Implementing A Scalable XML Publish/Subscribe System Using Relational Database Systems

Feng Tian
Department of Computer Science
University of Wisconsin, Madison
Madison, WI, 53706
ftian@cs.wisc.edu

Berthold Reinwald Hamid Pirahesh Tobias Mayr Jussi Myllymaki
IBM Almaden Research Center
650 Harry Road, San Jose, CA, 95120
{reinwald, pirahesh, tmayr, jussi}@almaden.ibm.com

ABSTRACT

An XML publish/subscribe system needs to match many XPath queries (subscriptions) over published XML documents. The performance and scalability of the matching algorithm is essential for the system when the number of XPath subscriptions is large. Earlier solutions to this problem usually built large finite state automata for all the XPath subscriptions in memory. The scalability of this approach is limited by the amount of available physical memory. In this paper, we propose an implementation that uses a relational database as the matching engine. The heavy lifting part of evaluating a large number of subscriptions is done inside a relational database using indices and joins. We described several different implementation strategies and presented a performance evaluation. The system shows very good performance and scalability in our experiments, handling millions of subscriptions with moderate amount of physical memory.

1. INTRODUCTION AND REQUIREMENTS

A publish/subscribe (pub/sub) system receives messages from publishers and notifies subscribers if the messages match the subscriptions. The earliest publish/subscribe systems are topic-based. In these systems, subscribers subscribe to a certain topic (or group, subject etc.) and all messages published on that topic are delivered to the subscribers. More recent pub/sub systems support the content-based paradigm. For a content-based pub/sub system, each subscriber can register a rule in the system. When a publisher publishes a message to the system, the system matches the message with all the registered rules and delivers the message to the corresponding subscribers. Generally speaking, a content-based pub/sub system is more flexible and more powerful than a topic-based pub/sub system.

Content-based pub/sub technology has been widely used in message-oriented middleware systems. Example applications of content-based pub/sub systems include real estate applications, financial information exchange, online auctions, content-based document routing, and data replication (match data changes with replication rules). All major database vendors and middleware system vendors offer pub/sub technology as a standard feature of their business software suits. For example, IBM, Oracle and Microsoft all offer pub/sub applications. Pub/sub in Java Message Service (JMS) [16] is included as part of J2EE [17] and supported by many companies.

All of these systems use SQL or a SQL-like language to express subscription rules, and the message is either a relational tuple or a dictionary data structure with name value pair entries. As XML becomes widely adopted as the standard data exchange format, there is an increasing demand for XML-based pub/sub systems. An XML-based pub/sub system should meet the following requirements. The system should have many publishers and the published XML messages can have very flexible document structures. Subscription rules should be expressed by a powerful language based on XPath [20] or XQuery [21] and support joins of the messages with existing reference data. Subscribers may also want to specify acceptable delay of notifications as a quality of service (QoS) requirement. A pub/sub system for large business applications must be able to maintain a high throughput of messages with millions of subscriptions.

The main technical challenge of implementing a content-based pub/sub system is to efficiently match a published message with many subscriptions. The existing pub/sub approaches can be classified into two major categories depending on the matching strategy. The first strategy treats subscriptions as data that is stored in the database system. When a message is published, the matching of the message with the subscriptions is translated into a join query of the database system. With proper indices built over the subscription data, the join query can be very efficient. The major advantage of this approach is that the scalability is no longer limited by main memory. Also, subscriptions that query reference data stored in the database can be evaluated using the same database engine. The second strategy treats subscriptions as filters. The subscribers are notified when a message passes through the filters. The usual implementation for this strategy is a decision tree or a finite state automaton (FSA). Common computations between filters can often be shared. This strategy can be quite efficient if the decision tree or the FSA fits in memory. However, the scalability is limited by the amount of available physical memory.

Our approach is to use a relational database system to build an XML-based pub/sub system. Such an XML-based pub/sub system proposes many new technical challenges. The XML data has a tree structure and XPath can query both values, and tree structures of the XML documents. Previous implementations of the database-based matching strategy are not applicable because they lack the ability to evaluate XPath against the XML messages. We solve this problem by breaking subscriptions into two parts, an atomic

SIGMOD 2004, June 13–18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 …$5.00.

value predicate matching part and an XML tree structure matching part. The matching of subscriptions with published XML messages is turned into a join query that evaluates both the value predicate part and the tree structure matching part.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the implementation of our system. The cost analysis and several optimizations are introduced in Section 4. Experimental results are given in Section 5. We conclude in Section 6.

2. RELATED WORK

The earliest topic (or subject) based publish/subscribe systems have been studied extensively and there are mature and scalable implementations, for example, [3][11][13]. More recent publish/subscribe systems use a more flexible, and more powerful, content-based paradigm.

Some content-based pub/sub systems treat subscriptions as data and the matching of published messages and subscriptions is evaluated by a join query. Early implementations of this strategy are in the context of group query optimization or continuous query systems. [14] studies how to optimize a group of similar queries that share common computations. TriggerMan [10] uses signatures to group similar queries together in order to provide a scalable trigger mechanism. NiagaraCQ [5] also uses signatures to group similar continuous queries. NiagaraCQ uses a table of constants extracted from the queries along with a join to evaluate a group of queries simultaneously. Recently, commercial database vendors started to use their relational database engine to implement publish/subscribe system [15][19][22]. The scalability of database-based solutions is not limited by the amount of physical memory. However, most of these systems only handle tuple-like messages and the rules are expressed in SQL. Some of the systems can handle XML messages, but the system either uses a wrapper to extract tuple data from XML messages [15] or uses a simple language for the rules [5].

Other systems treat the subscriptions as filters on the messages. For example, [2] constructs an in-memory decision tree for the subscriptions. For each message, the system walks down the decision tree to a leaf node and the subscribers registered at the leaf node are notified. Several XML-based pub/sub research projects that use this "filter" strategy employ a finite state automaton for all the XPath subscriptions in the main memory. A SAX Parser is invoked to parse the published XML messages then the SAX events are streamed through the finite state automata to match the XPath expressions in subscriptions. [1] is the first paper that studies the problem of matching an XML document against many XPath expressions. [1] proposes using an in-memory FSA algorithm as the solution and later [7] introduces state sharing in the FSA construction. [4] builds an index on sub-strings of path expressions that only contain parent-child relationship, and introduces sharing of computations between the common sub-strings. [8][12] use FSA to evaluate XPath expressions over a stream of XML data. [9] proposes building state "lazily" as the solution to tame the exponential explosion of the number of states. In [9], the states of the FSA are treated as a "cache" for matching XPath expressions and the states can be constructed in memory only when they are actually used. This approach is less attractive for a long running system because as more data is processed, sooner or later most states will be constructed in memory. The main disadvantage of a main-memory based approach is that the number of subscriptions in the system is limited by the amount of available physical memory. Also, building a finite state automaton for all subscriptions makes it difficult to add or to delete a subscription, especially when the system is running.

To the best of our knowledge, this paper is the first research that attacks the problems of [1] and [9] using a relational database. By using a database, our solution is not limited by the amount of available physical memory; therefore can handle orders of magnitude more subscriptions. The focus of our work is also slightly different from that of [9]. We assume that our system is long running. Therefore a lazy building approach is not applicable. Also, online insertions/deletions of subscriptions must be handled efficiently for a long running system.

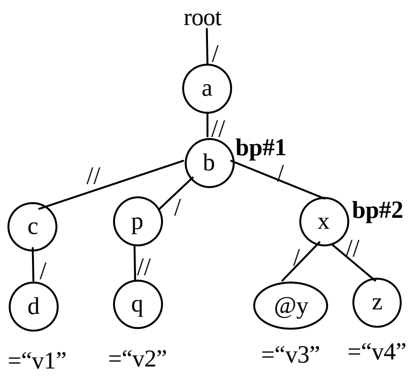
3. XML PUBLISH/SUBSCRIBE USING A RELATIONAL DATABASE

3.1 XML messages and XPath subscriptions

In our system, a subscription ***sub*** is a pair ***(id, xpath)*** where ***sub.id*** is unique across all subscriptions. ***Sub.xpath*** can be expressed by the grammar in Figure 3.1 which implements a subset of XPath [20]. The XPath subset implemented in the paper is usually called a "branching path expression".

```
ConjPath   ::= QPath | OPath AND ConjPath
QPath      ::= Path Op Const
Path       ::= Step | Path Step
                 | Path [ ConjPath ]
Step       ::= Axis tag | Axis * | Axis @ attr
                 | Axis text()
Axis       ::= / | //
Op         ::= < | <= | = | > | >= | !=
```

Figure 3.1 Grammar for the XPath subset considered in the paper



Subscription: id = 5

XPath: /a//b[//c/d/text()=”v1” AND /p//q/text()=”v2”]
/x[@y=”v3”]//z/text() = “v4”

Figure 3.2 Example subscription and its graph representation

Figure 3.2 shows a graph representation of an XPath example with four branches. Each node in the graph is an XML tag in the XPath and each edge shows the axis (slash or double slash) of each step. We assign a number to each branching point of the XPath. In this example, node *b* and *x* are assigned branching point number *bp#1* and *bp#2* respectively.

The subset of XPath implemented by the grammar in Figure 3.1 does not include the Boolean OR expression. An XPath with Boolean OR expression must first be rewritten to a disjunctive normal form and each disjunction is submitted as a separate subscription. The constants of the predicate are taken from a fixed, ordered domain. Our implementation provides support for both string and numerical values.

A publication is an XML message ***xmsg(id, data)*** where ***xmsg.id*** is unique across all published messages and ***xmsg.data*** is a valid XML document. The matching algorithm matches each published XML message with all the subscriptions in the system and outputs notifications in the form of ***(xmsg.id, sub.id)***.

For simplicity, the subscription language in Figure 3.1 does not allow access to the reference data in the database. In practice, we can join an XML message ***m*** with the reference data to produce a “enriched” message ***m’***, which is used to match against the XPath in the subscription.

3.2 Relational representation of the XPath subscriptions

When a subscription is added to the system, the XPath subscription is rewritten as a conjunction of predicates. Each predicate contains one branch in the graph presentation of the XPath. The path of the tag and axis leading to the leaf of the branch is called a *linear path*. Each linear path is annotated with branching point numbers in order to retain the branching information of the XPath expression along the linear path. A predicate that checks all the branching points in each linear path of the XPath match is also added to the conjunction. The predicate that checks the matching of branching points is referred to as ***BrPred*** later in the paper. For example, the XPath is Figure 3.2 can be rewritten as a conjunction of four branch predicates and the *BrPred* predicate, as shown in Figure 3.3.

/a//b{bp#1}//c/d/text() = “v1” AND

/a//b{bp#1}/p//q/text() = “v2” AND

/a//b{bp#1}/x{bp#2}/@y = “v3” AND

/a//b{bp#1}/x{bp#2}//z/text() = “v4” AND

BrPred (i.e. bp#1 appeared above matches the same element in the published XML messages, similarly for bp#2.)

Figure 3.3 Rewrite XPath to conjunction of branches

Each branch in the rewritten form of the XPath expression contains three types of information: the linear path without branch, the atomic value predicate, and the branching information. For example, the first predicate in Figure 3.3 has a linear path */a//b//c/d/text()*, an atomic value predicate *=“v1”* and has the branching point number 1 at the second step (the *b* node) along the linear path. These three types of information are stored in tables in the relational database. Figure 3.4 shows the schema of the tables and how the example subscription is stored.

Each linear path is stored as a tuple in the *Query_Linear_Path* table. The whole linear path in text format is stored in the *LinearPath* column and is used as the primary key of the table. The system also generates a unique id for the linear path. We break each linear path into three pieces at the first and the last double slash so that only the middle part may contain additional double slashes. The three pieces are stored in the *prefix*, *middle*, and *postfix* columns, respectively. The *prefix*, *postfix* and *middle* columns are used to match tag paths in published XML messages against the linear paths. The order of the steps in the *postfix* column is reversed for efficient index access during the match. The details of the path matching algorithm are deferred to Section 3.4.

Query_Linear_Path table

LinearPath	**Id**	**Prefix**	**Postfix**	**Middle**
/a//b//c/d/text()	100	/a	text()/d/c	b
/a//b/p//q/text()	101	/a	text()/q	b/p
/a//b/x/@y	102	/a	@y/x/b	null
/a//b/x//z/text()	103	/a	text()/z	b/x

Query_Predicate table

pre_id	**LinearPath_Id**	**Op**	**Value**	**Chain_id**	**final**
0	100	=	“v1”	501	F
501	101	=	“v2”	502	F
502	102	=	“v3”	503	F
503	103	=	“v4”	5	T

LinearPath_BrInfo table

LinearPath	**Linear Path_Id**	**BrInfo**	**BrInfo _id**
/a//b//c/d/text()	100	“bp#1=2”	50
/a//b/p//q/text()	101	“bp#1=2”	51
/a//b/x/@y	102	“bp#1=2&bp#2=3”	52
/a//b/x//z/text()	103	“bp#1=2&bp#2=3”	53

Query_BranchPoint table

Sub_id	**BranchPoint_Info**
5	“50&51&52&53”

The atomic value predicates of the conjunction are re-ordered so that the more selective predicates are pulled up to the front. Currently we pull the equal predicates in front of the non-equal ones. Each value predicate in the XPath, (*linearpath_id*, *op*, *value*) is stored in corresponding columns of the *Query_Predicate* table. Predicates in the same subscription are chained using *chain_id* and *pre_id* of the *Query_Predicate* table, starting with the *pre_id* of the first predicate set to 0. In our example, starting with *pre_id* equals 0, predicate *(linear path with id 100, =, "v1")* evaluated to true leads to *chain_id* 501. *Pre_id* 501 plus *(linear path with id 101,=, "v2")* evaluated to true leads to *chain_id* 502 and so on. A true value in the *final* column of the *Query_Predicate* table indicates that there are no more entries in the chain and that the subscription is true subject to the evaluation of the *BrPred* predicate. The subscription id of the subscription is stored in the *chain_id* column of the final predicate.

The branching information of each linear path is packed as a binary varchar and stored in the *BrInfo* column of the *BranchPoint_Info* table. For example, the third tuple in the *BranchPoint_Info* table has the *BPInfo* column set to *"bp#1=2&bp#2=3"* and *LinearPath* column set to*"/a//b/x/@y"*, which indicates that branching point number 1 is at the second step (the *b* node) and branching point number 2 is at the third step (the *x* node) of the linear path. The id of the *LinearPath*, which is generated in the *Query_Linear_Path* table, is also included in the table for fast index lookup. For each *(linearpath, brinfo)* pair, the system generates a unique id and store the id in the *LinearPath_Brinfo* table. The *BranchPoint_Info* table is used to check the *BrPred* predicates for the subscriptions. We will discuss the algorithm for checking *BrPred* in Section 3.5.

The branching information of a subscription is stored separately in the *Query_BranchPoint* table. Table *LinearPath_BrInfo* maps the branching information of each branch of the XPath subscription to a *BrInfo_id*. The *BrInfo_id* of all the branches of the XPath expression is packed into a binary varchar and stored in the *BranchPoint_Info* column of the *Query_Branchpoint* table. In our example, *"50&51&52&53"* identifies four tuples in *LinearPath_BrInfo* table that represent the branching information of the four branches of the example subscription, whose subscription id is 5.

Deleting a subscription online is as easy as deleting all the chained predicates in the predicate table, starting from the last predicate of the conjunction, whose *chain_id* is actually the subscription id and for which the value of the *final* column is true. The tuple in the *Query_BranchPoint* table also needs to be deleted. The entries in the *Query_Linear_Path* table and the *BranchPoint_Info* table that are related to the subscription, however, cannot be deleted unless the linear paths or branching point information are no longer used by any other subscriptions. This is not a problem because the two tables are much smaller than the *Query_Predicate* table and some obsolete tuples in these two tables will not significantly degrade the performance of the system. Periodical garbage collection can be performed using a simple SQL delete statement.

3.3 Publishing XML messages

We use the example XML message in Figure 3.5 to demonstrate how the system handles published XML messages. The XML document is parsed by a SAX parser and a node id is assigned for each element, attribute or text data during parsing. The assigned id is shown in the brackets in Figure 3.5.

```
Message: id = 12
XML: <a (1)><b (2)>
            <c (3)><d (4)>v1 (5)</d></c>
           <p (6)><p2 (7)><q (8)>v2 (9)</q></p2></p>
           <x (10) @y="v3" (11)>
                  <z (12)>v4 (13)</z>
                  <z (14)>v5 (15)</z>
           </x>
       </b>
       <b (16)><b (17)><x (18) @y="v4" (19)/></b></b>
</a>
```

Figure 3.5 Example publication

For each *text()* value or attribute value in the published XML document, the SAX parser generates a triple (*tagpath, idpath, value*), where the *tagpath* and *idpath* are the tag names and assigned node ids along the path from the document root to the value. We use the single slash to separate tags in the *tagpath* to match the notation of the linear path of the XPath. One should note that a *tagpath* cannot contain any double slashes. The XML element nesting (tree structure) information is encoded in the *idpath* of the triple. Figure 3.6 shows the result of parsing XML data in Figure 3.5.

TagPath	IdPath	Value
/a/b/c/d/text()	1.2.3.4.5	"v1"
/a/b/p/p2/q/text()	1.2.6.7.8.9	"v2"
/a/b/x/@y	1.2.10.11	"v3"
/a/b/x/z/text()	1.2.10.12.13	"v4"
/a/b/x/z/text()	1.2.10.14.15	"v5"
/a/b/b/x/@y	1.16.17.18.19	"v4"

Figure 3.6 SAX parsing result

The system maintains a *Tagpath_Map* table that maps each *TagPath* to a *TagPath_id*. The table acts as a cache of all the tag paths that the system has encountered. During SAX parsing, if the system sees a tag path that is not in the table yet, it inserts the tag path into the table and automatically generates a unique id. We also reverse the tag names of each tag path and store the reversed *TagPath* in the *Inv_Tagpath* column of the table. Both the *TagPath* and the *Inv_Tagpath* columns are used to match linear paths in the subscriptions. The details of the path matching algorithm are deferred to Section 3.4. The SAX parsing results of the XML message are stored in a global temporary table of the relational database in order to save logging overhead and reduce the cost of moving data into the database. Figure 3.7 shows the *Tagpath_Map* table and the temporary table containing parsing result of the example XML message.

TagPath_Map table

TagPath	Inv_Tagpath	TagPath_Id
/a/b/c/d/text()	text()/d/c/b/a	311
/a/b/p/p2/q/text()	text()/p2/p/b/a	312
/a/b/x/@y	@y/x/b/a	313
/a/b/x/z/text()	text()/z/x/b/a	314
/a/b/b/x/@y	@y/x/b/b/a	315

Temporary SAX_Parse_Result table

Event_id	Tagpath_Id	IdPath	Value
12	311	1.2.3.4.5	“v1”
12	312	1.2.6.7.8.9	“v2”
12	313	1.2.10.11	“v3”
12	314	1.2.10.12.13	“v4”
12	314	1.2.10.14.15	“v5”
12	315	1.16.17.18.19	“v4”

Figure 3.7 SAX parsing result as stored in the database

Next the linear paths of the subscriptions must be matched with the tag paths in the messages. The matching linear path and tag path are stored in the *PathMatch* table, which is shown in Figure 3.8. How to compute and maintain the *PathMatch* table is described in detail later in Section 3.4. Each tuple in the table indicates a match of a tag path with a linear path. For example, tuple (312, 101) means that the tag path with id 312 matches the linear path with id 101, that is, tag path */a/b/p/p2/q/text()* matches linear path */a/b/p//q/text()*.

PathMatch table

TagPath_Id	LinearPath_Id
311	100
312	101
313	102
314	103
315	102

Figure 3.8 The PathMatch table

With the *PathMatch* table, we can run a recursive SQL query to match published XML messages with subscriptions. We do not show the SQL statement here because the SQL statement is rather long and complex. Figure 3.9 shows a graphical representation of the recursive query plan. Matching tag paths with query linear paths and match atomic values in XML message with value predicates are encoded as join predicates. Value predicates from the same subscription are chained by the *chain_id* column with the *pre_id* column of the *Query_Predicate* table. Each iteration of the recursion evaluates one more value predicate of a subscription

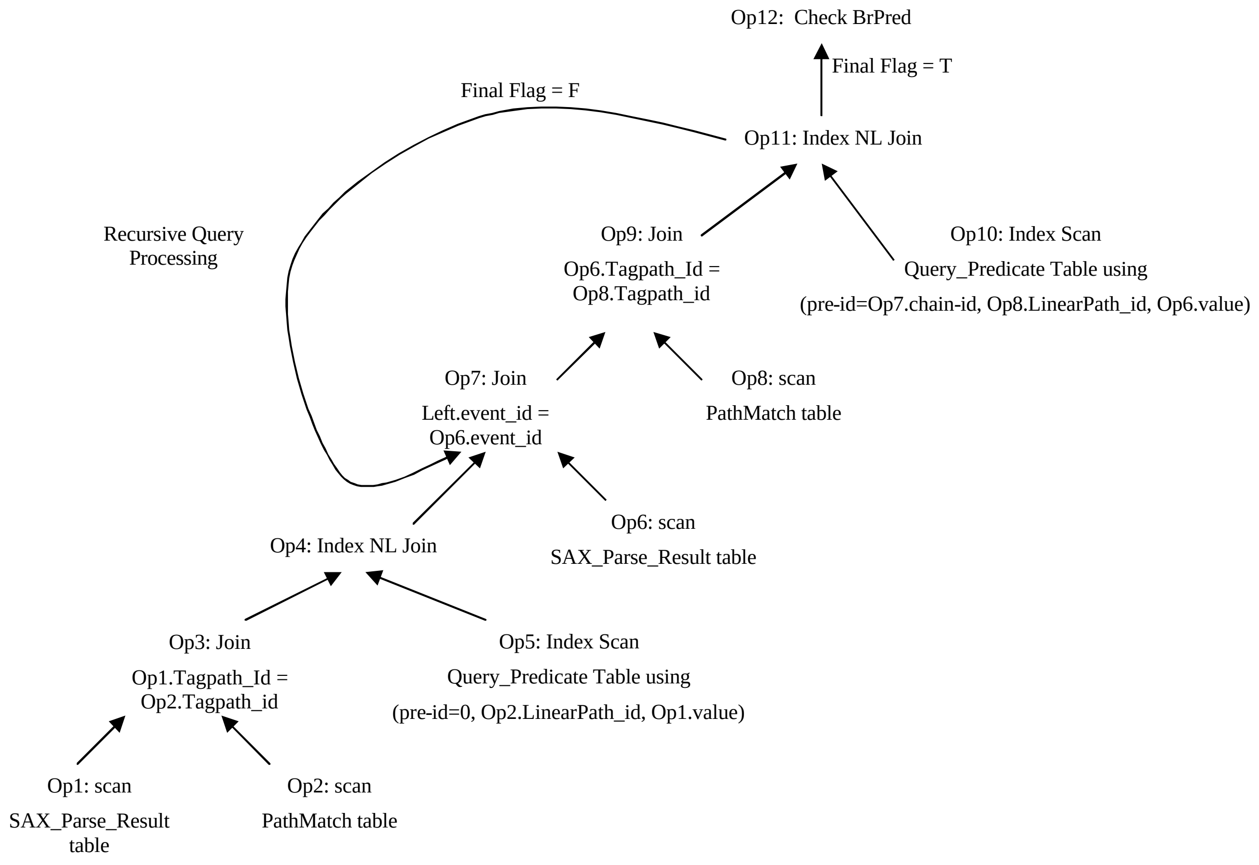

Op12: Check BrPred
Final Flag = T
Final Flag = F
Op11: Index NL Join
Recursive Query Processing
Op9: Join
Op6.Tagpath_Id = Op8.Tagpath_id
Op10: Index Scan
Query_Predicate Table using
(pre-id=Op7.chain-id, Op8.LinearPath_id, Op6.value)
Op7: Join
Left.event_id = Op6.event_id
Op8: scan
PathMatch table
Op6: scan
SAX_Parse_Result table
Op4: Index NL Join
Op3: Join
Op1.Tagpath_Id = Op2.Tagpath_id
Op5: Index Scan
Query_Predicate Table using
(pre-id=0, Op2.LinearPath_id, Op1.value)
Op1: scan
SAX_Parse_Result table
Op2: scan
PathMatch table

Figure 3.9 Query plan for matching messages with subscriptions

until the evaluation reaches a predicate whose *final* flag is set to true. The *TagPath_Id* and the *Idpath* from the temporary *SAX_Parse_Result* table is gathered during the recursive processing and packed into a binary varchar. This binary varchar is used as the input to a UDF to evaluate the *BrPred* predicate. The implementation of the UDF is discussed in detail in Section 3.5. The SQL query can be run either in a per-message mode or in a batch mode to save query invocation cost. The temporary *SAX_Parse_Result* table is cleared between the query invocations (declared as "on commit delete rows" in the SQL definition statement of the temporary table).

As discussed in Section 3.2, the equal predicates of a subscription are pulled above the non-equal ones. In practice, the SQL statement is written as two stages. Each stage has a recursive plan similar to that in Figure 3.9. In the first stage, only equal predicates are evaluated. The results from the first stage are further processed by the second stage which evaluates the non-equal predicates. This two-stage processing has better performance and in Section 4, we will show that two-stage processing has desirable properties when common computations from different subscriptions can be shared.

Using the join plan of Figure 3.9 to evaluate XPath subscriptions may produce duplicates of the notifications due to the tree structure of the XML messages. If the semantic of the pub/sub system requires unique notification per matched message subscription pair, we can add a distinct operator on top of the plan in Figure 3.9.

3.4 Computing and maintaining the PathMatch table

This section describes how to compute and maintain the *PathMatch* table used in the recursive query plan of Section 3.3.

First, let us consider the situation that a new query linear path is added to the system. The system adds the query linear path into the *Query_Linear_Path* table and generates a unique id for it. The linear path is broken into three pieces at the first and last double slash so that only the middle part may contain additional double slashes. Then, the system uses the index on the *TagPath* column and the index on the *Inv_Tagpath* column of the *Tagpath_Map* table to look for tuples in the *Tagpath_Map* table that match both the prefix and the postfix of the linear path. Notice that the order of tags in the *postfix* column of the *Query_Linear_Path* table is reversed. The postfix matching is turned into a character string prefix matching so that an index can be used. It is still necessary to check that the tag paths found by the index lookup really match the middle part of the linear paths. This checking is done by a regular expression matching algorithm using a state machine. Once a match is found, it is entered into the *PathMatch* table.

The process is symmetric when the system encounters a new tag path. The system uses the tag path and the reversed tag path (as stored in the *Inv_Tagpath* column of the *Tagpath_Map* table) to do index lookups to find records in the *Query_Linear_Path* table with matching *prefix* and *postfix* columns. Then a regular expression matching on the middle part is performed. The matched results are also stored in the *PathMatch* table.

We can consider the *PathMatch* table as a cache for matched tag paths and query linear paths. Usually, the number of different query linear paths is much smaller than the number of subscriptions. However, the cost of adding a new tag path is still high. Fortunately the tag paths are also shared among the published XML messages. Caching the matched results of tag paths with linear paths is especially attractive when the publishers use one or more XML schemas (or DTDs) for XML messages. When a limited number of schemas for the messages are used, the tag paths are highly repetitive and most of the tag paths will be cached (except for the very first few messages).

Tuples in the *Query_Linear_Path* table and the *Tagpath_Map* table can be deleted. Deleting a subscription may result in the deletion of linear paths. It may also be desirable to clean entries in *Tagpath_Map* table, for example, schema changes may guarantee that some tag paths will never appear in the future messages. Deletions to the *Query_Linear_Path* table and *Tagpath_Map* table trigger the deletions to the *PathMatch* table.

3.5 Checking the *BrPred* predicate

The recursive query processing of the plan in Figure 3.9 evaluates all the atomic value predicates of the subscriptions. The XML tree structure matching part of the subscription is expressed by the *BrPred* predicate, which checks all the branching points in the XPath matches. In this section, we describe how to check the branching predicate *BrPred* for each subscription. First, we introduce the *TagPath_BrInfo* table and describe how this table is computed and maintained.

3.5.1 The TagPath_BrInfo table

For each linear path predicate of a subscription, the *LinearPath_BrInfo* table records the branching point positions in the linear path. In order to find the XML node id of the branching point, we need to translate the branching point positions in the linear path to the branching point positions of the tag path. The translation results are stored in the *TagPath_BrInfo* table. The *TagPath_BrInfo* table can be considered as a materialized view defined by the SQL in Figure 3.10, where *bp_pos_udf* is a UDF that returns a binary string.

```
SELECT tp.Tagpath_id, lbr.Brinfo_Id,
            bp_pos_udf(tp.Tagpath, lbr.BrInfo)
FROM Tagpath_Map tp, LinearPath_BrInfo lbr, PathMatch
pm
WHERE tp.Tagpath_id = pm.Tagpath_id AND
```

Figure 3.10 Definition of TagPath_Brinfo table

TagPath_BrInfo table

TagPath_Id	BrInfo_Id	BP_Position
311	50	"{bp#1}, {2}"
312	51	"{bp#1}, {2}"
313	52	"{bp#1,bp#2},{2,3}"
314	53	"{bp#1,bp#2},{2,3}"
315	52	"{bp#1,bp#2},{2,4},{3,4}"

Figure 3.11 The TagPath_BrInfo table

The content of the *TagPath_BrInfo* table with our example subscription and example message from the previous sections are shown in Figure 3.11. We will explain the meaning of the binary string stored in the *BP_Position* column. The implementation of *bp_pos_udf* is straightforward once the meaning of the binary string is clear.

The *BrInfo_Id* column of the *TagPath_BrInfo* table is a foreign key that refers to the *Brinfo_Id* column of the *LinearPath_BrInfo* table. Each binary string stored in the *BP_Position* column of the *TagPath_BrInfo* table is a sequence of comma separated arrays. The first array in the sequence contains the branching point numbers that occur in the branching information identified by the *BrInfo_Id* column. The rest of the arrays in the sequence, which are called the position arrays, represent branching point positions in the tag paths that is identified by the *TagPath_Id* column. For example, the third tuple *(313, 52, "{bp#1,bp#2},{2,3}")* of the *TagPath_Brinfo* table indicates the followings:

1. *TagPath* with id 312, that is, */a/b/x/@y*, matches the linear path associated with *Brinfo* 52, that is, */a//b/x/@y*.
2. *BrInfo* with id 52 has two branch points, *bp#1* and *bp#2*, as indicated by the first array in the sequence.
3. The position array *{2,3}* indicates that *bp#1* of *BrInfo* 52 is the second node of the tag path 312, that is, the *b* node of */a/b/x/@y*, and *bp#2* is the third node, that is, the *x* node of */a/b/x/@y*.

It is worth pointing out that there may be several position arrays in a *BP_Postion* string due to the double slashes in a linear path. Tuple *(315, 52, "{bp#1,bp#2},{2,4},{3,4}")* is such an example. In this case, branch point *bp#1* is the *b* node in the linear path */a//b/x/@y*. This branching point may evaluate to either the first *b* node or the second *b* node in the tag path */a/b/b/x/@y*.

Like the *PathMatch* table, the *TagPath_BrInfo* table is maintained incrementally. The updates to the *PathMatch* table or the *LinearPath_BrInfo* table are propagated to the *TagPath_BrInfo* table immediately. Our implementation of the *bp_pos_udf* UDF is simplistic by using brutal force to enumerate all possible position arrays. The inputs to the UDF are usually very short strings because the depth of an XML document and the length of a linear path usually are small; therefore a simple algorithm actually performs very well. Also, the updates to the *TagPath_BrInfo* table happen only when a new subscription is added or a new tag path is encountered. The update is not in the recursive query that evaluates the matching of messages with subscriptions. Therefore, it is not performance critical. Deletions to the *PathMatch* table or the *LinearPath_Brinfo* table are also cascaded to the *TagPath_BrInfo* table by triggers.

3.5.2 Evaluating the BrPred predicate

The output of the recursive query processing of the plan in Figure 3.9 must be further checked against the *BrPred* predicate (Op12 in Figure 3.9). During iterations of the recursion, the *TagPath_Id, IdPath* from the *SAX_Parse_Result* table are packed into a binary varchar. All the binary varchars from the iterations are concatenated to form the input to a UDF *brpred_udf*, which returns a Boolean value indicating whether all the branching points in a subscription match. The input to Op12 in Figure 3.9 using our example will contain two tuples, which are shown in Figure 3.12.

Event_Id	**Sub_Id**	**(TagPath_id, idpath)**
12	5	(311,"1.2.3.4.5"), (312,"1.2.6.7.8.9"), (313,"1.2.10.11"), (314,"1.2.10.12.13")
12	5	(311,"1.2.3.4.5"), (312,"1.2.6.7.8.9"), (315,"1.16.17.18.19"), (314,"1.2.10.12.13")

Figure 3.12 Input to Op12 (BrPred_udf)

The *brpred_udf* first uses the subscription id to find the *BranchPoint_Info* from the *Query_BranchPoint* table. In our example, the subscription with id 5 has a *BranchPoint_Info* of value *"50&51&52&53"*. Both the *BranchPoint_Info* and the binary varchar from the recursive query processing are unpacked into array of triples (*Tagpath_Id, Brinfo_Id, idpath*). Next, the first two entries, (*Tagpath_Id* , *Brinfo_id*) pairs in the triple are converted to the *BP_Position* string using the *TagPath_BrInfo* table from Figure 3.11. The conversion results are shown in Figure 3.13.

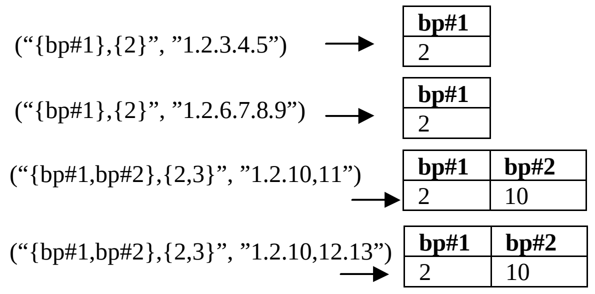
Event_ Id	**Sub_ Id**	**(BP_Position, idpath)**
12	5	("{bp#1},{2}", "1.2.3.4.5"), ("{bp#1},{2}", "1.2.6.7.8.9"), ("{bp#1,bp#2}{2,3}","1.2.10.11"), ("{bp#1,bp#2}{2,3}","1.2.10.12.13")
12	5	("{bp#1},{2}", "1.2.3.4.5"), ("{bp#1},{2}", "1.2.6.7.8.9"), ("{bp#1,bp#2}{2,4}{3,4}", "1.16.17.18.19"), ("{bp#1,bp#2}{2,3}","1.2.10.12.13")

Figure 3.13 Convert TagPath_Id, BrInfo_Id to

Remember that the *BP_Positon* string "{bp#1},{2}" simply means branching point number 1 is at the second level of the associated *IdPath* starting from the XML document root. Therefore, we can transform each *(BP_Postion, idpath)* pair into a relation that has the branching point numbers as column names and XML node ids as tuples in the relation. Figure 3.14 shows the result of converting *(BP_Position, idpath)* into relations for the first tuple from the table in Figure 3.13.

The conversion for the second tuple in Figure 3.13 is similar except for the pair *("{bp#1,bp#2}{2,4}{3,4}", "1.16.17.18.19")*, there are two tuples *(16,18)* and *(17, 18)* in the result relation.

Now we can match the branching points from different linear paths by doing a natural join (equal join on columns with same column name) of the converted relations. If the result of the natural join is not empty, the branching points from each of the different linear paths match. Therefore, the *BrPred* predicate is true. Otherwise, the *BrPred* predicate is false. In our example, the natural join result for the first tuple in Figure 3.13 contains a tuple (2, 10) and the natural join result for the second tuple is empty.



("{bp#1},{2}", "1.2.3.4.5")
bp#1
2
("{bp#1},{2}", "1.2.6.7.8.9")
bp#1
2
("{bp#1,bp#2},{2,3}", "1.2.10,11")
bp#1 bp#2
2 10
("{bp#1,bp#2},{2,3}", "1.2.10,12.13")
bp#1 bp#2
2 10

Figure 3.14 Converting *(BP_Position, idpath)* to relations

Even though the branching point matching algorithm described here has a strong connection with relational algebra, the conversion results of the *(BP_Position, idpath)* pairs are not stored in tables and the natural join is not performed by the database engine. We implement the algorithm in *brpred_udf* using C for efficiency and convenience reasons.

4. OPTIMIZATIONS

In this section, we will describe a simple cost metric for the recursive processing in Figure 3.9 and introduce several optimization techniques.

4.1 A cost metric for the recursive plan

The Op5 and Op10 from the recursive query plan in Figure 3.9 use an index on the (*pre-id, LinearPath_id, Op, value*) columns of the *Query_Predicate* table. Since the index lookups dominate the cost of the query plan, the cost of the plan is measured by the number of index accesses and the number of tuples retrieved by the lookup.

Table 4.1 shows the parameters used in the cost metric. For simplicity, we do not distinguish the cost of the index lookups in the equal or non-equal stage of the two stage processing. Also, we let the selectivity *s* in Table 4.1 be the average selectivity of the two stages.

C_i	The cost of one index access to the *Query_Predicate* table
C_t	The cost of retrieve one tuple from the *Query_Predicate* table
N	The number of subscriptions
b	The average number of branches per subscription
m	The average number of atomic values in an XML message
p	The average number of matching query linear paths per tag path
s	Selectivity of a linear path with an XML atomic value
R_i	Number of tuples retrieved in iteration i
$Cost_i$	The cost of the i-th iteration in the recursive plan

Table 4.1 Parameters in the cost metric

In the first iteration, an index probe is performed for each atomic value. There are exactly *N* linear paths with *pre-id* equals 0. Therefore, this iteration retrieves $R_1 = spmN$ tuples.

$$Cost_1 = pmC_i + R_1 C_t = mpC_i + smpNC_t$$

On average a subscription has *b* linear paths which implies the probability that the *final* flag of a tuple in the *Query_Predicate* table is false is *(b-1)/b*. Tuples with *final* flag set to true do not enter the second iteration, therefore, in the second iteration, the number of index access is $R_1 * (b-1)/b * p * m$. Starting from the second iteration, the *chain-id* from the previous iteration is used to chain with the *pre-id* of the second iteration. We have

$$R_2 = spm(b-1)/b\ R_1$$

$$Cost_2 = pmR_1(b-1)/bC_i + R_2 C_t$$

Similarly, we have

$$R_n = spm(b-1)/bR_{n-1}$$

$$Cost_n = pmR_{n-1}(b-1)/bC_i + R_{n-1}C_t$$

The total cost of the plan is

$$Total\_Cost = ?_{i=0,1,\ldots\ max\ iterations} Cost_i$$
$$= pmCi$$
$$+ N\ spm/(1-spm(b-1)/b)\ (pm\ (b-1)/b\ Ci + Ct)$$

From the formula, we see the total cost scales linearly with respect to the number of subscription *N*. In the following subsections, we introduce several optimization techniques.

4.2 Sharing common predicates in the recursive plan

The relational representation of the subscriptions described in Section 3.2 does not exploit sharing of computations between linear path predicates from different subscriptions. Two linear path predicates from different subscriptions are stored in the *Query_Predicate* table separately even if they have exactly the same linear path, Boolean operator, and value. We implemented a simple sharing optimization for linear path predicates whose *final* flags are false. Before a linear path predicate with false *final* flag is added to the *Query_Predicate* table, we check if there already is a non-final predicate in the table with same *pre-id, linear_path_id,* Boolean *op,* and *value.* A predicate is added to the *Query_Predicate* table only when a match cannot be found. The strategy we used here is simple and does not consider any global optimization strategies. Sharing common predicates between subscriptions raises many interesting optimization problems. For example, the order of the linear path predicates decides whether a predicate can be shared between predicates. If a predicate is very common, it may be desirable to pull this predicate to front to maximize sharing even though the predicate is not very selective. We refer to [6][18] for studies of the cost models and global optimization strategies in similar situations.

The simple sharing optimization strategy used is especially attractive in the first stage (equal stage) of the two stage recursive processing. Each index probe in the first stage will produce at most one tuple with a false *final* flag. Therefore, the number of index probes at first stage is bounded by $m^{number\ of\ max\ iterations}$ regardless of the number of subscriptions *N*. This number can be much smaller than that of the non-sharing case for small or medium size XML messages.

The simple sharing strategy is less effective in the second (non equal) stage of the two stage processing. Assume that many subscriptions have similar non-final predicates of the form *"/a/text() > x_i AND P_i"*, where x_i is a constant value and P_i is the conjunction of the remaining non-equal predicates in this subscription. These similar predicates are not shared because the value x_i is different and the index lookup on the *Query_Predicate* table in each iteration of the recursive plan may produce many output tuples. We propose a strategy that partitions the x_i values in these predicate in to *n* buckets, *[b0, b1), [b1, b2), … [bn-1, bn)*. Then the subscription can be rewritten to *"/a/text() > bk AND P_i AND /a/text() > x_i"* if x_i fall in bucket *[bk, bk+1)*. Subscriptions that fall into the same bucket can share the first iteration and the number of output tuples from the first iteration is significantly reduced. Notice that even though each subscription has one more predicate, in the worst situation that all the remaining predicates P_i evaluate to true, the overhead is only *n* additional index lookups where *n* is the number of the buckets. The total number of index accesses to the *Query_Predicate* table will be reduced if the remaining predicates P_i are selective. Our experience suggests that the number of buckets *n* should be a small value, for example, around 20.

4.3 Unrolling the predicate table

Another way to reduce the number of index lookups to the *Query_Predicate* table is to unroll the recursive processing and store *k* linear path predicates in one row of the unrolled table. For example, if *k=4*, that is, we unroll four levels of recursion, the *Query_Predicate* table in Figure 3.4 can be stored as one tuple in the *Predicate_Unroll* table in Figure 4.1

Query_Predicate table

pre-id	Linear-Path	Op	Value	Chain-id	final
0	100	=	"v1"	501	F
501	101	=	"v2"	502	F
502	102	=	"v3"	503	F
503	103	=	"v4"	5	T

Predicate_Unroll table

SubId	Path1	Op1	Value1	Path2	Op2	Vaule2
5	100	=	"v1"	101	=	"v2"

Predicate_Unroll table continued

Path3	Op3	Value3	Path4	Op4	Value4	Final
102	=	"v3"	103	=	"v4"	T

Figure 4.1 Unroll the Query_Predicate table

For subscriptions with fewer than *k* linear predicates, the remaining columns in the unrolled table are filled with dummy values that always evaluate to true. For subscriptions with more than *k* linear predicates, the *final* flags in the unrolled table are set to false and the evaluation of the remaining linear path predicates uses the recursive processing technique.

Depending on statistics such as the size of unrolled table and the size of published message, the database may choose a query plan that does a self cross product of the XML *Sax_Parse_Result* table before performing the index lookup on the unrolled table. For example, Figure 4.2 shows a plan that does the self cross product once.

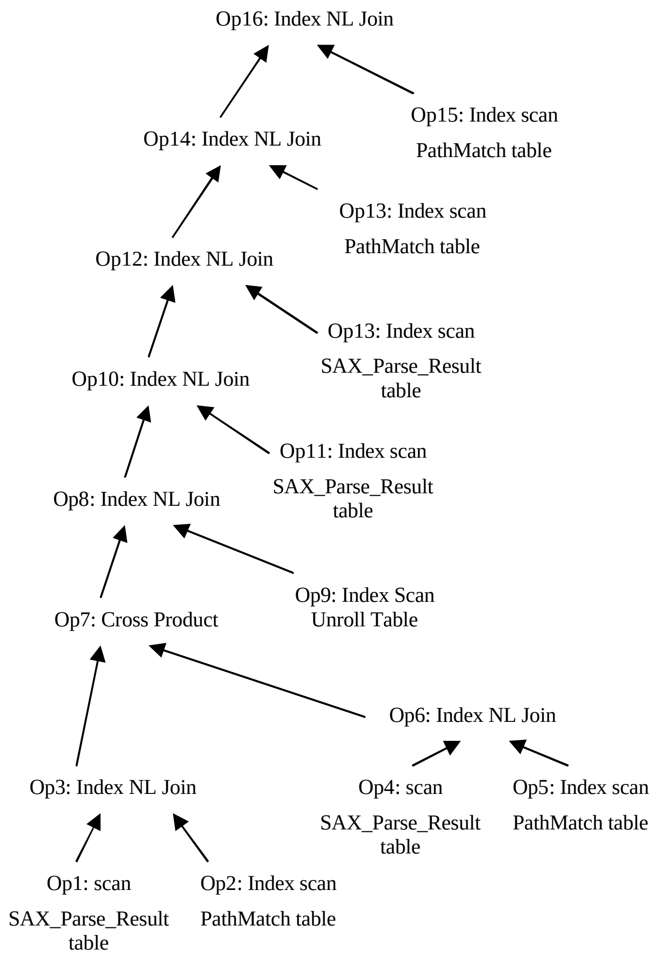

Op16: Index NL Join
Op14: Index NL Join
Op15: Index scan PathMatch table
Op12: Index NL Join
Op13: Index scan PathMatch table
Op10: Index NL Join
Op13: Index scan SAX_Parse_Result table
Op8: Index NL Join
Op11: Index scan SAX_Parse_Result table
Op7: Cross Product
Op9: Index Scan Unroll Table
Op6: Index NL Join
Op3: Index NL Join
Op4: scan SAX_Parse_Result table
Op5: Index scan PathMatch table
Op1: scan SAX_Parse_Result table
Op2: Index scan PathMatch table

Figure 4.2 Query plan after unrolling

Next we will estimate of the execution cost of the unrolled plan. Suppose we unroll the evaluation *k* levels and the database chooses to do a self cross product once. The record size of the unrolled table is approximately *k* times as large as the record size of the original *Query_Predicate* table therefore we assume the cost of retrieving one record from the unrolled table is kC_t. The number of index lookups to the unrolled table is p^2m^2. The tuples retrieved from the index lookups on unrolled table are used to probe the *Sax_Parse_Result* table to evaluate other predicates. We denote the cost of evaluate the predicates after the second level is C_r, which is usually much smaller than C_i because the *Sax_Parse_Result* table is small enough to fit in memory. For the subscriptions with more than k predicates, the plan reverts to recursive processing and more accesses to the large *Query_Predicat* table are required. We denote this cost as C_x. In practice, C_x is small for a reasonable k (5 or 6) because:

1. The number of complex subscriptions with many predicates is usually small.
2. The size of the *Query_Predicate* table is much smaller than before because most predicates are stored in the unrolled table.
3. The k most selective predicates have been pulled up to the front.

The cost of the unrolled plan is

$$Total\_Cost = p^2m^2C_i + s^2p^2m^2N(kC_t + C_r) + C_x$$

The cost formula shows that the main advantage of unrolling the *Query_Predicate* table is that the number of index lookups to the unrolled table is not related to the total number of subscriptions. Also, the database optimizer now can choose how many self cross products to use before doing the index lookups on the unrolled table. Furthermore, the plan is not recursive and a non-recursive plan is usually more efficient and more scalable. The main disadvantage is that the opportunity of sharing computation across subscriptions is lost. Given that the most expensive operation is the index lookup on the query predicate table or unrolled table, unrolling is worthwhile in most situations unless the linear path predicates are very selective and there are lots of common predicates across different subscriptions.

5. EXPERIMENTAL RESULTS

We evaluate the performance and scalability of our system using two sets of experiments. The first set uses a synthetic XML dataset with a simple flat structure. This workload is representative for publishers that publish relational tuples in XML format. The second set of experiments uses a real XML dataset of a semi-structured nature. We always enable sharing common linear path predicates between subscriptions for the recursive plan because the analysis in Section 4 shows sharing common predicate is always desirable.

5.1 Experiment 1: Simple Flat XML Dataset

A synthetic stock information dataset is used in the first set of experiments. Each XML message contains information, such as symbol, price, of one stock. The linear paths in the subscriptions are also simple (without double slash or wild char). Figure 5.1 shows an example of an XML message and a subscription.

```
Message: id 100
<?xml?><stock>
  <symbol>YHOO</symbol><price>70.2</price>
  <open>50</open><change>20.2</change>
  <low>47.2</low><high>74.5</high>
  <volume>10000</volume>
</stock>

Subscription id: 345
/stock/symbol/text() = "YHOO" AND
/stock/price/text() > 50 AND /stock/open/text() < 38 AND
/stock/high/text() > 80 AND /stock/volume/text() > 20000
```

Figure 5.1 Example XML stock data and subscription

We run the experiments on a 1.4 GHz Pentium IV with 512 M of memory running Windows XP. The relational database we used is DB2 V 8.1. Table 5.1 shows the running time of matching 10,000 messages with one million subscriptions. Each subscription has one equal predicate on stock symbol and four non-equal predicates on other attributes giving a total of 5 million predicates. The analysis in Section 4 shows that for the recursive plan, the selectivity of the linear path is important to determine the number of index lookups for the next iteration of the recursive processing. Several experiments were conducted using subscriptions with different selectivities. The performance results are shown in Table 5.1. The running times reported here include the time to parse the XML data and to insert the parse results into the temporary table.

Number of Notifications	**Unroll**	**Share**	**Share-Bucket**
1,000	56	107	26.3
10,000	61	154	44.6
100,000	70	504	195
1,000,000	161	2369	1210

Table 5.1 Execution time in seconds for one million subscriptions (5 million linear path predicates)

Because this XML dataset has a very simple structure and there are only 7 distinct tag paths in the subscriptions, the cost of computing the *PathMatch* table is ignorable. Parsing and populating the XML data into DB2 temporary tables takes about 24 seconds, which is included in the result reported in Table 5.1.

For the unroll algorithm, the number of index probes to the subscription table does not depend on the selectivity (number of match results) of the subscriptions. The total running time, however, increases as the total number of match results increases because more time is needed to check the remaining linear path predicates for the tuples retrieved by the index-nested loop join. Checking the remaining linear path predicates is preformed against the *SAX_Parse_Result* table which is small and resident in memory. The running time slightly increases when the total number of notifications grows to 100,000 and the running time only doubles when total notifications grow another magnitude to one million.

The recursive plans are much more sensitive to changes in the selectivity of the subscriptions. As more tuples are retrieved in each iteration, the number of index lookups on the *Query_Predicate* table increases. The rapid increase of the running time of the recursive plans reflects the increase in the number of I/Os performed on the *Query_Predicate* table and its index.

Comparing the last two columns of Table 5.1, we can see that bucket optimization can be very effective when there are several non-equal predicates in the subscriptions. The bucket optimization is very effective at reducing the number of index lookups on the *Query_Predicate* table. With the bucket optimization, the recursive plan actually can out-perform the unroll plan when the selectivity is high (i.e. when the total number of notification is small). This is because the tuple size of the *Query_Predicate* table is smaller than that of the unrolled table. When the numbers of index lookups to the tables are similar, the cost of retrieving tuples for the unrolled plan is higher.

Table 5.2 shows how the different implementations scale with number of subscriptions with fixed selectivity. The total number of messages is 10,000 and there are 100,000 matched results for the test with 1 million subscriptions.

Number of Subscriptions	Unroll	Share	Share-Bucket
100,000	52	101	75
200,000	55	221	119
1,000,000	70	504	195

Table 5.2 Execution time for different number of subscriptions

All the algorithms showed good scalability with respect to the number of subscriptions. As the number of subscriptions grows by an order of magnitude (from 100,000 to 1 million), the execution time of the unroll algorithm is only slightly increased. The execution time of the recursive algorithms also exhibit sub-linear behavior. The reason is that all the implementations use indices to find subscriptions that are related to the published messages, instead of evaluating all the subscriptions.

5.2 Experiment 2: The NASA Dataset

The second set of experiments use the NASA dataset. The NASA data set has a recursive DTD and the nesting structure is much more complex than the synthetic Stock dataset.

We used the XPath query generator from [7] to randomly generate branching XPath subscriptions. Following the practice in [9], the atomic value in each linear path predicate is set to a value that appears in some XML document of the dataset. There are two to ten linear path predicates in each subscription with an average of 7 linear paths per subscription. 80 percent of the linear path predicates are equal predicates. The experiments are conducted on a 2.4GHz Pentium with 512 M of memory. The software we use is DB2 8.1 on Redhat Linux (kernel version 2.4.20).

Our experiments varied the number of subscriptions. Table 5.3 shows the number of subscriptions and linear path predicates for each experiment. Compared to the work in [9], which solves the matching problem with in memory finite state automata, our experiments contain two order of magnitude as many as linear path predicates using only half the amount of physical memory.

	Scale 1	Scale 5	Scale 10	Scale 15	Scale 20
Number of subscriptions (millions)	0.13	0.7	1.4	2.1	2.7
Number of Linear Path Predicates (millions)	1	5	10	15	20

Table 5.3 Number of subscriptions and linear path predicates

The NASA dataset we used in the experiments has 2433 files with a total of 23 MB XML data. There are total 357 thousands atomic values in the dataset, but there are only 73 distinct tag paths leading to these atomic values. These statistics support our assumption that for XML dataset with a DTD, the number of distinct tag paths is small. If the *TagPath_Map* table and the *PathMatch* table are considered as caches for tag paths, the hit ratio is very high except for the very first few XML messages. Unlike the first set of experiments, the number of distinct linear paths in this set of experiments is large. Table 5.4 shows the number of distinct linear paths and the time used to compute the *PathMatch* table for this set of experiments.

	Scale 1	Scale 5	Scale 10	Scale 15	Scale 20
Number of Distinct Linear Paths	6,565	21,637	32,159	38,026	42,271
Time for computing PathMatch	7 sec	20 sec	29 sec	35 sec	37 sec

Table 5.4 Number of distinct of linear paths and time for computing PathMatch table

Figure 5.2 shows the execution time of using the NASA dataset. Because the number of non-equal predicates is small, the bucket optimization does not change the result of the recursive plan. Only results of share without bucket optimization and the results of the unroll plan are shown for this set of experiments. All experiments started with an empty *PathMatch* table. Parsing the XML files and inserting the parsing results into DB2 temporary tables cost about 55 seconds for each run. Both time spent in parsing XML data and computing the *PathMatch* table are included in the reported number.

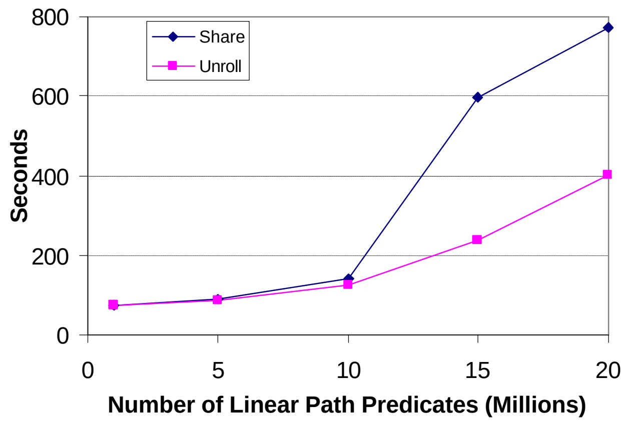


Figure 5.2 Execution time for the NASA dataset

Unrolling the *Query_Predicate* table also demonstrates better scalability for this set of experiments. The DB2 buffer pool is set to 256MB, which cannot hold the working set of the *Query_Predicate* table and its index when more than 10 millions of linear path predicates have been installed into the system. The running time of the unrolled plan degrades more gracefully than the recursive plan when the number of subscriptions is very large due to the difference in the number of index lookups on the *Query_Predicate* table.

6. SUMMARY AND FUTURE WORK

The main technical challenge of implementing an XML publish/subscribe system is to build an efficient, scalable engine to match published XML messages with millions of XPath expressions. In this paper, we designed and implemented such an engine using a relational database. The matching algorithm in our system exploits the commonalities shared between subscriptions as well as between XML messages in the following respect:

1. The common linear paths of the XPath subscriptions and the common tag paths in XML messages are stored in tables of the relational database.
2. The linear paths and the tag paths are matched against each other and the matching result are stored in the *PathMatch* table. The computation of the matching only happens when the system encounters a new linear path or tag path.
3. The predicates on atomic values are stored in tables, and we use the relational join operator to evaluate the predicates against the values in XML messages. The evaluation is efficient by using indices on the predicate tables.
4. The branching structure of the XPath subscription is also store in tables. The branching structure is checked against XML messages using a UDF after the linear path predicates are evaluated.

We analyzed the performance of our implementations and proposed several optimization techniques. Our experiments showed that by unrolling the predicate table, we can achieve high performance and scalability on both simple flat XML datasets and complex semi-structured XML datasets. Compared to earlier XML publish/subscribe systems, using a relational database provides better scalability because the system is no longer limited by the amount of physical memory.

In terms of future work, there can be many extensions to our current implementation. For example, our current subset of XPath does not support joins within a single XML document. We plan to investigate this problem in the future.

7. ACKNOWLEDGEMENTS

The authors would like to thank Marc Schunk for designing the test case for the simple flat XML data. David J. DeWitt carefully reviewed a draft of the paper and gave many valuable feedbacks.

8. REFERENCES

[1] M. Altinel, M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In Proceedings of VLDB 2000, 53-64.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, et al. Matching Events in a Content-based Subscription System. In Proceedings of PODC, 1999.

[3] K. P. Birman. The process group approach to reliable distributed computing. Communications of ACM, 36(12): 36-53, 1993.

[4] C. Chan, P. Felber, M. Garofalakis, R. Rastogi. Efficient filtering of XML documents with XPath expressions. In Proceedings of ICDE 2002.

[5] J. Chen, D. J. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases, In Proceedings of SIGMOD 2000, 379-390.

[6] J. Chen, D. J. DeWitt, J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In Proceedings of ICDE 2002, 345-356.

[7] Y. Diao, P. Fischer, M. J. Franklin, R. To. Yfilter: Efficient and scalable filtering of XML documents. In Proceedings of ICDE, 2002.

[8] T. J. Green, G. Miklau, M. Onizuka, D. Suciu. Processing XML Streams with deterministic automata. IN Proceedings of ICDT, 2003.

[9] A. Gupta, D. Suciu. Stream Processing of XPath Queries with Predicates. In Proceedings of SIGMOD 2003, 419-430.

[10] E. N. Hanson, C. Carnes, L. Huang, et al. Scalable Trigger Processing. In proceedings of ICDE 1999, 266-275.

[11] B. Oki, M. Pfleugl, A. Siegal, et al. The information bus: An Architecture for extensible distributed systems. Operating Systems Review, 27(5)58-68, 1993.

[12] F. Peng, S. S. Chawathe. XPath Queries on Streaming Data. In Proceeding of SIGMOD, 2003.

[13] D. Powell. Group Communications. Communications of the ACM, 39(4):50-97, 1996.

[14] T. K. Sellis. Multiple Query Optimization. TODS 13(1): 23-52 (1988)

[15] P. Seshadri, Building Notification Services with Microsoft SQL Server. In Proceedings of SIGMOD 2003

[16] Sun Microsystem Inc. Java Message Service (JMS) specification. http://java.sun.com/products/jms

[17] Sun Microsystem Inc. Java 2 Enterprise Edition (J2EE) specification. http://java.sun.com/j2ee/1.4/docs/

[18] Y. W. Wang, E. N. Hanson. A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions, In Proceedings of ICDE, 1992.

[19] A. Yalamanchi, J. Srinivasan, D. Gawlick. Managing Expressions as Data in Relational Database Systems. In Proceedings of CIDR 2003.

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

[21] XQuery 1.0: An XML Query Language. http://www.w3.org/XML/XQuery

[22] H. Zeller. Non-Stop SQL/MX Publish/Subscribe: Continuous Data Streams in Transaction Processing. In Proceedings of SIGMOD 2003.