mary Fernández, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/, 1998.

[DG92] David Dewitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Processing. *Communications of the ACM*, 35(6):85–98, 1992.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*, pages 137–150, 2004.

[DS08] David J. DeWitt and Michael Stonebraker. MapReduce: A Major Step Backwards. The Database Column http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html, January, 17 2008.

[EH84] Wolfgang Effelsberg and Theo Haerder. Principles of Database Buffer Management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984.

[Eng07] Daniel Engovatov. XML Query 1.1 Requirements. http://www.w3.org/TR/xquery-11-requirements, 2007.

[FHK+04] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, and Arvind Sundararajan. The BEA Streaming XQuery Processor. *VLDB Journal*, 13(3):294–315, 2004.

[FJM+07a] Mary Fernández, Trevor Jim, Kristi Morton, Nicola Onose, and Jerome Simeon. Highly Distributed XQuery with DXQ. In *Proc. SIGMOD*, 2007.

[FJM+07b] Mary F. Fernández, Trevor Jim, Kristi Morton, Nicola Onose, and Jerome Simeon. DXQ: A Distributed XQuery Scripting Language. In *Proc. XIME-P*, 2007.

[FK99] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[FKS+02] Mary Fernández, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Trans. Database Syst.*, 27(4):438–493, 2002.

[Fra05] Massimo Franceschet. XPathMark: An XPath benchmark for XMark. Research report PP-2005-04, University of Amsterdam, Netherlands, 2005.

[GMW99] Roy Goldman, Jason McHugh, and Jennifer Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Proc. WebDB*, pages 25–30, 1999.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, 1992.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. VLDB*, pages 436–445, 1997.

[HHN+00] Arnaud Le Hors, Philippe Le Héaret, Lauren Wood Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) Level 2 Core Specification. http://www.w3.org/TR/DOM-Level-2-Core/, 2000.

[HLAM06] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance Tradeoffs in Read-Optimized Databases. In *Proc. VLDB*, pages 487–498, 2006.

[HLS95] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. Scalable Concurrent Counting. *ACM Trans. Comput. Syst.*, 13(4):343–364, November 1995.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[HW87] Maurice P. Herlihy and Jeannette M. Wing. Axioms for Concurrent Objects. In *POPL*, pages 13–26, 1987.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[IAC] IAC Search & Media. Bloglines. http://www.bloglines.com/.

[IHW02] Z.G. Ives, AY Halevy, and DS Weld. An XML query engine for network-bound data. *The VLDB Journal*, 11(4):380–402, 2002.

[Ili61] John K. Iliffe. *The Use of The Genic System in Numerical Calculations*, volume 2 of *Annual Review in Automatic Programming*. 1961.

[Inf] Information-technology Promotion Agency, Japan. IPA OSS iPedia. http://ossipedia.ipa.go.jp/en/
http://ossipedia.ipa.go.jp/en/capacity/EV0612250287EN/
http://ossipedia.ipa.go.jp/en/capacity/EV0612260303EN/.

[Int01] Intel Corporation. Using spin-loops on Intel Pentium 4 processor and Intel Xeon processor. Order Number: 248674-002, May 2001.

[Izv] Alex Izvorski. mubench. http://mubench.sourceforge.net/.

[JAKC+02] H.V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. TIMBER: A Native XML Database. *VLDB Journal*, 11(4):274–291, December 2002.

[JCZ05] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proc. USENIX*, page 35, April 2005.

[JLWO03] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proc. ICDE*, pages 253–264, 2003.

[JPA08] Ryan Johnson, Ippokratis Pandis, and Anastassia Ailamaki. Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines. In *Proc. DaMoN*, 2008.

[JS94] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. VLDB*, pages 439–450, 1994.

[Keg] Dan Kegel. The C10K problem. http://www.kegel.com/c10k.html.

[KHMU07] Hiroto Kurita, Kenji Hatano, Jun Miyazaki, and Shunsuke Uemura. Efficient Query Processing for Large XML Data in Distributed Environments. *Proc. AINA*, 0:317–322, 2007.

[KM00] Carl-Christian Kanne and Guido Moerkotte. Efficient Storage of XML Data. In *Proc. ICDE*, page 198, 2000.

[Knu98] Donald E. Knuth. *Art of Computer Programming*, volume 3. Addison-Wesley Professional, April 1998.

[KP05] Georgia Koloniari and Evaggelia Pitoura. Peer-to-Peer Management of XML Data: Issues and Research Challenges. *SIGMOD Rec.*, 34(2):6–17, June 2005.

[Lea] Doug Lea. The JSR-133 Cookbook for Compiler Writers. http://gee.cs.oswego.edu/dl/jmm/cookbook.html.

[Leh] Marc Alexander Lehmann. lzf - an extremely fast/free compression/decompression-method. http://liblzf.plan9.de/.

[LR05] Bin Liu and Elke A. Rundensteiner. Revisiting Pipelined Parallelism in Multi-Join Query Processing. In *Proc. VLDB*, pages 829–840, 2005.

[MAA+05] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Fred Dang Ngoc. Exchanging Intensional XML Data. *ACM Trans. Database Syst.*, 30(1):1–40, 2005.

[MAG+97] Jason Mchugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Rec.*, 26(3):54–66, September 1997.

[MBB+06] Norman May, Matthias Brantner, Alexander Böhm, Carl-Christian Kanne, and Guido Moerkotte. Index vs. Navigation in XPath Evaluation. In *Proc. XSym*, volume 4156, pages 16–30. 2006.

[Mei06] Wolfgang Meier. Index-driven XQuery processing in the eXist XML database. XML Prague, 2006.

[MLLA03] Xiaofeng Meng, Daofeng Luo, Mong-Li Lee, and Jing An. OrientStore: A Schema Based Native XML Storage System. In *Proc. VLDB*, pages 1057–1060, 2003.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. *SIGPLAN Not.*, 40(1):378–391, January 2005.

[MS99] Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proc. ICDT*, pages 277–295. 1999.

[MTY92] Shlomo Moran, Gadi Taubenfeld, and Irit Yadin. Concurrent Counting. In *Proc. PODC*, pages 59–70, 1992.

[NDD92] Victor F. Nicola, Asit Dan, and Daniel M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. *SIGMETRICS Perform. Eval. Rev.*, 20(1):35–46, June 1992.

[NJ03] Matthias Nicola and Jasmi John. XML Parsing: A Threat to Database Performance. In *Proc. CIKM*, 2003.

[NS] Mark Nottingham and Robert Sayre. The Atom Syndication Format, RFC 4287. http://www.ietf.org/rfc/rfc4287.

[OV99] Tamer M. Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition, 1999.

[PAM96] Yannis Papakonstantinou, Serge Abiteboul, and Hector G. Molina. Object Fusion in Mediator Systems. In *Proc. VLDB*, pages 413–424, 1996.

[PC03] Feng Peng and Sudarshan S. Chawathe. XPath Queries on Streaming Data. In *Proc. SIGMOD*, pages 431–442, 2003.

[PH05] Chris Purcell and Tim Harris. Non-blocking Hashtables with Open Addressing. In *Proc. Distributed Computing (DISC)*, pages 108–121. Springer, 2005.

[RBHS04] Christopher Ré, Jim Brinkley, Kevin Hinshaw, and Dan Suciu. Distributed XQuery. In *Proc. IIWeb*, pages 116–121, 2004.

[RD06] Kenneth Russell and David Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. *SIGPLAN Not.*, 41(10):263–272, October 2006.

[RG02] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Science/ Engineering / Math, third ed. edition, August 2002.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html, 1998.

[Roh95] Hans Rohnert. The Proxy Design Pattern Revisited. *Pattern Languages of Program Design*, 1995.

[RSS00] RSS-DEV Working Group. RDF Site Summary (RSS) 1.0. http://purl.org/rss/1.0/spec, 2000.

[Sax] Saxonica Ltd. Saxon. http://www.saxonica.com/.

[Sch98] Harald Schöning. The ADABAS Buffer Pool Manager. In *Proc. VLDB*, pages 675–679, 1998.

[SD90] Donovan A. Schneider and David J. Dewitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *Proc. VLDB*, pages 469–480, 1990.

[Smi78] Alan J. Smith. Sequentiality and Prefetching in Database Systems. *ACM Trans. Database Syst.*, 3(3):223–247, September 1978.

[SS06] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, May 2006.

[Sto86] Michael Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.

[Suc02] Dan Suciu. Distributed Query Evaluation on Semistructured Data. *ACM Trans. Database Syst.*, 27(1):1–62, March 2002.

[Sun] Sun Microsystems, Inc. SUN UltraSPARC T2 Processor. http://www.sun.com/processors/UltraSPARC-T2/.

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[SWK+01] Albrecht R. Schmidt, Florian Waas, Martin L. Kersten, Daniela Florescu, Ioana Manolescu, Mike J. Carey, and Ralph Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, 2001.

[SYT93] Eugene J. Shekita, Honesty C. Young, and Kian-Lee Tan. Multi-Join Optimization for Symmetric Multiprocessors. In *Proc. VLDB*, pages 479–492, 1993.

[TDCZ02] Feng Tian, David J. DeWitt, Jianjun Chen, and Chun Zhang. The Design and Performance Evaluation of Alternative XML Storage Strategies. *SIGMOD Record*, 31(1):5–10, 2002.

[The01] The XML:DB Initiative. Application Programming Interface for XML Databases (Working Draft). http://xmldb-org.sourceforge.net/, Sept. 2001.

[Tol07] Doug Tolbert. Scaling PostgreSQL on SMP Architectures - An Update. In *The PostgreSQL Conference*, 2007.

[TPK97] Thin-Fong Tsuei, Allan N Packer, and Keng-Tai Ko. Database Buffer Size Investigation for OLTP Workloads. In *Proc. SIGMOD*, pages 112–122, 1997.

[TVB+02] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. SIGMOD*, pages 204–215, 2002.

[Val96] John D. Valois. *Lock-free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 1996.

[W3Ca] W3C. XML Path Language (XPath) 1.0. http://www.w3.org/TR/xpath.

[W3Cb] W3C. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/.

[W3Cc] W3C. XML Schema. http://www.w3.org/XML/Schema.

[W3Cd] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.

[W3Ce] W3C. XQuery 1.0 and XPath 2.0 Data Model (XDM). http://www.w3.org/TR/xpath-datamodel/.

[W3Cf] W3C. XQuery 1.0 and XPath 2.0 Formal Semantics. http://www.w3.org/TR/xquery-semantics/.

[W3Cg] W3C. XQuery Update Facility. http://www.w3.org/TR/xqupdate/.

[Wid99] Jennifer Widom. Data Management for XML: Research Directions. *IEEE Data Engineering Bulletin*, 22:44–52, 1999.

[YASU01] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Trans. Internet Technology (TOIT)*, 1(1):110–141, 2001.

[YMUK07] Makoto Yui, Jun Miyazaki, Shunsuke Uemura, and Hirokazu Kato. Efficient XML Storage based on DTM for Read-oriented Workloads. In *Proc. IEEE International Workshop on Advanced Storage Systems (ADSS)*, pages 559–564. IEEE CS Press, October 2007.

[YÖK04] Benjamin Bin Yao, M. Tamer Özsu, and Nitin Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proc. ICDE*, 2004.

[YS05] Sihem A. Yahia and Jayavel Shanmugasundaram. XML Full-Text Search: Challenges and Opportunities. In *Proc. VLDB*, page 1368, 2005.

[ZB07a] Ying Zhang and Peter A. Boncz. Integrating XQuery and P2P in MonetDB/XQuery. In *Proc. Workshop on Emerging Research Opportunities for Web Data Management (EROW)*, 2007.

[ZB07b] Ying Zhang and Peter A. Boncz. XRPC: Interoperable and Efficient Distributed XQuery. In *Proc. VLDB*, 2007.

[ZCRS05] Jingren Zhou, John Cieslewicz, Kenneth A. Ross, and Mihir Shah. Improving Database Performance on Simultaneous Multithreading Processors. In *Proc. VLDB*, pages 49–60, 2005.

[ZKO04] Ning Zhang, Varun Kacholia, and M. Tamer Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proc. ICDE*, pages 54–65, 2004.

Appendix

A XMark queries

The followings are XMark benchmark [SWK+01] queries formulated in XQuery.

Q1. *Return the name of the person with ID 'person0'.*

```
let $auction := fn:doc("auction.xml")
return
  for $b in $auction/site/people/person[@id = "person0"]
  return $b/name/text()
```

Q2. *Return the initial increases of all open auctions.*

```
let $auction := fn:doc("auction.xml")
return
  for $b in $auction/site/open_auctions/open_auction
  return <increase>{ $b/bidder[1]/increase/text() }</increase>
```

Q3. *Return the first and current increases of all open auctions whose current increase is at least twice as high as the initial increase.*

```
let $auction := fn:doc("auction.xml")
return
  for $b in $auction/site/open_auctions/open_auction
  where zero-or-one(
```

```
   $b/bidder[1]/increase/text()) * 2 <= $b/bidder[last()]/increase/text())
  return <increase first="{ $b/bidder[1]/increase/text() }"
                   last="{ $b/bidder[last()]/increase/text() }"/>
```

Q4. *List the reserves of those open auctions where a certain person issued a bid before another person.*

```
let $auction := fn:doc("auction.xml")
return
  for $b in $auction/site/open_auctions/open_auction
  where some $pr1 in $b/bidder/personref[@person = "person20"],
             $pr2 in $b/bidder/personref[@person = "person51"]
        satisfies $pr1 << $pr2
  return <history>{ $b/reserve/text() }</history>
```

Q5. *How many sold items cost more than 40?*

```
let $auction := fn:doc("auction.xml")
return
  count(for $i in $auction/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price)
```

Q6. *How many items are listed on all continents?*

```
let $auction := fn:doc("auction.xml")
return
  for $b in $auction//site/regions
  return count($b//item)
```

Q7. *How many pieces of prose are in our database?*

```
let $auction := fn:doc("auction.xml")
return
  for $p in $auction/site
  return count($p//description) + count($p//annotation)
         + count($p//emailaddress)
```

Q8. *List the names of persons and the number of items they bought. (joins person, closed_auction, item)*

```
let $auction := doc("auction.xml")
return
  for $p in $auction/site/people/person
  let $a := for $t in $auction/site/closed_auctions/closed_auction
            where $t/buyer/@person = $p/@id
            return $t
  return <item person="{ $p/name/text() }">{ count($a) }</item>
```

Q9. *List the names of persons and the names of the items they bought in Europe. (joins person, closed_auction, item)*

```
let $auction := fn:doc("auction.xml")
return
  let $ca := $auction/site/closed_auctions/closed_auction
  return
    let $ei := $auction/site/regions/europe/item
    for $p in $auction/site/people/person
    let $a := for $t in $ca
              where $p/@id = $t/buyer/@person
              return
                let $n := for $t2 in $ei
                          where $t/itemref/@item = $t2/@id
                          return $t2
                return <item>{ $n/name/text() }</item>
    return <person name="{ $p/name/text() }">{ $a }</person>
```

Q10. *List all persons according to their interest; use French markup in the result.*

```
let $auction := fn:doc("auction.xml")
return
  for $i in distinct-values(
    $auction/site/people/person/profile/interest/@category
  )
  let $p := for $t in $auction/site/people/person
            where $t/profile/interest/@category = $i
            return
              <personne>
                <statistiques>
                  <sexe>{ $t/profile/gender/text() }</sexe>
                  <age>{ $t/profile/age/text() }</age>
```

```
                <education>{ $t/profile/education/text() }</education>
                <revenu>{ fn:data($t/profile/@income) }</revenu>
              </statistiques>
              <coordonnees>
                <nom>{ $t/name/text() }</nom>
                <rue>{ $t/address/street/text() }</rue>
                <ville>{ $t/address/city/text() }</ville>
                <pays>{ $t/address/country/text() }</pays>
                <reseau>
                  <courrier>{ $t/emailaddress/text() }</courrier>
                  <pagePerso>{ $t/homepage/text() }</pagePerso>
                </reseau>
              </coordonnees>
              <cartePaiement>{ $t/creditcard/text() }</cartePaiement>
            </personne>
  return <categorie>{ <id>{ $i }</id>, $p }</categorie>
```

Q11. *For each person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.*

```
let $auction := fn:doc("auction.xml")
return
  for $p in $auction/site/people/person
  let $l := for $i in $auction/site/open_auctions/open_auction/initial
            where $p/profile/@income > 5000 * exactly-one($i/text())
            return $i
  return <items name="{ $p/name/text() }">{ count($l) }</items>
```

Q12. *For each richer-than-average person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.*

```
let $auction := fn:doc("auction.xml")
return
  for $p in $auction/site/people/person
  let $l := for $i in $auction/site/open_auctions/open_auction/initial
            where $p/profile/@income > 5000 * exactly-one($i/text())
            return $i
  where $p/profile/@income > 50000
  return <items person="{ $p/profile/@income }">{ count($l) }</items>
```

Q13. *List the names of items registered in Australia along with their descriptions.*

```
let $auction := fn:doc("auction.xml")
return
  for $i in $auction/site/regions/australia/item
  return <item name="{ $i/name/text() }">{ $i/description }</item>
```

Q14. *Return the names of all items whose description contains the word 'gold'.*

```
let $auction := fn:doc("auction.xml")
return
  for $i in $auction/site//item
  where contains(string(exactly-one($i/description)), "gold")
  return $i/name/text()
```

Q15. *Print the keywords in emphasis in annotations of closed auctions.*

```
let $auction := fn:doc("auction.xml")
return
  for $a in
   $auction/site/closed_auctions/closed_auction/annotation/desc-
ription/parlist/listitem/parlist/listitem/text/emph/keyword/text()
  return <text>{ $a }</text>
```

Q16. *Return the IDs of those auctions that have one or more keywords in emphasis. (cf. Q15)*

```
let $auction := fn:doc("auction.xml")
return
  for $a in $auction/site/closed_auctions/closed_auction
  where not(empty(
$a/annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/text()
  ))
  return <person id="{ $a/seller/@person }"/>
```

Q17. *Which persons don't have a homepage?*

```
let $auction := fn:doc("auction.xml")
return
  for $p in $auction/site/people/person
  where empty($p/homepage/text())
  return <person name="{ $p/name/text() }"/>
```

Q18. *Convert the currency of the reserve of all open auctions to another currency.*

```
declare namespace local = "http://www.foobar.org";
declare function local:convert($v as xs:decimal?) as xs:decimal?
{
  2.20371 * $v
(: convert Dfl to Euro :)
};

let $auction := fn:doc("auction.xml")
return
  for $i in $auction/site/open_auctions/open_auction
  return local:convert(zero-or-one($i/reserve))
```

Q19. *Give an alphabetically ordered list of all items along with their location.*

```
let $auction := fn:doc("auction.xml")
return
  for $b in $auction/site/regions//item
  let $k := $b/name/text()
  order by zero-or-one($b/location) ascending empty greatest
  return <item name="{ $k }">{ $b/location/text() }</item>
```

Q20. *Group customers by their income and output the cardinality of each group.*

```
let $auction := fn:doc("auction.xml")
return
  <result>
    <preferred>
      { count($auction/site/people/person/profile[@income >= 100000]) }
    </preferred>
    <standard>
      {
       count(
$auction/site/people/person/profile[@income < 100000 and @income >= 30000]
       )
      }
    </standard>
    <challenge>
      { count($auction/site/people/person/profile[@income < 30000]) }
    </challenge>
```

```
    <na>
      {
        count(for $p in $auction/site/people/person
        where empty($p/profile/@income)
        return $p)
      }
    </na>
  </result>
```

List of Publications

Journal Papaers

1. Makoto Yui, Jun Miyazaki, Shunsuke Uemura, Hirokazu Kato: "Distributed XQuery Processing using Remote Proxy", IPSJ Transactions on Databases (TOD 41), Information Processing Society of Japan, March. 2009. (*Accepted*)

2. Makoto Yui, Jun Miyazaki, Shunsuke Uemura: "XML Storage Based on DTM for Efficient XQuery Processing", IPSJ Transactions on Databases, Vol. 48, No. SIG 11 (TOD 34), pp. 128–148, Information Processing Society of Japan, June. 2007.

International Conferences

1. Makoto Yui, Jun Miyazaki, Shunsuke Uemura and Hirokazu Kato: "XBird/D: Distributed and Parallel XQuery Processing using Remote Proxy", Proc. ACM SYMPOSIUM ON APPLIED COMPUTING (SAC), Special Track on Database Theory, Technology, and Applications, pp. 1003-1007, March 2008.

2. Makoto Yui, Jun Miyazaki, Shunsuke Uemura and Hirokazu Kato: "Efficient XML Storage based on DTM for Read-oriented Workloads", Proc. IEEE International Workshop on Advanced Storage Systems (ADSS 2007), pp.559-564, IEEE CS Press, October 2007.

Other Publications

Domestic Meetings (Reviewed)

1. 油井誠, 宮崎純, 植村俊亮, 加藤博一:“ロックフリー GCLOCK ページ置換アルゴリズム”, Web とデータベースに関するフォーラム (WebDB Forum 2008), 2008 年 12 月.
2. 油井誠, 宮崎純, 植村俊亮: “DTM に基づく XML データベースのための逆経路索引” , 第 18 回データ工学ワークショップ (DEWS2007) 論文集, 電子情報通信学会, 2007 年 2 月.

Domestic Meetings (Non-reviewed)

1. 油井誠, 宮崎純, 植村俊亮, 加藤博一:“遅延評価を利用した並列分散 XML 問合せ処理手法”, 第 19 回 データ工学ワークショップ (DEWS2008) 予稿集, 2008 年 3 月.
2. 油井誠, 宮崎純, 植村俊亮, 加藤博一:“call-by-need 呼出しを利用した分散 XML 問合せ”, データベースと Web 情報システムに関するシンポジウム (DBWeb2007), 2007 年 11 月. (ポスタ)
3. 油井誠, 宮崎純, 植村俊亮, 加藤博一:“Remote Proxy を利用した並列分散 XML 問合せ処理手法の提案”, 電子情報通信学会技術研究報告, Vol.107, No.131, DE2007-22〜114, pp.217-222, 2007 年 7 月.
4. 油井誠, 宮崎純, 植村俊亮: “効率的な XQuery 処理のための DTM に基づく XML ストレージ”, 電子情報通信学会技術研究報告, Vol.196, No.148, DE2006-34, pp.73-78, 2006 年 7 月.
5. 油井誠, 宮崎純, 植村俊亮: “XML コンテンツ統合を支援する XQuery プロセッサ”, 電子情報通信学会第 17 回データ工学ワークショップ (DEWS2006), 2006 年 3 月. (ポスタ)

Awards

1. NAIST 最優秀学生賞, 2009 年 3 月.

2. 情報処理学会データベースシステム研究会学生奨励賞，Web とデータベースに関するフォーラム (WebDB Forum 2008)，2008 年 12 月.

3. WebDB Forum 2008 企業賞 ×3，Web とデータベースに関するフォーラム (WebDB Forum 2008)，2008 年 12 月.

4. 日本データベース学会/電子情報通信学会データ工学研究会/情報処理学会データベースシステム研究会 優秀若手研究者賞, 2008 年 6 月.

5. DBWS2006 学生研究発表奨励賞, 第 140 回データベースシステム研究会 (DBWS2006), 2006 年 7 月.