Hybrid approach for XML access control (HyXAC)

By

Manogna Thimma

Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Arts.

Thesis committee:

Dr. Bo Luo: Chairperson

Dr. Prasad kulkarni

Dr. Luke Huan

Date Defended: 07/19/2012

The Thesis Committee for Manogna Thimma
certifies that this is the approved version of the following thesis:

Hybrid approach for XML access control (HyXAC)

____

Dr. Bo Luo: Chairperson

Date approved: ________________

ABSTRACT

While XML has been widely adopted for sharing and managing information over the Internet, the need for efficient XML access control naturally arise. Various access control models and mechanisms have been proposed in the research community, such as view-based approaches and preprocessing approaches. All categories of solutions have their inherent advantages and disadvantages. For instance, view based approach provides high performance in query evaluation, but suffers from the view maintenance issues.

To remedy the problems, we propose a hybrid approach, namely HyXAC: Hybrid XML Access Control. HyXAC provides efficient access control and query processing by maximizing the utilization of available (but constrained) resources. HyXAC uses pre-processing approach as a baseline to process queries and define sub-views. It dynamically allocates the available resources (memory and secondary storage) to materialize sub-views to improve query performance. Dynamic and fine-grained view management is introduced to utilize cost-effectiveness analysis for optimal query performance. Fine-grained view management also allows sub-views to be shared across multiple roles to eliminate the redundancies in storage.

Table of Contents

CHAPTER 1

INTRODUCTION

The eXtensible Markup Language (XML) is one of the most active areas in the IT industry. It was designed to store and transport data. A large quantity of information is presented in XML format on the web for easy transportation. Due to the increased use of XML documents over the web, the need to secure these documents has increased. In a multi – user system, where the information is being shared across users who have different permissions, the need to implement a security model which gives controlled access to the authorized users is very important. XML access control was introduced to suit this purpose. XML access control is a security mechanism which restricts the access of the XML data to authorized users. Many access control models and enforcement mechanisms have been proposed to prevent the unauthorized disclosure of XML data. An access control model defines who can access which information under what circumstances. It is implemented based on the access control policies. Most of the access control models use a tuple format, where every rule is represented in the form of a tuple. This tuple denotes the permissions to a node or an element for a given role. An access control enforcement mechanism defines the way the access control is employed over the XML documents.

There are different enforcement mechanisms such as pre – processing approaches, view based approaches, post – processing approaches etc. which are popular. The view based approaches create and manage views for every user/role by making a copy of all the nodes that are accessible by the user/role. All the user queries are directed at the users view. This eliminates the fear of the user querying unauthorized information. The pre-processing approaches modify the incoming user query to a new query which is safe and accesses only the authorized information. Such queries can then be evaluated on the data base without any further security. Post processing approaches evaluate all the queries from the user on the database and get the results. Once the results are obtained, they prune them to discard any of the unauthorized data. All these approaches are secure, but the view based approaches are considered the fastest because of the smaller documents (views). Views are relatively easy to implement when compared to other approaches. Though this method provides good security, it is still at a disadvantage of updating and maintaining large number of views. The pre – processing approach suffers from a high cost of evaluation of the queries as they are being evaluated against the entire data base. There is a high amount of risk involved in case of post – processing approaches because the results are obtained first and then pruned in the second stage of this approach. Each of these approaches suffers from their own problems. However to the best of our knowledge there is no good approach which can provide good security, along with, decreasing the cost of evaluation of the queries as well as the storage costs.

An approach that can combine the advantages of a few of the above mentioned approaches to develop a better solution sounds like a promising idea. Use of a pre- processing approach makes sure that partially safe queries are rewritten to safe queries and the view based approach adds additional security by providing only accessible data. The cost of evaluating the queries on smaller documents like the views is comparatively better than evaluating on the entire document. In this thesis work, we introduce the hybrid approach for XML access control (HyXAC), which combines these two approaches and adds a few additional features to make query processing very safe and efficient. The HyXAC model uses the pre – processing approach as the base line to process the queries. It defines views for every positive access control rule or a set of rules. This approach is beneficial in terms of security and efficiency, but what about cases where there are limited resources available? HyXAC handles this problem by dynamically allocating resources and materializing the views. The dynamic allocation is done in a way to provide a cost effective solution as well as improve the average query processing time. The fine grained views are shared among various users/roles. This eliminates the need for maintaining a view for every role, thus reducing the number of views. Therefore, views are materialized since the storage overhead is always minimal

Advantages of this approach: (1) the access control mechanism is very secure because of the use of both the pre – processing and view based approaches, (2) fine grained views are defined for an access control rule or a set of access control rules, instead of a user/role. This eliminates the overlapping of nodes or elements between views, (3) sharing of views between numerous roles is possible, this reduces the redundancy of storage data and also improves the performance, (4) depending on which views are frequently used and other dynamic factors, this model decides to materialize those views which are beneficiary and are cost effective.

The rest of this paper is organized as follows. The background and related works are discussed in chapter 2. An introduction to our hybrid approach is developed in chapter 3. Chapter 4 discusses the dynamic implementation and performance optimization techniques used in this thesis work. The experimental results for all the different cases and solutions discussed in chapter 4 are shown in chapter 5.

CHAPTER 2

BACKGROUND

2.1. Preliminaries

Extensible Markup Language (XML) is a simple markup language used to structure electronic documents. It can encode text and data in a format which can be processed easily and exchanged across multiple platforms, applications etc. The XML document is composed of a sequence of nested elements, each of which has a start tag and an end tag. XML now-a-days is most commonly used for storing and interchange of data over the internet. Because of the increasing importance of XML, an amazing amount of work related to its security has been done over the years. Access control is one of the basic security mechanisms. The XML access control model is important to limit the access of XML data to authorized users. In most cases, the access control is implemented using access control lists, or access control rules or policies. An access control list is defined for every element in the XML document. Each access control list for an element contains the list of users who can access the element. An access control rule is a rule that defines the accessibility information with respect to a node (or set of nodes) or element in the XML document for a given user. It defines which user can or cannot access which part of the document. A list of access control rules forms an access control policy.

2.1.1. Access control model

An access control model controls the ability to access secure information by authorized users. As stated above, it is generally implemented using access control rules/policies or access control lists. Few of the earlier access control models that are widely recognized are: Role based access control (RBAC) [55], Attribute based access control (ABAC), Discretionary access control (DAC) and Mandatory access control (MAC). Most of these previous models are not fine grained, i.e. they enforce access control over top level objects or objects with simple structures like, tables, directories etc. The XML access control is a fine grained access control model which defines access control over the XML elements and nodes. Instead of controlling access to the entire document, it limits access to the sub structures in XML documents.

Numerous amount of work was done on the different authorization models. [21, 43, 25] were a few of the earliest works. [21] defines an access request as a triple which specifies the subject, object and the access modality which specifies whether the subject is requesting a read or a modify access. In addition to the authorization model for XML, [43] presents an XML access control language that integrates authorization; confidentiality etc. [25] defines a 5 – tuple ACR where the type is L, R, LW, or RW. LW is local weak, RW is recursive weak. These give preference to the schema (DTD) authorization over the instance (XML) authorization. [47] proposes a different rule function based access control model to improve scalability and performance. [56, 16] talk about different labeling schemes for XML documents. [24] introduces the concept of fine grained access control in XML documents. [20, 4, 48, 22] are a few other papers related to access control and authorization models.

The access control model in our approach is specified using a 4 – tuple format. Every access control rule (ACR) consists of a tuple, ACR = {subject, object, action, sign}, where (1) Subject is the user/role who is authorized to access the information; (2) Object is a set of XML elements or nodes (or set of nodes) that are accessible to the subject; (3) Action is the operation that can be performed on the object by the given subject (ex: read, write, update etc.) (4) Sign specifies whether the action can be performed on the object or not (ex: {+, –}. + specifies “granted” and – specifies “denied”). The “–“ takes precedence, when there is a conflict between rules concerning the same node/set of nodes. When there is no explicitly defined rule for a node then the access to that node is denied. In this thesis work we only consider the positive rules.

Generally, XML access control rule (ACR) is described using a 5 – tuple model like in [25]. Each ACR is represented as a 5 – tuple: {subject, object, action, sign, type}, the subject, object, action and sign are the same as in the 4 – tuple model defined in this thesis work. The type refers to a local check (LC) or a recursive check (RC). When type is LC, it implies that the action specified in the action field can or cannot be applied on the elements or nodes specified in the object field based on the sign (+ or –) by the subject. When the type is RC, the action is not only performed on the element or the node (or set of nodes) specified in the object field but is recursively applied on all of the descendants of the node or set of nodes. In our access control model we assume that RC is applied by default to all access control rules.

The object is usually specified using some XML query languages like XPath. XPath (XML Path Language) is a query language used for selecting nodes in a XML document. XPath uses path expressions to select nodes. The following are a few path expressions that are commonly used.

Expression	**Description**
/x	Selects the node x from the root node
//x	Selects all the paths from the current node to the node x, one or more levels deep
@	Selects attributes
/*	Matches any element node
//*	Selects all the elements under the selection

Table 1

Example rules:

R1 {role1, /site/catgraph/edge, read, +}

R2{role1, /site/closed_auctions/closed_auction/quantity,read, +}

R3 {role1, /site/people/person//*, read, +}

R4 {role1, /site/open_auctions/open_auction/privacy, read, +}

R5 {role1, /site/open_auctions/open_auction/seller, read, +}

R6 {role1, /site/closed_auctions/closed_auction/date, read, +}

R7 {role1, /site/open_auctions/open_auction/quantity, read, +}

R8 {role1, /site/regions/namerica//*, read, +}

R9 {role1, /site/regions/africa/item//*, read, +}

R10 {role1, /site/categories/category//*, read, +}

Each of the rules describe an action that role1 can perform on the given object. The “+” sign in all the rules, denotes that the object is accessible to the subject. Rule R1 says that role1 is permitted to access the “edge” node under the “catagraph” node. Rule R2 permits role1 to access the “quantity” node under “closed_auction”. The //* in rule R3 denotes all the nodes under the node “person”. Similarly, rules R4 through R10 denote different nodes that can be accessed by role1.

2.1.2. Enforcement mechanisms:

The access control enforcement mechanism defines the way in which the access control model is enforced. There are various enforcement mechanisms that are available: view based approaches, pre – processing approaches, post – processing approaches etc. A few examples: [18, 56] describe different labeling methods which help answer queries and [3, 23]

View based approaches:

View based approaches [24] create a separate view for every role. Each view contains a copy of all the accessible information i.e. all the nodes and elements in the XML document which can be accessed by the role. When the user queries, the view related to that user is loaded and the queries are answered over the view. Using views have several advantages: Views are generally very small and require very less memory space to be stored. They improve query performance, as all the queries are evaluated on a much smaller document than the original document. Since the view contains only authorized information there is no risk of retrieving undisclosed information and no additional security checks are required, query evaluation is very safe. However, view based approaches have their own drawbacks too. The major disadvantages include the updating and maintaining the views. As the number of views increases it is difficult to maintain and update these views. Any simple change in the database requires all the views containing that information to be updated. Another disadvantage of higher number of views is the requirement of larger resources. A lot of memory space is required to store all these views. Each role has a view of its own. Although most of the roles have similar access control rules, there is no sharing of views. This increases redundancy of the data being stored in the views.

A significant amount of work has been done on database views. Most of the papers tried to improve the efficiency or speed up the query processing in different ways. [32, 7, 5, 19] are a few initial papers which talk about materialized views in relational databases. [32] is one of the earliest paper which addresses the problem of updating materialized views. A great deal of work has also been done on improving efficiency of answering queries in XML databases. [63, 54] are a few papers which talk about answering queries using XML views and improving efficiency. They use different methods, [63] uses inverted lists evaluation model where sub lists of the inverted lists for the labels of every view node is materialized. [54] defines an algorithm called the MiniCon algorithm which collects MiniCon Descriptor's (MCD) similar to buckets to find containment checks to speed up the answering. [62, 60, 1, 2] define different techniques which rewrite XML queries using views to enhance the performance of the queries. [9, 40] present models for updating views in an efficient manner when there is any change in the database.

Pre – Processing approaches:

Pre – processing approaches check the queries against access control rules before evaluating the query. Only safe queries that pass through are evaluated. All the unsafe queries are rejected at the first go. [44] defines static analysis which creates automata and uses regular expressions generated from queries to compare with the automata. Then, decides if the query is accepted or denied. QFilter [12] is another pre – processing model which uses NFA structures developed from access control rules to check queries and decide if they are accepted or denied. If the query is neither completely accepted nor rejected then QFilter rewrites them to a safe query.

The pre – processing approaches eliminate the need to create, maintain and update views. This kind of an approach is very good for use in situations where the database is independent from the access control rules. It improves the overall query performance and is very safe because it eliminates all the unsafe queries. The disadvantage of such an approach is that the safe queries are evaluated against the entire database. This requires the entire data base to be loaded into the memory. Also, the query evaluation time increases with the size of the database.

Our approach (HyXAC) uses the QFilter as the baseline and develops a better model that creates fine grained views to answer queries.

Post – processing approaches:

Post – processing approaches evaluate all the unsafe queries on the documents in the first stage. After getting all the results, the post – processing approach has a post – filtering stage which prunes all the unwanted data from the results based on the access control rules. [37] describes one post – processing approach called the AFilter. The post – processing approaches are very useful when the access control rules are to be applied separate to the database. But however there is a higher risk of external threats because it carries unsafe data till the last step.

2.2. Related Work

2.2.1. QFilter

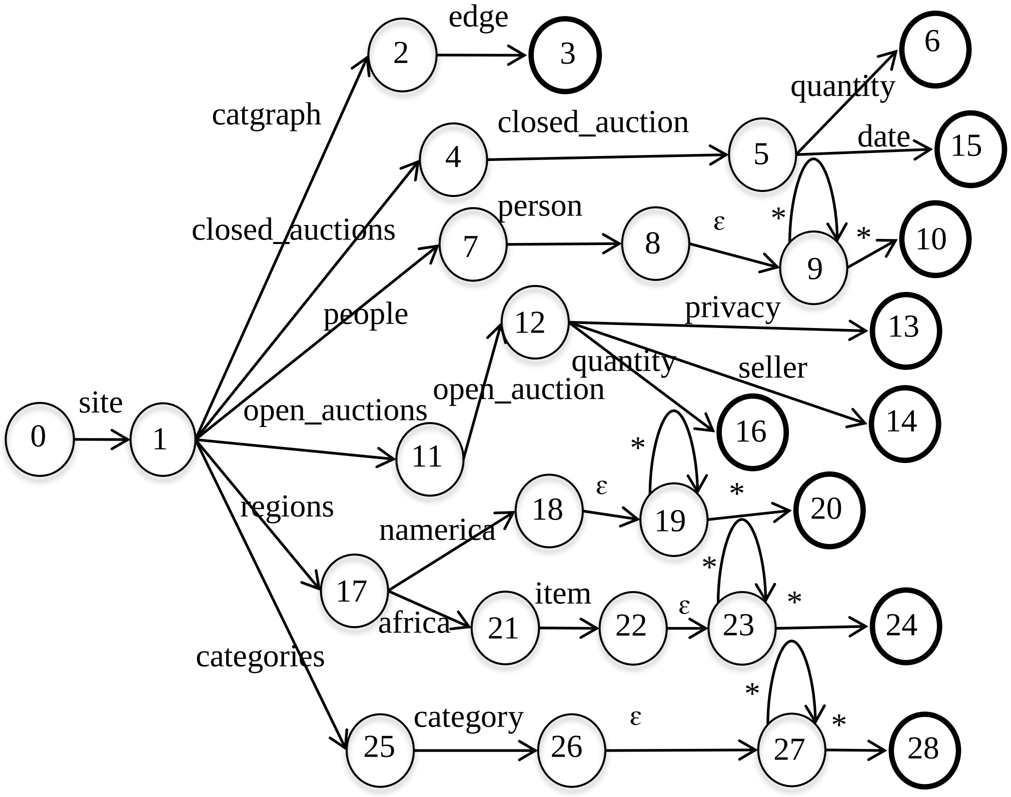
In this section, we present the pre – processing approach - QFilter used in our approach. QFilter is a NFA based implementation. It reads a query Q and access control rules (ACR) as input, builds an NFA structure using the ACR, process the query over the NFA and outputs a modified safe query Q'. There are two stages in the QFilter, one is the construction stage and the other is the execution stage.

In the construction phase, the QFilter builds an NFA structure from the access control rules. In the execution stage, it filters out the incoming queries and produces a list of safe queries. QFilter performs three types of operations when filtering a query: (1) Accept – if the query is completely answered by the access control rules then it is considered to be a safe query and is accepted by the QFilter. (2) Rewrite – if the query is partially answered by the access control rules i.e. when a part of the answer is denied or is not accessible to the user then the query is rewritten to new safe query Q' which returns only the authorized part (3) Deny – if the query is not answered by the access control rules then the query is denied by the QFilter.

By the end of the construction step, the QFilter generates a list of safe queries which can be evaluated against the database.

The NFA structure generated by QFilter for the access control rules given in example rules is shown in figure 1.

Figure 1:



The object in the first access control rule R1 is /site/catgraph/edge. The QFilter breaks down the XPath into fragments. It starts with the first fragment /site. The state 0 is created from /site. Then the next state, state 1 is created from the next fragment /catgraph. State 2 is created from /edge. When the QFilter reaches the end of the XPath, then it creates a new state i.e. state 3 and marks it as “Accept”. Now the next rule R2 is added. The XPath in the rule is /site/closed_auctions/ closed_auction/ quantity. The first fragment in this XPath is identical to the state 0 in the QFilter.

So, this step is reused. The next element is /closed_auctions, which cannot be mapped to the next state in QFilter, so a new state – state 4 is created. The rest of the fragments in R2 are added to the QFilter in the same way. Rule R3 is added in a similar manner till it reaches state 7. The next fragment is a //*. The QFilter adds a ε transition state to handle the "//" state. The //x is a notation used to search for element x from the current node which can be found with any path or paths preceding it. And a //* generalizes it further by stating that any node from the current state. The ε transition is defined at state 8. The * is added as state 9. The //* denotes any path or paths that are one or more levels deep. So, there is a self-loop created at state 9 which makes sure that any number of elements can be included in the path. Then a new state – state 10 is added by the QFilter and marked as "accept".

All the remaining rules are then added to the QFilter in a similar fashion. Since we are creating a QFilter for all the positive rules, every end state generated by each rule is an accept state.

After the generation of the NFA structure from the access control rules, QFilter checks every query against the NFA to classify it as accept or a reject. If all the data that the query Q requests is allowed by the access control rules then QFilter accepts the query. If all the data the Q requests is rejected by the access control rules then the query is rejected. If some of the data that Q requests is rejected, but the rest is allowed by the access control rules then the query is modified such that the part of the query that is requesting the rejected data is pruned. This is the execution stage in the QFilter.

Example 1:

Given the following queries,

Q1 /site/open_auctions/open_auction/privacy

Q2 /site/catgraph/edge

Q3 /site/regions/*/item/mailbox

Q4 /site/open_auctions/open_auction/interval/end

And the access control rules in Example rules:

QFilter generates the NFA shown in figure 1. Consider the first query Q1. QFilter splits the query into XPath fragments. It sends the first fragment, /site to the NFA structure. This matches the key “site” in the structure. The execution continues to the next state – state 1. The next fragment /open_auctions is mapped to the key “open_auctions” at state 1. The execution continues to state 11. Similarly, /open_auction is mapped to “open_auction” at state 11 and /privacy is mapped to “privacy” at state 12 and the execution moves to state 13 which is an accept state. The query Q1 reached an accept state in the NFA structure so the QFilter accepts this query.

The query Q2 is accepted by the QFilter as it reaches accept state, state 3. Query Q3 is started in a similar manner. The /site and /regions are mapped at state 0 and state 1. The /* in Q3 is accepted by state 13. * is a wildcard character and is mapped with all the elements. The execution at this point branches into 2. The first branch continues to state 18 and the other to state 21 i.e. /site/regions/namerica and /site/regions/africa. In the first branch, a ε transition is encountered at state 18. The /item in the query is mapped to ε and reaches state 19. The next fragment in the query /mailbox is then mapped to the * at state 19 and reaches state 20 which is an accept state. The second branch, maps /item to the “item” key at state 21 and continue to state 22. The ε at state 22 is considered to be a null and the QFilter execution moves to state 23 where the /mailbox is then mapped to *. The execution continues to state 24 which is an “accept” state. Both the branches end at an accept state. So the query is accepted by QFilter.

The query Q4 is processed in the same way and is rejected by the QFilter because it does not reach an “accept” state in the NFA structure. All the elements or nodes that are not specified by the access control nodes are considered to be inaccessible to the user.

QFilter uses the deep set operators, DEEP – EXCEPT, DEEP – UNION and DEEP – INTERSECT defined as (1) DEEP – EXCEPT: For any P1 DEEP-EXCEPT P2, if a set of P2 nodes are present in P1, they are pruned and the rest of P1 is returned; if P1 nodes are contained in P2 then all nodes corresponding to P2 are pruned and the remaining part is returned (2) DEEP – UNION: For any P1 DEEP – UNION P2, if nodes of P1 are descendants of P2 then return P2, if P2 nodes are descendants of P1, return P1, if there is no overlap, return P1 ∪ P2 (3) DEEP – INTERSECT: For any P1 DEEP – INTERSECT P2, if nodes of P2 are descendants of P1 then return P2, if P1 nodes are descendants of P2, return P1, if none of them are contained within one another then return empty. These operations are used to rewrite the queries.

Apart from the normal queries, QFilter can handle queries or access control rules with predicates as well. Predicates provide additional information to be considered while answering the query. A predicate is contained within the [] in the XPath and usually follows the parent element of what will be tested. Predicates are generally used to check the amount or value of an element. When the predicate is present in the query, the predicates are stored and attached to the query if the query reaches an accept state or is rewritten. The predicates are rejected if the query is rejected. When predicates are present in an access control rule, QFilter attaches the predicate to the element in the NFA structure and it requires further processing. When a query is being processed and reaches that element, then QFilter processes the predicates further. If the element has only one corresponding predicate then it attaches the predicate to the query. In case of different corresponding predicates, QFilter merges the different predicates and attaches them to the query. The merging is done in a simple way by attaching the multiple different predicates one after the other. In case of same or related predicates, they can be further optimized by combining the values or by rejecting.

Until now, we have seen how QFilter handles positive rules. When negative rules are included in the access control policy then, QFilter needs special handling. A separate QFilter is generated for negative rules called the negative QFilter. The QFilter generated for positive rules is called the positive QFilter. The construction and the execution stages for the negative QFilter are the same as the positive QFilter. The outputs from both the QFilters are combined using the DEEP – EXCEPT operation discussed above. In this thesis work, we assume that the access control policy consists of only the positive rules.

The general QFilter described above was designed for only one role. In a distributed environment, there are multiple roles which require multiple QFilters to enforce the access control. An extension of the QFilter called the Multiple role QFilter (MRQ) was designed to address this problem specifically. MRQ construction is similar to QFilter construction. The access control rules for different roles have a great similarity which in turn makes the QFilters of the roles similar. MRQ merges these similar QFilters into a single data structure. For every node in the NFA structure, MRQ stores two lists: the access list and the accept list. The access list is a list which contains information about which role can access the given node. A 0 indicates that the node is not accessible and 1 indicates that the role can access the node. The accept list states which is an accept state for which role. The length of the accept list and the access list is the number of roles in the system.

Example 2:

Consider two roles with the following access control rules:

R1 {role1, /site/people/person/name, read, +}

R2 {role1, /site/regions/namerica, read, +}

R3 {role2, /site/people/person/address/city, read, +}

R4 {role2, /site/regions, read, +}

The /site/people/person and /site/regions part in both the NFA are the same for role1 and role2. The NFA structure developed by Multiple Role QFilter for the given access control rules is as follows:

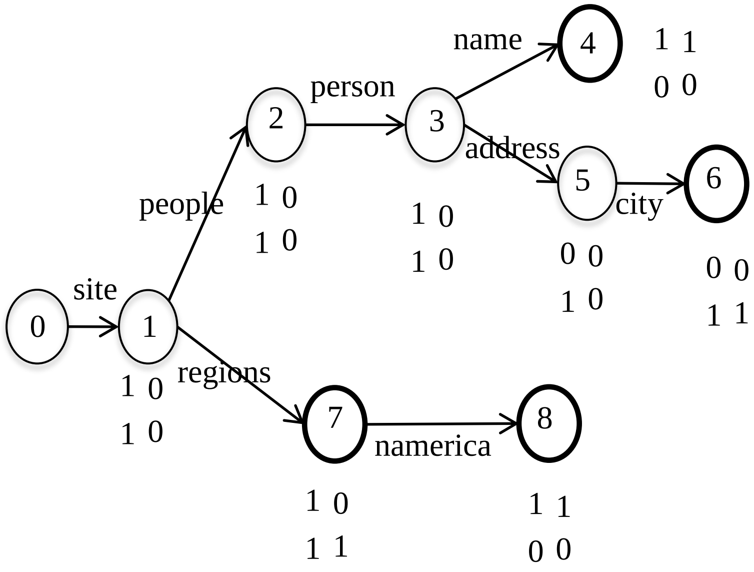


Figure 2

At every state there are two lists. The first list is the access list and the second list is the accept list. State 1 is accessible by both role 1 and role 2, so, the access list contains a 1 for both the roles. And it is not an accept state for both the roles, hence accept list contains 0. States 2 and 3 are accessible by both roles but is not an accept state for either. State 4 and 8 are accessible by role 1 alone and are accept states for role 1. State 5 and 6 are accessible by only role 2 and state 6 is an accept state for role 2. State 7 is accessible by both roles but is an accept state for role 2.

The query evaluation on a MRQ is similar to the query evaluation on a QFilter. The query is broken down into XPath fragments and each fragment is matched to the list of elements stored at each node. When the final fragment in the query is matched, the access list is verified to check if the role can access the node or not. If the role can access the node then it verifies if the node is an accept state. If the node is accessible and is an accept state for the given role MRQ accepts the query. If the node is not accessible by the role, then MRQ rejects the query.

MRQ also has three types of operations: accept, rewrite and deny. These three operations are similar to QFilter operations.

QFilter guarantees that authorized users can access authorized information. It ensures that none of the queries violate the access control rules by rejecting or rewriting the queries. This pre – processing model can be used external to the database. Another advantage of this approach is that there are no maintenance or storage issues associated with this approach.

CHAPTER 3

HYBRID XML ACCESS CONTROL

3.1. Introduction

In this section, we introduce our new model named Hybrid XML Access Control (HyXAC). HyXAC is a hybrid model produced by the combination of a pre – processing approach – Qfilter and the view based approach. It takes the access control rules and queries as input, dynamically materializes views and evaluates safe queries to produce results. The HyXAC model is implemented in two stages. In the first stage, we use QFilter to filter the input queries and produce safe queries, and we create views. In the second stage, the views are dynamically materialized and are used to answer queries.

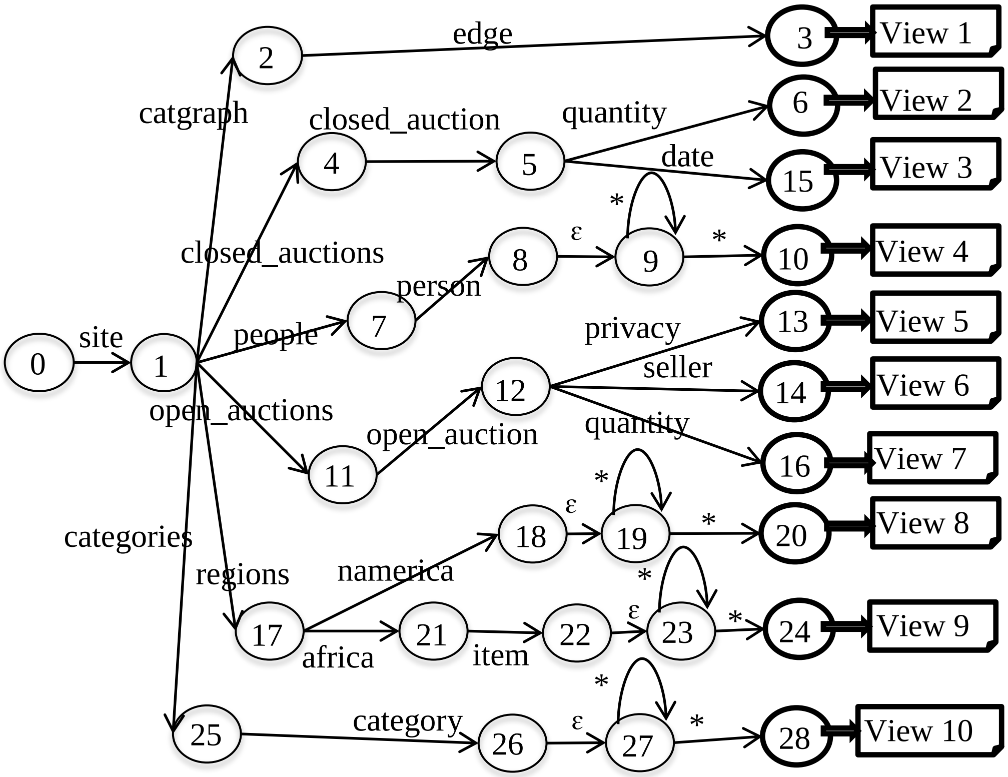
The QFilter used in our approach is similar to the QFilter described in the previous section. In this thesis work, we only consider positive access control rules, i.e. only a positive QFilter is used to filter queries. The list of access control rules are used to construct an NFA structure. When an access control rule is added to the NFA structure, the last XPath fragment from the object XPath in the access control rule ends at an accept state. So, every accept state in the QFilter corresponds to one access control rule or a set of access control rules. In the execution stage of the QFilter, the queries are tokenized and are mapped to the NFA structure. When the last token from the query reaches an accept state in the NFA structure, it implies that the query is accepted by the QFilter. That is, every safe query reaches an accept state; every accept state corresponds to an access control rule or a set of access control rules; so, every safe query is answered by one or more access control rules. Using this concept, in HyXAC model we decided to create views from access control rules so that all the queries reaching an accept state can be answered by the view that corresponds to the (set of) access control rule(s).

We know that, in the traditional view based approaches, a view is created for every role. Most of the roles have identical access control rules and there is a lot of identical data being copied into most of the views. The redundant storage of data is one of the drawbacks of the view based approaches. Creating a view for every accept state which is formed by one or more access control rules rather than creating one for every role, answers the problem of redundant data. All the queries reaching an accept state are answered by the view. Moreover, roles having identical access control rules can share the views. Similar access control rules end at the same accept state and queries reaching this accept state are answered by the view created for this accept state. This eliminates the storage of the same data for different roles.

The QFilter is modified to keep track of the access control rules reaching the accept state. When a query is accepted or rewritten to a safe query by the QFilter, it generally outputs the query. We added a new function to the QFilter which not only outputs the query but also gives the ACR number that answered the query. Using this rule number, we can create a view for the corresponding ACR.

Figure 3 below, shows the NFA structure developed by the QFilter for the example rules, along with a view being created for every access control rule. View 1 contains the edge nodes under all the catgraph nodes. View 2 contains the quantity information under closed_auctions/ closed_auction. Similarly, all the other views are created. One view can answer all the queries reaching that accept state.

Figure 3:



The second part of HyXAC model is to dynamically materialize the views. Now that we have all the views and we know which view answers the query, we can materialize the view when required and answer the query. Materializing all the views and loading them into memory to answer all the queries is a good idea, but is not always a feasible solution. So we need to decide on which are the best views to be loaded into memory. There are several factors that are to be considered when deciding which views to load, such that the average query performance is improved and the total cost is reasonable. The dynamic view loading and other solutions are further discussed in chapter 4.

There are several advantages of using the HyXAC model:

a. One view can be defined for one accept state or a set of accept states (access control rule or a set of rules)

   A view is created for every accept state in the NFA structure. Each accept state can be formed by one or more similar access control rules. These views are less expensive, easy to load and improve the query performance greatly. In some cases there are multiple accept states having the same parent node. In such a scenario, materializing the parent node might be beneficial. When a view is created for the parent node and is materialized, the access control will still be intact. That is, all the queries are evaluated in the same way as before. A query is accepted only if it reaches the accept state. Once the query is considered safe, then in place of loading the view related to that accept state, we load the view created from the parent node to answer it.

Example 3:

Consider four rules,

R1 {role1, /site/people/person/name, read, +}

R2 {role1, /site/people/person/address/city, read, +}

R3 {role1, /site/people/person/address/street, read, +}

R4 {role1/site/people/person/address/zipcode, read, +}

The QFilter NFA structure for these rules is as follows:

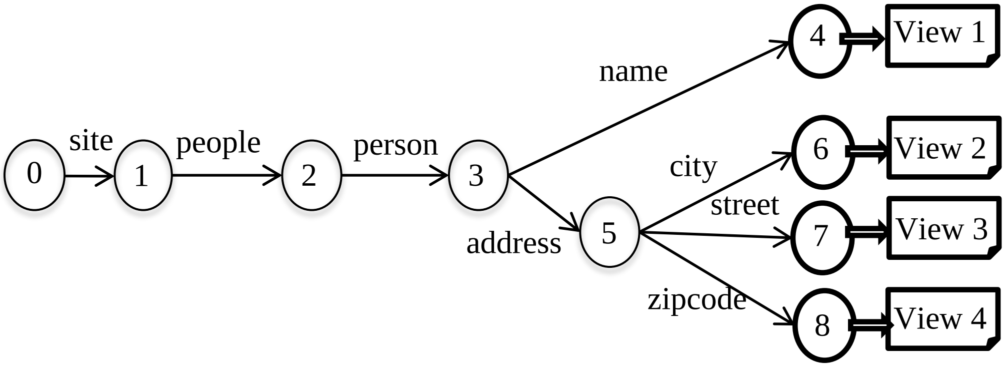


Figure 4

By observation, we can say that accept state nodes 6, 7 and 8 have the same parent node i.e. node 5. Materializing the parent node (node 5) can be beneficial when (i) the total benefit of materializing parent node is higher (ii) the cost of materializing the parent is not too high, i.e. we do not want views that degrade the query performance. All the queries reaching nodes 6, 7 or 8 are then answered by the view created for node 5. In this example, we can also consider materializing node 3 i.e. going back two levels. Then all the queries reaching nodes 4, 6, 7, or 8 are answered by this view. In this case we also need to check if the total cost of materializing node 3 is not too expensive than the sum of the costs of materializing nodes 4, 6, 7 and 8.

---

b. The fine grained views are less expensive to keep in memory

In HyXAC, views are defined for an accept state formed from an access control rule or a set of access control rules. A view for an accept state is relatively less expensive when compared to a view for a role. This is because a view created by the traditional view based method for a role, contains all the accept nodes that are accessible to the role and the fine grained views in HyXAC are like parts of the traditional view which contains a single accept node. This reduces the storage space required. Because of their inexpensive nature, it is always better to materialize such fine grained views.

Example 4:

Consider the following rules:

R1 {role1, /site/categories//name, read, +}

R2 {role1, /site/people/*/name, read, +}

R3 {role1, /site/catgraph/edge, read, +}

R4 {role1, /site/regions/asia/item/location, read, +}

In the traditional view based approach, a view is created for every role. A single view View_t is created for the above rules. This view contains all the “name” nodes which are descendants of the /site/categories, the “name” nodes which are one level under /site/people, the “edge” nodes under /site/catgraph and the “location” nodes under /site/regions/asia/item.

Suppose there is a query Q: /site/people/person/name. The entire view View_t, is materialized and loaded into memory to answer the query. But the query Q is answered by rule R2 alone. There is additional information in the view which has to be loaded into the memory as well.

In our approach, 4 different views are created for the above four rules. The first view View_1, contains all the "name" nodes descendants of /site/categories, the second view View_2, contains all the "name" nodes one level under /site/people/. Similarly views 3 and 4 contain nodes specified by the object field in the access control rule. So when the query Q is to be answered, $\mathrm{View}_2$is loaded into memory. This eliminates loading of additional data which is not useful to answer the query.

So, the views created by HyXAC are small and are less expensive when compared to the views created by the traditional method.

---

c. Fine grained views can be shared across roles.

   Views once created can be shared among multiple roles. Multiple roles having similar access control rules can be combined together to form a single NFA structure in the QFilter (multi role QFilter). The multi role QFilter (MRQ) defines the accept states for every role in the accept list at each node. When queries from multiple roles reach the same state, the view associated with that state is loaded to answer the queries.

Example 5:

Consider the following access control rules for two different roles.

R1 {role1, /site/regions/europe//*, read, +}

R2 {role1, /site/catgraph/edge, read, +}

R3 {role2, /site/open_auctions/open_auction/bidder, read, +}

R4 {role2, /site/catgraph/edge, read, +}

R5 {role2,/site/categories/category/description/parlist, read,+}

When creating views using the traditional view based approach for the given access control rules, a view is created for role 1 and another view for role 2. The first view View_{t1}, contains all the nodes under /site/regions/europe and the "edge" nodes under /site/catgraph. The second view View_{t2}, contains "bidder" nodes under /site/open_auctions/ open_auction, the "edge" nodes under /site/catgraph and the "parlist" nodes under /site/categories/category/description. When comparing the two views, we find that the "edge" nodes under /site/catgraph are repeated in both the views.

Using HyXAC model, one view is created for each of the rules. View_1 contains all the nodes under /site/regions/Europe, View_2contains all the "edge" nodes under /site/catgraph. View_3 contains all the "bidder" nodes under /site/open_auctions/open_auction. Rule R4 is identical to R2. When adding this rule to the multi role QFilter it ends at the same accept state as rule R2. Therefore, we do not create a new view to represent this rule. View_4 contains all the "parlist" nodes under /site/categories/category/description. On the whole, we have 4 small views in place of two bigger views.

Suppose there are two queries Q1: /site/catgraph/edge and Q2: /site/*/edge, Q1 is placed by role 1 and Q2 by role 2. In traditional method to answer Q1, the view View_{t1} is loaded into memory and to answer Q2 View_{t2} is loaded, though both the queries give the same answer. Loading these two views take up additional amount of space and the average query processing time is also increased because of loading two views. In HyXAC, we load only View_2 into memory and this answers both Q1 and Q2. Thus decreasing the memory usage and improving the query processing time. Views in HyXAC are shared among different roles.

---

d. Views do not overlap

   A view is created for an accept state and every access control rule when added to the NFA structure ends at an accept state. So, all the access control rules which are unique end at different accept states and similar access control rules end at the same accept state. We cannot have the same data in two views because all the similar access control rules come to the same accept state and one single view is created for this accept state. So, we can say that the data in the views do not overlap.

3.2. Discussion

The HyXAC model ensures better security and performance when compared to the other access control models. There are certain aspects in the model which needs a little bit of further addressing.

Consider the following rules:

R1 {role1, /site/open_auctions/open_auction/privacy, read, +}

R2 {role1, /site/categories/category/description, read, +}

R3 {role1, /site/open_auctions/open_auction/quantity, read, +}

The QFilter generated from these three rules will contain three accept states. HyXAC creates three fine grained views, one for each of the accept states. Given a query Q: /site/ open_auctions/ open_auction [privacy=”Yes”]/quantity. This query is answered by both rules R1 and R3 combined together. The query requires accept states/views 1 and 3 to be materialized to answer it. When multiple views are required to answer a query it gets complicated. We suggest merging of the views to create a bigger view, but this might not always be advantageous. We are still working on handling situations like these.

Another problem faced during one of our experiments is: Given a query Q: /site/ closed_auctions/ closed_auction/ annotation/description/text and let the access control rules for a given role be:

R1{role1,/site/closed_auctions/closed_auction/annotation,read,+}

The query Q cannot be answered by the rule R1. According to the QFilter "annotation" is considered to be an accept state. When the query is being matched to the NFA structure by the QFilter, it comes across additional path elements "description" and "text" which cannot be mapped to in the NFA structure. The QFilter denies this query, because it cannot reach an accept state.

For all such cases, we can add a //* at the end of the XPath in the rule if all the descendants of the node are also accessible to the user. By doing this, the QFilter adds additional states like ε and * after the end of "annotation". And when the query comes in, it can map the descendants of "annotation" to ε and * and reach the accept state.

CHAPTER 4

PERFORMANCE OPTIMIZATION

Until now, we have seen how HyXAC works and the several advantages it brings forward. In this section, we will look at the dynamic implementation of the HyXAC model. The main aim of HyXAC model is to improve the query performance under constrained situations where there are limited resources available. Not all the views can be materialized and stored in memory because of memory constraints. Only the required views need to be materialized to help save resources and this should be done in such a way as to improve the query performance. We select the views to be materialized keeping in mind the several factors like the availability of resources, the requirement of the view and other factors that impact the evaluation and performance.

Baseline Method:

The traditional method is the most commonly used method of answering queries. This is the conventional pre – processing approach where safe queries are evaluated over the document. In the first stage, all the queries are passed through the QFilter. The QFilter eliminates all the unsafe queries and rewrites the partially safe queries to completely safe ones. Let the time taken for processing a query $\mathrm{q_i}$ by QFilter be $\mathrm{t_{QF,q_i}}$. Assuming that there are N safe queries which passed through the QFilter, then, the time taken for processing all the N queries by the QFilter will be $\sum_{\mathrm{i=0}}^{\mathrm{N}}(\mathrm{t_{QF,q_i}})$. The N queries from the QFilter are evaluated on the document(s) to get the results in the next stage.

In a few cases, when the memory available is quite huge and the entire document can be loaded into memory, the time taken for loading the document would be high and increases linearly as the size of the documents increases. The time required for evaluating each query over the document also increases proportionally with the size of the document and so does the average query processing time. Let the time taken to load a document be $t_{L(D_m)}$ where $m = G(q_i)$ i.e. $G(q_i)$ is a function that gives the number of the document that answers the query. Let the time taken to answer a query q_i over this document be t_{D_m,q_i}. A document D_m is loaded into memory and the query q_i is answered on it. Once the query is answered, the memory is cleared of the document. Then the document that answers the next query is loaded into memory and the query is answered. Assuming that all the documents can be loaded into memory, this process is repeated till all the queries are answered. The query processing time for a single query q_i is given by:

$$\overline{T}_{Bi} = t_{QF,q_i} + t_{L(D_m)} + t_{D_m,q_i}$$

And the average query processing time is:

$$\overline{\overline{T}}_B = \frac{\sum_{i=0}^{N}\left(t_{QF,q_i} + t_{L(D_m)} + t_{D_m,q_i}\right)}{N}$$

$$\overline{\overline{T}}_B = \frac{\sum_{i=0}^{N}\left(t_{QF,q_i} + t_{D_m,q_i}\right)}{N} + \frac{\sum_{i=0}^{N}\left(t_{L(D_m)}\right)}{N}$$

The first part is moderate but the second part will generally be large because the loading of a document is a time consuming task.

In the other cases, where the memory available is less and the XML document is too big to be materialized and loaded into the memory to answer the queries, generally a part of the document is loaded every time a query needs to be answered. In case an indexing method is available (like in the relational databases), then it is easy to locate the part of the document that answers the query and load that part (assuming that the part is selected such that it fits the memory). But no such methods are available. When there is no indexing available then the worst case would be, to divide the document into different parts which fit the memory and load each part and check if it answers the query. This is repeated until the query is answered. The same procedure is continued for all the queries. This method is very inefficient. The average query processing time increases drastically because it includes the searching of all parts of the document to find one that answers for every query (or) additional cost for indexing all the documents.

In order to avoid loading the document into memory and also to improve the evaluation time, and average processing time, a better method is to use views to answer queries in place of the document.

Solution 1 – Fine grained view method:

Views are relatively small and easy to load when compared with the entire document. As discussed in the previous sections, every accept state in the NFA structure of the QFilter generates a view. In solution 1, all the views are materialized, but none of them are loaded into memory initially. We assume that the entire memory is used as a working space and none of the views can be cached. The QFilter in this method is modified to include an abstract function $\mathrm{F(q_i)}$ which outputs the view ID of the view which answers the query $\mathrm{q_i}$. Assuming that a query $\mathrm{q_i}$ is evaluated by a view $\mathrm{V_k}$, where $\mathrm{k = F(q_i)}$, let the time taken to load the view $\mathrm{V_k}$ be $\mathrm{t_{L(V_k)}}$and the time taken to evaluate a query $\mathrm{q_i}$ against the view $\mathrm{V_k}$ be $\mathrm{t_{V_k,q_i}}$.

The query processing time for a query $\mathrm{q_i}$ is:

$$t_{S1,i} = t_{QF,q_i} + t_{L(V_{F(q_i)})} + t_{V_{F(q_i)},q_i}$$

Considering that a view V_k is loaded into memory before answering the query q_i, the average end to end query processing time is:

$$\overline{T}_{S1} = \frac{\sum_{i=0}^{N}\left(t_{QF,q_i} + t_{L(V_{F(q_i)})} + t_{V_{F(q_i)},q_i}\right)}{N}$$

If there are N_k queries that are answered by a view V_k, the above formula can be decomposed into:

$$\overline{T}_{S1} = \frac{\sum_{i=0}^{N}\left(t_{QF,q_i} + t_{V_{F(q_i)},q_i}\right)}{N} + \frac{\sum_{k=0}^{M}\left(t_{L(V_k)} \times N_k\right)}{N}$$

where M is the number of views and $N = \sum_{k=0}^{M} N_k$

The first part of the formula is similar to the first part in the formula $\overline{T}_B$. The time for evaluating the query on a view is slightly better than evaluating the query on a document. The second part is beneficial when compared to $\overline{T}_B$ because the loading time of the views is very small when compared to the loading time of the document.

This solution is better than traditional method, but can be improved further. The query processing by QFilter and the evaluation of the query over the views remains the same but the second part can be enhanced. So we came up with a new solution which can improve the average query processing time.

Solution 2 – Dynamic view caching:

This new solution is a cost benefit model, which uses dynamic view caching. Here, we assume that a part of the memory space is allocated for caching and the rest is used as the working space.

Caching a view in memory saves the time required for loading the view every time a query needs to be answered by that view. This eliminates the loading time $\mathrm{t}_{\mathrm{L}(\mathrm{V}_{\mathrm{k}})}$ from $\mathrm{t}_{\mathrm{S1,i}}$. So the query processing time reduces to:

$$\mathrm{t}_{\mathrm{S2,i}} = \mathrm{t}_{\mathrm{QF},\mathrm{q}_{\mathrm{i}}} + \mathrm{t}_{\mathrm{V}_{\mathrm{F}(\mathrm{q}_{\mathrm{i}})},\mathrm{q}_{\mathrm{i}}}$$

So, the benefit of caching the view V_{k} is $\Delta\mathrm{t}_{\mathrm{i}} = \mathrm{t}_{\mathrm{S2,i}} - \mathrm{t}_{\mathrm{S1,i}} = \mathrm{t}_{\mathrm{L}(\mathrm{V}_{\mathrm{F}(\mathrm{q}_{\mathrm{i}})})}$

For every view V_{k} there is a benefit associated with it. The best of the views which give maximum benefit and improve the average query processing time are cached. All the queries reaching these views are answered immediately. For the rest of the queries whose views are not present in the cache we load the view at the time of answering the query into the working space that is available. We decide the views to be cached into memory based on the following three criteria: (i) cost (ii) benefit and (iii) decision criteria.

(i) Cost: The cost of caching a view in memory is a function of the size of the view. If the size of the view V_{k} is considered to be $\mathrm{S}_{\mathrm{V}_{\mathrm{k}}}$, then the cost of caching this view is $\mathrm{C}_{\mathrm{V}_{\mathrm{k}}} = \mathrm{C}(\mathrm{S}_{\mathrm{V}_{\mathrm{k}}})$. In most cases, the cost increases with the size. So it can be assumed to be a linear growth. However, there are certain scenarios where the cost varies based on certain additional factors involved (such as database – as – a – service scenarios, where the price of renting the resources may vary). In this thesis work, we consider the cost to be linear to size and the $\mathrm{C}_{\mathrm{V}_{\mathrm{k}}} = \mathrm{S}_{\mathrm{V}_{\mathrm{k}}}$. Selecting views with minimum cost and caching such views, will let a higher number of views to be cached into the memory.

(ii) Benefit: The benefit of caching a view is the amount of loading time saved by the usability of the view. It is a function of the loading time of the view: $\mathrm{b}_{\mathrm{V}_\mathrm{k}} = \mathrm{B}(\mathrm{t}_{\mathrm{L}(\mathrm{V}_\mathrm{k})})$ In this thesis work, we assume that $\mathrm{b}_{\mathrm{V}_\mathrm{k}}$is linear to the loading time, so $\mathrm{b}_{\mathrm{V}_\mathrm{k}} = \mathrm{t}_{\mathrm{L}(\mathrm{V}_\mathrm{k})}$. Assuming that there are N_k queries being answered by a view V_k, the total benefit of caching the view V_k in memory is $\mathrm{b}_{\mathrm{V}_\mathrm{k}} \times \mathrm{N}_\mathrm{k}$. When there is no restriction on the amount of memory available for caching then, selecting views with higher benefit to be cached is advantageous.

(iii) Decision criteria: If there is a limited resource available, and let's assume the maximum cost that can be afforded (cached) is $\mathrm{C}_{\mathrm{max}}$, then, we can only select a few views to be cached in memory such that $\sum_\mathrm{k} \mathrm{C}_{\mathrm{v}_\mathrm{k}} \leq \mathrm{C}_{\mathrm{max}}$. We want to select a subset of the views to be cached in the memory such that the total cost is below the threshold $\mathrm{C}_{\mathrm{max}}$ and the total benefit is maximized. That is, we want identify a view loading vector $\mathrm{L} = \mathrm{l}_1, \mathrm{l}_2, \dots, \mathrm{l}_\mathrm{m}$, where $\mathrm{l}_\mathrm{k} = 1$, the view is loaded into the memory and $\mathrm{l}_\mathrm{k} = 0$, the view is materialized and is on the hard drive but not loaded into the memory, which maximizes the total benefit:

$$\mathrm{B} = \sum_{\mathrm{i}=0}^{\mathrm{N}} \Delta \mathrm{t'}_\mathrm{i} = \sum_{\mathrm{i}=0}^{\mathrm{N}} \mathrm{b}_{\mathrm{V}_\mathrm{k}} \times \mathrm{N}_\mathrm{k} \times \mathrm{l}_\mathrm{k}$$

$$\text{and}$$

$$\mathrm{C} = \sum_{\mathrm{k}=0}^{\mathrm{M}} \mathrm{C}_{\mathrm{V}_\mathrm{k}} \times \mathrm{l}_\mathrm{k} \leq \mathrm{C}_{\mathrm{max}}$$

Case 1: When we know the query pattern

When the query pattern is known, we know the number of queries being answered by a view. Let's assume that N_k number of queries are answered by a view V_k. The cost of each view V_k

is $\mathrm{C}_{\mathrm{V}_\mathrm{k}}$, where $\mathrm{C}_{\mathrm{V}_\mathrm{k}} = \mathrm{S}_{\mathrm{V}_\mathrm{k}}$ and the benefit of each view V_k is $\mathrm{b}_{\mathrm{V}_\mathrm{k}}$, where $\mathrm{b}_{\mathrm{V}_\mathrm{k}} = \mathrm{t}_{\mathrm{L}(\mathrm{V}_\mathrm{k})}$ (assuming no additional costs).

For every view the cost benefit ratio is defined as:

$$\mathrm{CBR} = \frac{\mathrm{b}_{\mathrm{V}_\mathrm{k}} \times \mathrm{N}_\mathrm{k}}{\mathrm{C}_{\mathrm{V}_\mathrm{k}}}$$

This gives the benefit for caching each byte of the view. An optimal solution is to find the best view to be loaded into memory that: (1) provides the maximum cost benefit ratio, i.e. the one which gives the maximum benefit per unit cost. (2) The cost is affordable, i.e. $\sum_\mathrm{k} \mathrm{C}_{\mathrm{v}_\mathrm{k}} \leq \mathrm{C}_{\mathrm{max}}$.

This is similar to the knapsack problem where, we have to choose the items with the maximum value to add to the knapsack but at the same time have to take care of the weight of the items such that the combined weight doesn’t affect the knapsack. We need to select the best views with the maximum benefit and observe the cost not to over exceed the available.

An algorithm to our problem in the knapsack version is as follows:

Algorithm:

```
for i from 0 to M
    T[i,0] = 0
for j from 0 to $\mathrm{C}_{\mathrm{max}}$
    T[0,j] = 0
for i from 1 to M
    for j from 1 to $\mathrm{C}_{\mathrm{max}}$
        if j $\geq \mathrm{C}_{\mathrm{V}_\mathrm{k}}$[i] then
```

```
            $T[i, j] = \max( T[i-1, j],\ CBR_{V_k}[i] + T\left[i-1, j - C_{V_k}[i]\right])$
            if $\left( CBR_{V_k}[i] + T\left[i-1, j - C_{V_k}[i]\right]\right) > T[i-1, j]$ then
                select[i][j]=1
            else
                select[i][j]=-1
        else
            $T[i, j] = T[i-1, j]$
i=M
j= $C_{max}$
while(i>0):
    if (select[i][j]==1)
        load view $C_{V_k}[i]$
        i=i-1
        j=j-CBR[i]
    else
        i=i-1
```

But the knapsack problem is NP – complete and the algorithm defined above is a pseudo polynomial time algorithm. The time complexity of this algorithm is O(MC_{max}) where M is the number of views and C_{max} is the maximum available memory. So we use the greedy approximation algorithm instead to solve the problem.

The greedy algorithm is defined as follows:

Algorithm:

```
$C_{total} = 0$
CBR[] = list of CBR values for all views
Sort(CBR) //sort in descending order
While isNotEmpty(CBR)
    Pop(CBR)
    Get corresponding $C_{V_k}$for the popped CBR
    If $C_{total} + C_{V_k} \leq C_{max}$ then
        $C_{total} = C_{total} + C_{V_k}$
        Load $V_k$ into memory
        Set $l_k$ to 1
```

Case 2: When we do not know the query pattern.

(1) Assume that there is a uniform distribution. For a unit cost, a certain number of queries N' are answered. The higher the cost C_{V_k}, the more the number of queries that are answered by the view. This means that number of queries being answered is linearly proportional to C_{V_k}. Therefore, we can consider the total number of queries answered by a view V_k to be $N_k = N' \times C_{V_k}$ where N' is the number of queries answered by unit cost. The cost benefit ratio is now modified as:

$$CBR = \frac{b_{V_k} \times N' \times C_{V_k}}{C_{V_k}}$$

$$\mathrm{CBR} = \mathrm{N}' \times \mathrm{b}_{\mathrm{V_k}}$$

Here N' is a constant. Therefore, CBR solely depends on $\mathrm{b}_{\mathrm{V_k}}$. The maximum $\mathrm{b}_{\mathrm{V_k}}$ gives the maximum CBR. $\mathrm{b}_{\mathrm{V_k}}$ is proportional to $\mathrm{C}_{\mathrm{V_k}}$, so selecting the maximum $\mathrm{C}_{\mathrm{V_k}}$ would give the best cost benefit ratio.

Algorithm:

```
$\mathrm{C}_{\mathrm{total}} = 0$
While $\mathrm{C}_{\mathrm{total}} \leq \mathrm{C}_{\mathrm{max}}$
    Get max $\mathrm{C}_{\mathrm{V_k}}$
    If $\mathrm{C}_{\mathrm{total}} + \mathrm{C}_{\mathrm{V_k}} \leq \mathrm{C}_{\mathrm{max}}$ then
        $\mathrm{C}_{\mathrm{total}} = \mathrm{C}_{\mathrm{total}} + \mathrm{C}_{\mathrm{V_k}}$
        Load $\mathrm{V_k}$ into memory
        Set $\mathrm{l_k}$ to 1
```

(2) Assume that every view answers equal number of queries.

In this case we assume that every view answers equal number of queries irrespective of the size/cost of the view. Let the number of queries answered by each view be N'', then the cost benefit ratio of the view will be:

$$\mathrm{CBR} = \frac{\mathrm{b}_{\mathrm{V_k}} \times \mathrm{N}''}{\mathrm{C}_{\mathrm{V_k}}}$$

Here, N'' is a constant. So, the cost benefit ratio is just a ratio of $\mathrm{b}_{\mathrm{V_k}}$ and $\mathrm{C}_{\mathrm{V_k}}$. $\mathrm{b}_{\mathrm{V_k}}$ is the time taken to load the view and it increases as the cost of the view increases. This increase is not particularly linear. It can be an exponential rise too. So we cannot conclude that the CBR is going to be a constant in all cases.

Assuming that the increase is not linear and the CBR is not a constant, an ideal solution is to load views which have (1) the maximum cost benefit ratio (CBR) and (2) the sum of the costs of the views is affordable i.e. $\sum_k C_{v_k} \leq C_{max}$. We use the greedy algorithm that we defined earlier to select the best views to load into memory.

When the CBR is constant, it implies that all the views are equally beneficial. In such cases, we load the views in order until the cost of all the loaded views reaches C_{max}.

CHAPTER 5

EXPERIMENT RESULTS

To demonstrate the effectiveness of our approach, we first construct a QFilter based on the access control rules. Then, the input queries are processed by the QFilter to generate safe queries. All the safe queries are outputted along with the number of the view that answers the query. We create views based on the accept states (access control rules) in the QFilter NFA structure and use these views when evaluating the queries. We test the loading time, query processing time and the total benefit for different solutions.

We used the XMark (an XML benchmark project [51]) to generate the XML documents. We used a collection of documents of varying sizes for our experiments. We used Galax 1.0 [34] for XPath query evaluation. QFilter implementation in Java was modified for our purpose and the results were transferred to Galax through its Java API. We used different sets of rules, each rule set containing minimum of 10 rules.

Rule set 1

R1 {role1, /site/regions/europe/*/description//*, read, +}

R2 {role1,/site/closed_auctions/closed_auction/quantity,read, +}

R3 {role1, /site/people/person//*, read, +}

R4 {role1, /site/open_auctions/open_auction/privacy, read, +}

R5 {role1, /site/open_auctions/open_auction/reserve, read, +}

R6 {role1, /site/closed_auctions/closed_auction/date, read, +}

R7{role1,/site/closed_auctions/closed_auction/annotation/author,read,+}

R8 {role1, /site/regions/namerica//*, read, +}

R9 {role1, /site/regions/asia//*, read, +}

R10 {role1, /site/categories/category//*, read, +}

Rule set 2

R1 {role2, /site/categories/category//*, read, +}

R2 {role2, /site/catgraph/edge, read, +}

R3 {role2, /site/closed_auctions/closed_auction/price, read, +}

R4 {role2, /site/people/person//*, read, +}

R5 {role2, /site/closed_auctions/closed_auction/type, read, +}

R6 {role2, /site/open_auctions/open_auction/initial, read, +}

R7 {role2, /site/regions/namerica//*, read, +}

R8 {role2, /site/open_auctions/open_auction/privacy, read, +}

R9 {role2, /site/open_auctions/open_auction/seller, read, +}

R10 {role2, /site/regions/africa/item//*, read, +}

We first construct the two different QFilters using the above rule sets. The recorded QFilter construction time is 122ms and 114ms respectively. The QFilter construction is fast enough. The construction time mainly depends on the XPath expression in the rules. In case of a multi role

QFilter, one single NFA structure is constructed for multiple roles as opposed to multiple QFilters i.e. one for each role. The construction time is greatly reduced because of this.

After the construction of the QFilter, queries are filtered to collect the safe queries. We use three different query sets of varying lengths: 250, 500 and 1000 and filter them using the QFilter. The total filtering time for the three different query sets using different rule sets are shown in the following table:

	Rule set 1	Rule set 2
Query set 1 (250 queries)	2.45699ms	2.25699ms
Query Set 2 (500 queries)	3.37099ms	2.55799ms
Query Set 3 (1000 queries)	5.68099ms	5.46299ms

Table 2

The filtering time is usually affected by the occurrence of wildcard character like * or //. The above three query sets have equal probability of * and //. The rule set 1 has a higher probability of * and // when compared to rule set 2. The filtering time depends not only on the total number of queries but also on the number of queries to be rewritten. From the experimental results in [9] [10], we can see that the QFilter filtering time is fast. The table below gives the number of queries accepted, rewritten and denied in each of the three query sets for the two rule sets.

	RS1			RS2		
	Accept	Rewrite	Deny	Accept	Rewrite	Deny
QS1	127	2	121	109	3	138
QS2	256	1	243	223	3	274
QS3	461	10	529	478	7	515

Table 3

After the construction and execution of the QFilter, the queries are then evaluated. We performed different sets of experiments for all the different approaches – baseline approach, using fine grained views and dynamic view caching and compare the performance of all the solutions.

Base line approach:

In the baseline approach, we used the three query sets mentioned above and evaluated each set on a collection of documents of different sizes: 52MB, 43MB, 25MB, 11MB and 5MB. We make two assumptions here: firstly we assume that there is enough memory available to load the documents (one at a time); and secondly the entire memory space available is considered to be the working space i.e. a document is loaded to answer a query, then it is cleared from the memory to load the next document and answer the next query and so on. The cost and loading time for each of the documents is shown in figure 5.

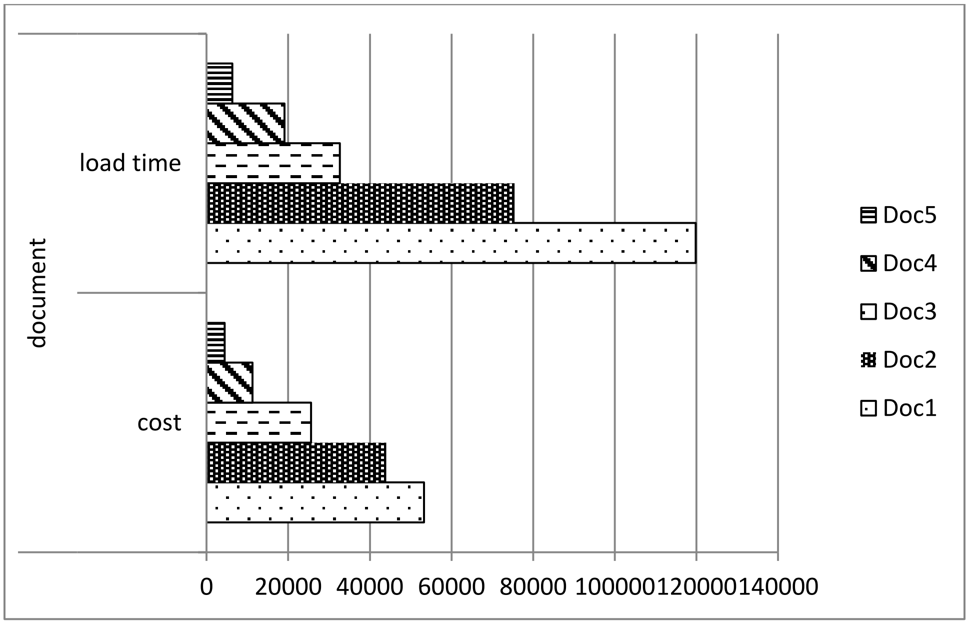


Figure 5

The QFilter construction and evaluation are independent of the document size, but the average query evaluation and the loading time of the document increases linearly with the document size. We can presume this from Figure 5 and 6.

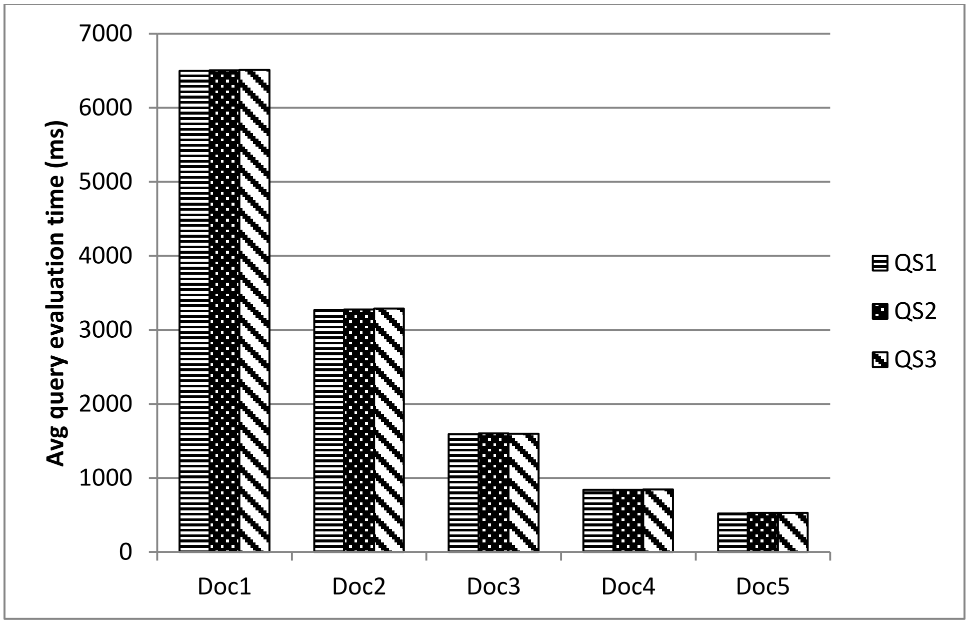


Figure 6: Average query evaluation time for different documents

From figure 6, we can see that there is a slight difference in the evaluation time between different query sets. The three query sets used in this experiment have equal probability of wild card characters and the rule sets used have 40% to 50% query acceptance rate. So we can deduce that the slight difference between the query evaluations of different query sets is because of (i) the number of queries in the query set (ii) the type of queries. We used rule set 1 for the above experiment.

To evaluate the documents, the required document is first loaded into memory and then the query is answered. Each document might answer a certain number of queries, but the disadvantage of the baseline approach is that we load the document to answer the query and the next query that comes in might require another document to answer it. So the current document is removed from the memory to make space for the next document. This is repeated till all the incoming queries are answered. The average end to end query processing includes the time taken to process the query by the QFilter, the time taken to evaluate the query on the document and the loading time of the document. The average end to end query processing time rapidly increases in a baseline approach. From our experiments the average end to end processing time, considering the collection of 5 documents to answer the three query sets is shown in figure 7.

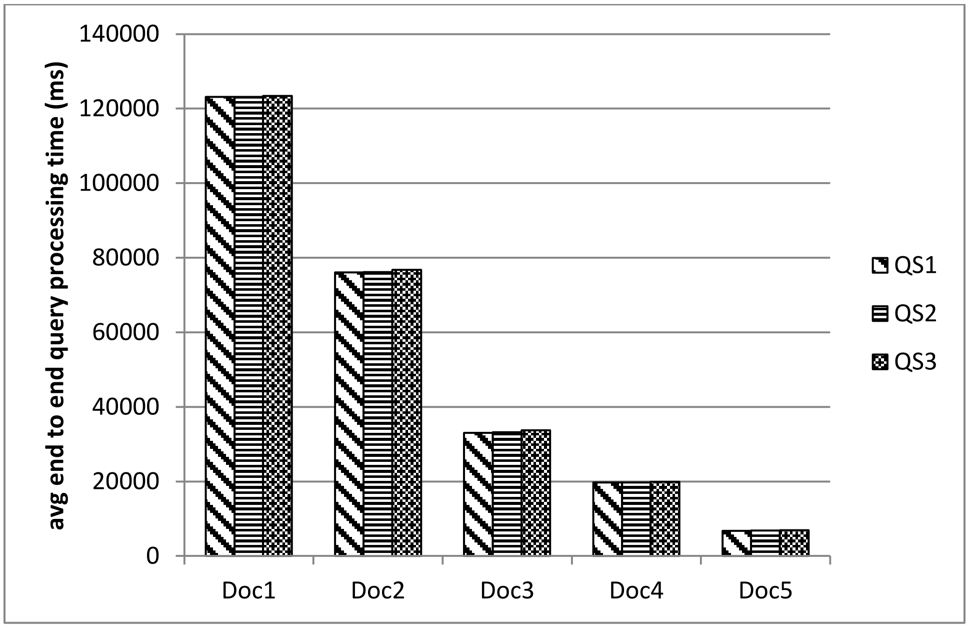


Figure 7: Average end to end query processing time for every document

The average end to end query processing time for all the documents together for different rule sets and query sets are shown in figure 8. The highest processing time for query set 3 is around 53000ms.

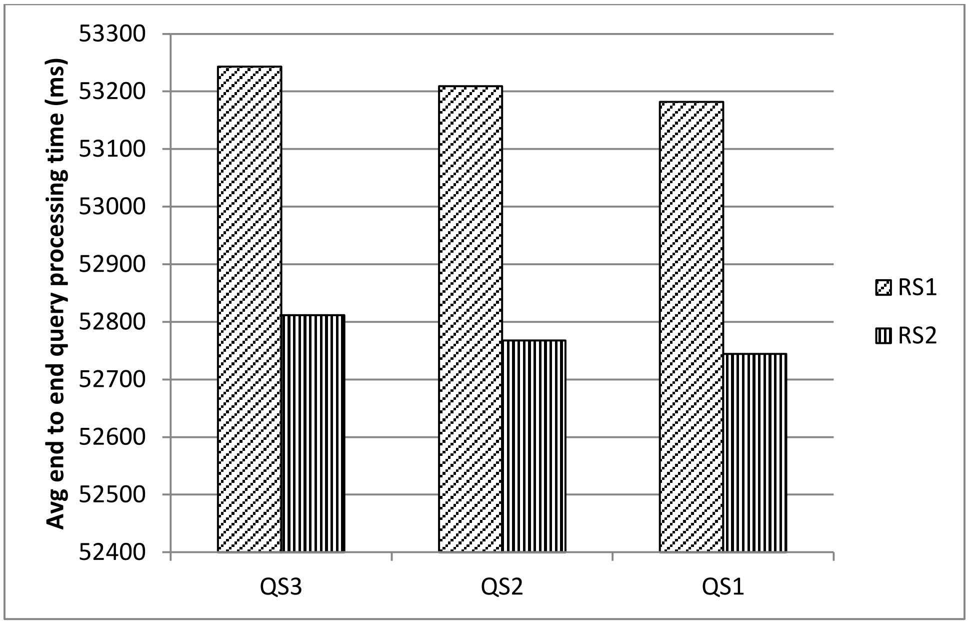


Figure 8: Average end to end query processing time for all documents combined

When the memory space is not sufficient to load the documents, then a part of the document is loaded in order to answer the query. We do not know which part of the document to load to answer the query. So, we divide the document into parts and load each part to memory one at a time to answer the query. This is repeated until the query is answered. The average query evaluation time in this case increases with the number of documents, the number of parts a document is divided into and the time to find the part that answers the query. The worst case situation includes where all the parts of the document are to be loaded to answer the query. The average end to end query processing time degrades because of the brute force technique to find the part of the document in this approach.

Fine grained view approach:

When the memory available is limited and it is difficult to load the document, we suggest the use of views. From the above experimental results, we know that the baseline approach is not advantageous. In this solution, we create fine grained views, which are easy to load into memory when there is limited memory available. Based on the accept states in the QFilter NFA structure we create views for every accept state. To answer a query, a view is loaded into memory and the query is evaluated on the view. The loading time of the views increases with the view size. The cost and the loading time of the views generated by different rule sets are shown in figure 9.

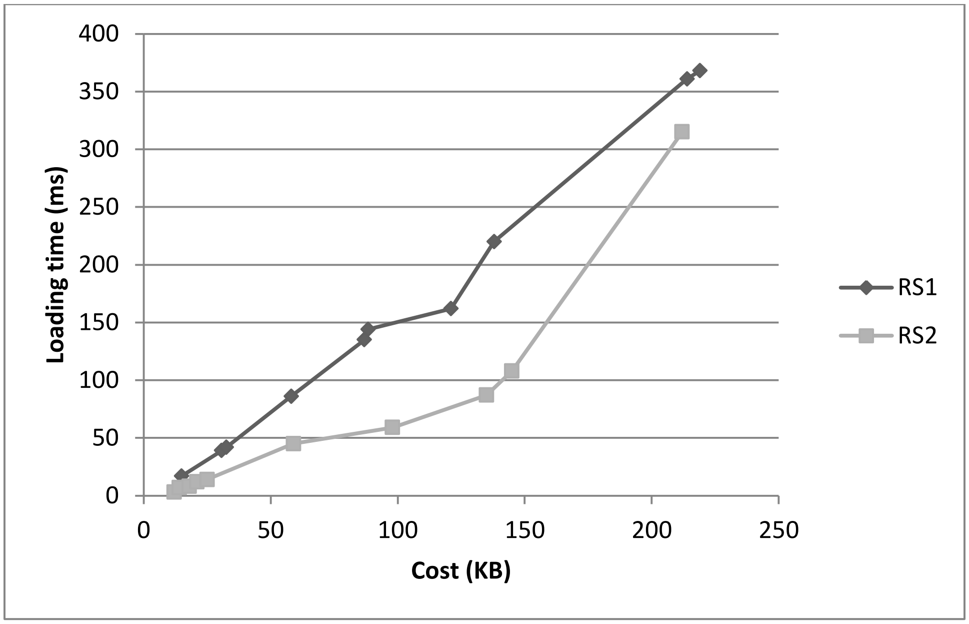
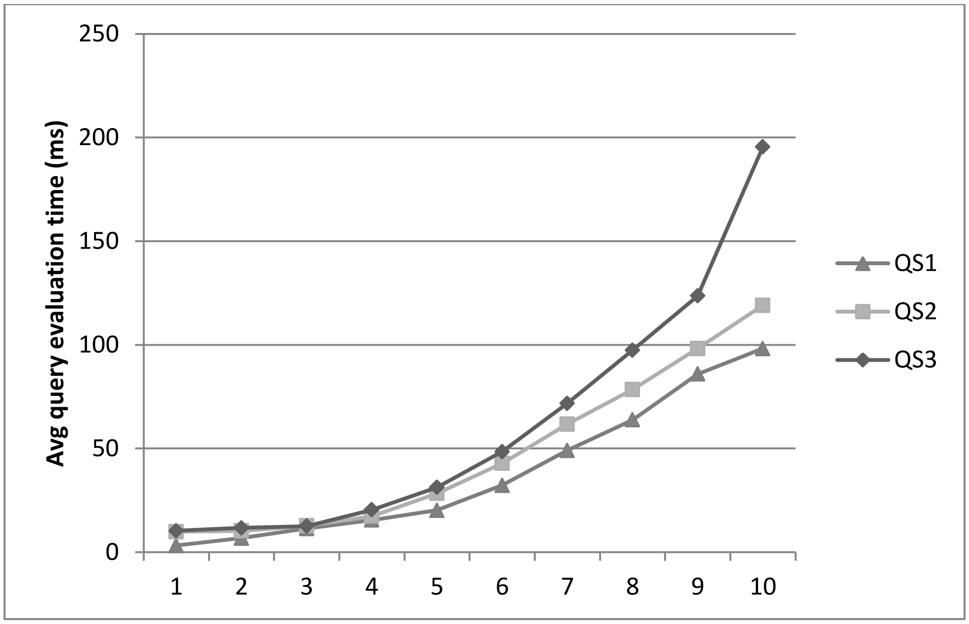
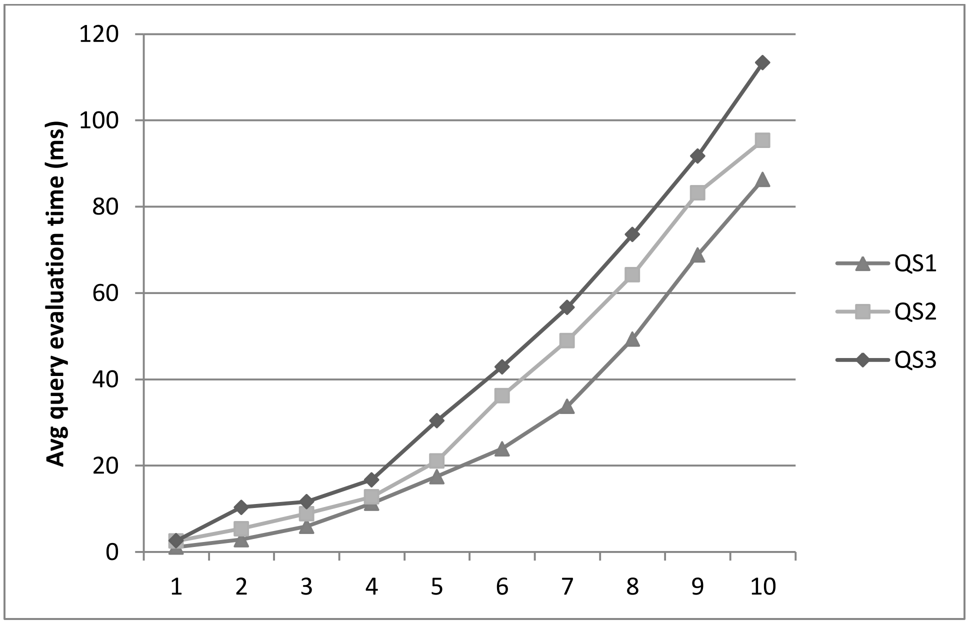


Figure 9: view loading time for different rule sets

We used the same three sets of queries to calculate the average query evaluation time. The results of evaluation of the three query sets for two rule sets are shown in figure 10. From figure 10, we can deduce that the average query evaluation time for each view (i) changes with the increase in the number of queries in the query set: lesser number of queries improves the evaluation time (ii) views with maximum size increase the evaluation time (iii) comparing the two graphs in figure 10, we see that rule set 2 has better evaluation time than rule set 1. The lower occurrence of wildcard characters like "*" and "//" enhances the evaluation time.

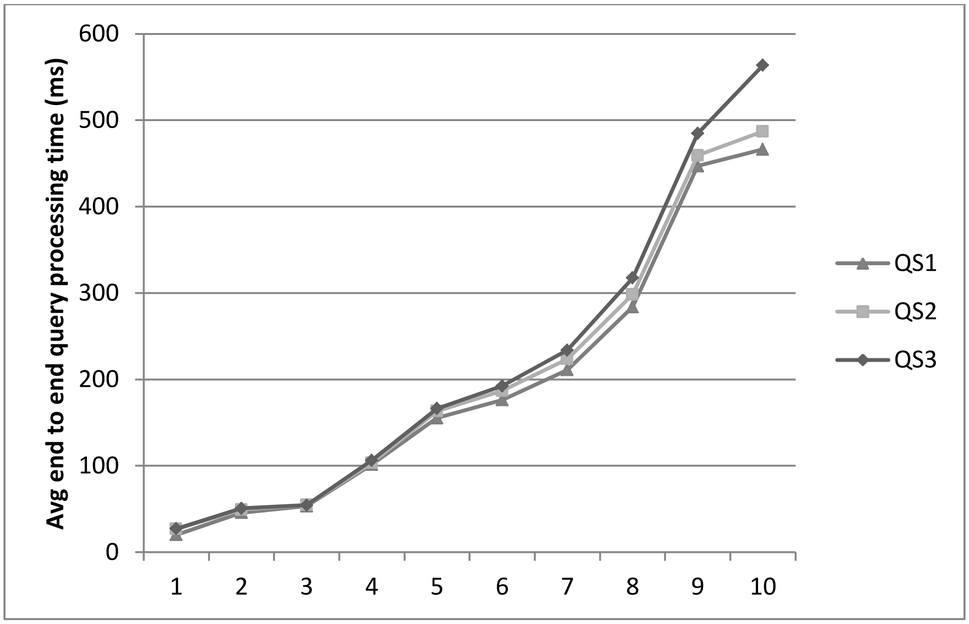


Query evaluation time for rule set 1

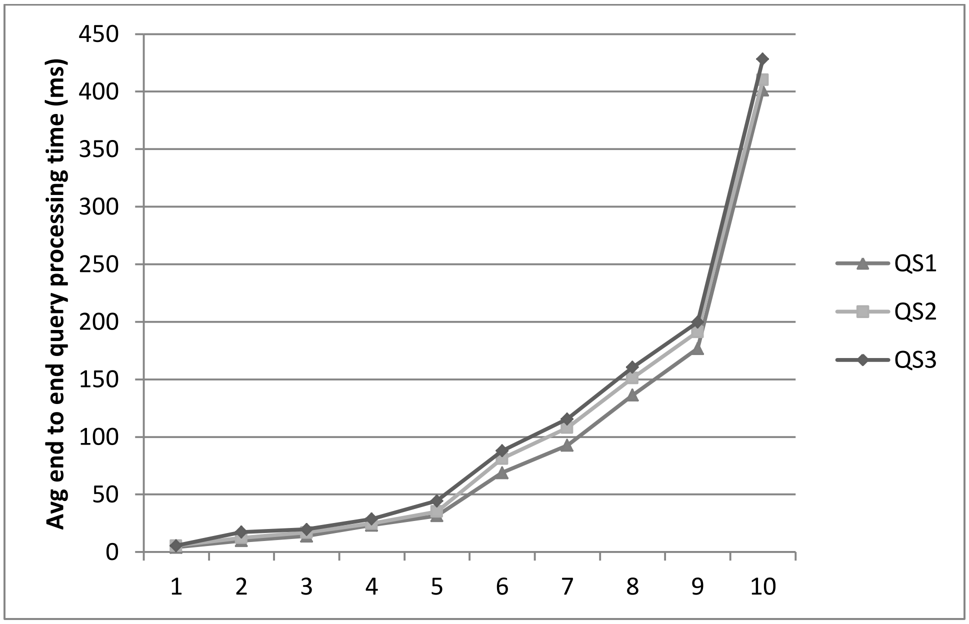


Query evaluation time for rule set 2

Figure 10



Average end to end query processing time for each view – Rule set 1



Average end to end query processing time for each view – Rule set 2

Figure 11

The average end to end query processing time includes the time taken by the QFilter to process the query, the time taken to load the view and the evaluation time of the query over the view.

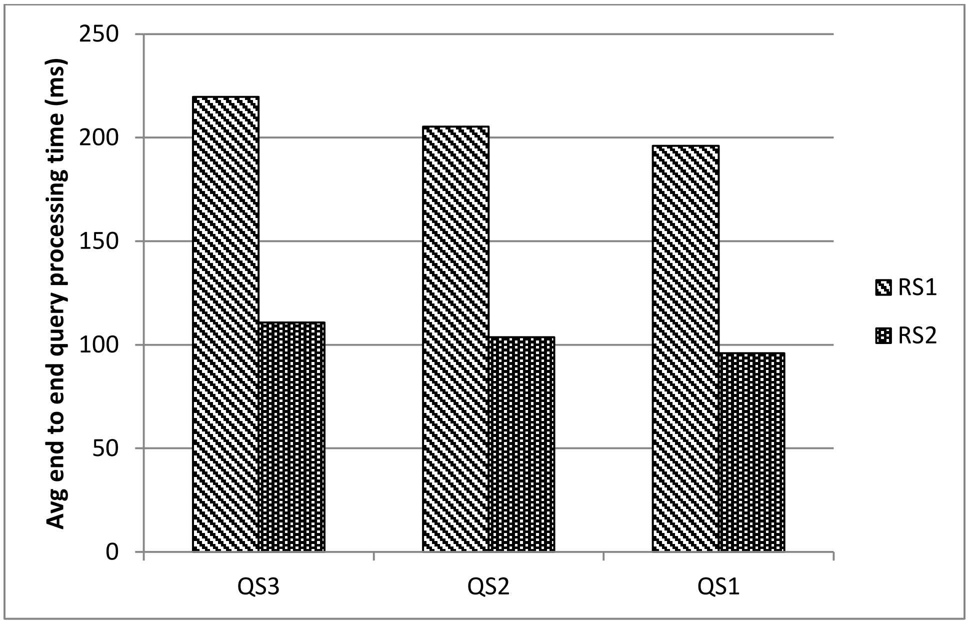


Figure 12: Average end to end query processing time for all views

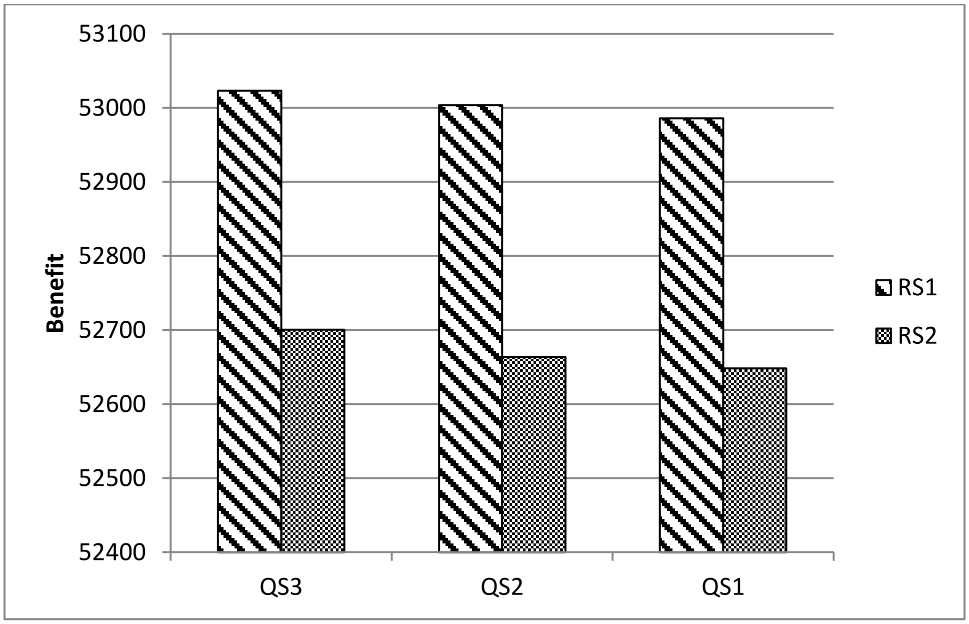


Figure 13: Total benefit obtained by using views

The average end to end query processing time for each of the views using different query sets and rule sets is shown in figure 11. And the average end to end query processing time for all the views combined together is shown in figure 12. This is greatly improved when compared to the baseline approach (comparing figure 12 to figure 8). We can find that the average query processing time for each of the query sets when using the documents is a lot higher because the time taken to load a document is comparatively much higher than the time taken to load the views. The overall benefit of using fine grained views over documents can be seen in figure 13.

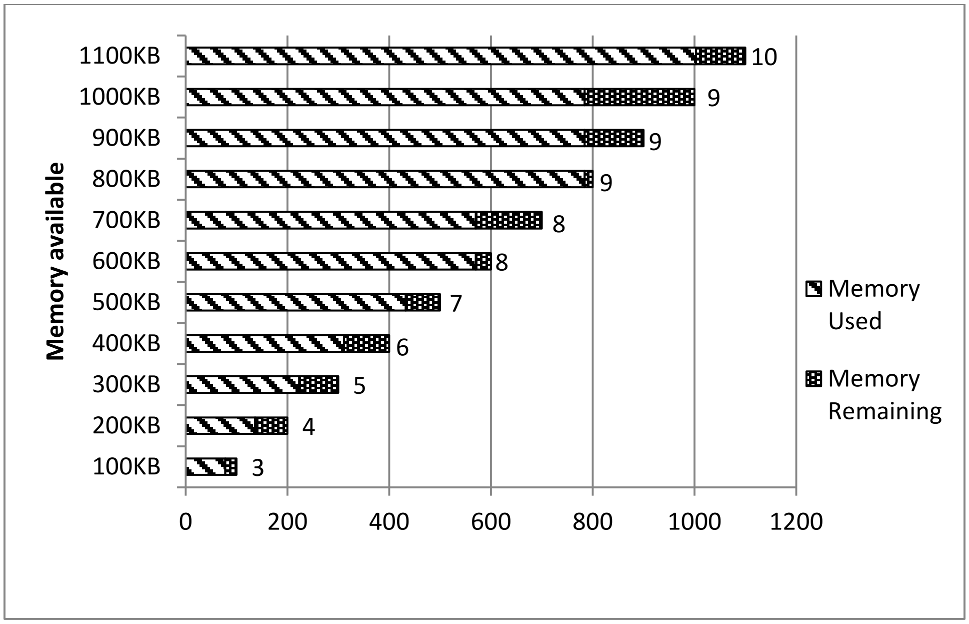
The fine grained view approach is beneficial when compared to the baseline approach and improves the overall end to end query processing. But as discussed in chapter 4, we extend our research to find a better solution that further improves the query processing.

Dynamic view caching approach:

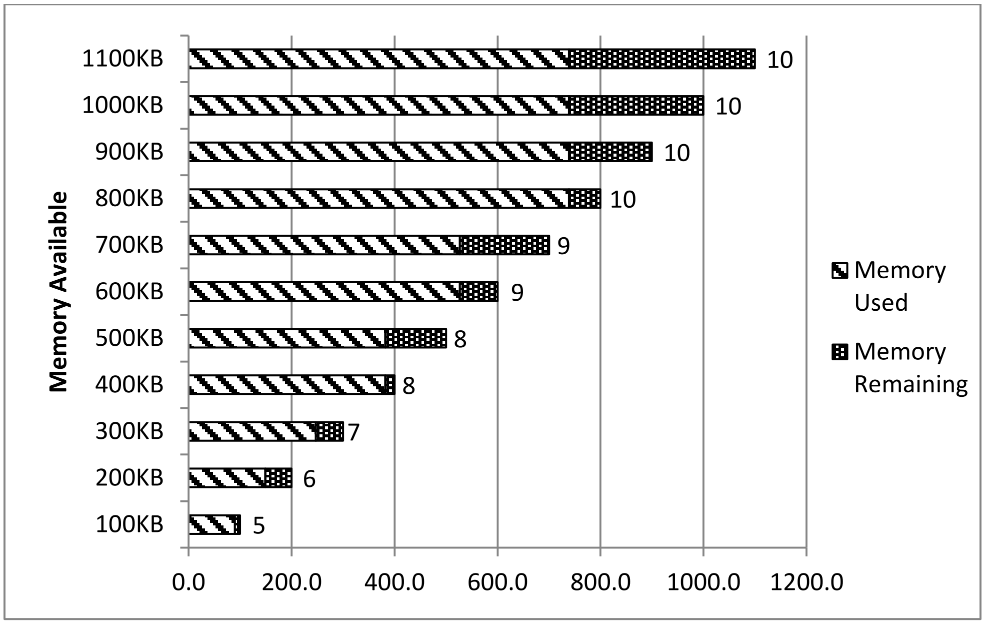
In this approach, we divide the memory into two parts: the working space and the cache memory. We select the views with improved benefit and affordable cost and store them in the cache memory. The working space is used to load the other views (views which are not cached) on the fly to answer queries. Caching the views improves the overall query processing. It eliminates the most time consuming process of loading views every time a query needs to be answered. A huge amount of benefit is attained by using caching. Assuming that the cache memory is limited, we consider three important aspects when deciding which views to cache. (i) cost (ii) benefit (iii) decision criteria.

For a cost effective solution, we consider the views starting with the least cost and store them in the cache memory until it’s full. Greedy algorithm is used to select the views with minimum cost at any moment. We examine the storage of views for different sizes of cache memory availability and calculate the average end to end query processing for each of the cases.

Figure 14 shows the amount of memory used and the amount of free memory available for different variations of cache memory for views constructed from two rule sets. It also shows the number of views being cached depending on the availability of the memory. From the figure, we can observe that this approach tries to fill in the memory space with the maximum number of views that can fit the memory at any given time. The cost effective solution is useful in cases where the query pattern is not known and we assume that loading more number of views into the memory will answer a larger number of queries and will provide more benefit.

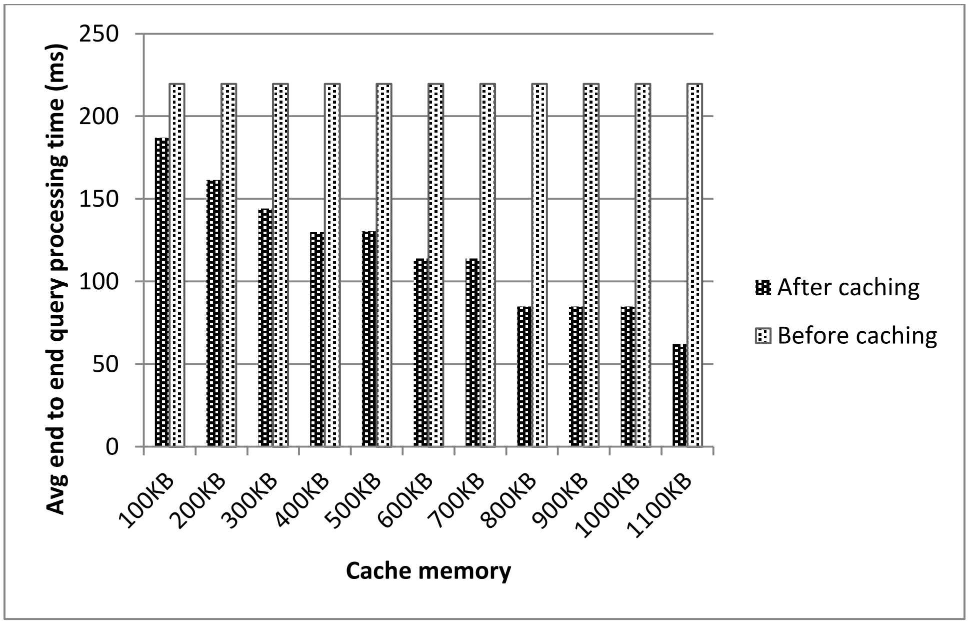


Memory usage for views created from rule set 1

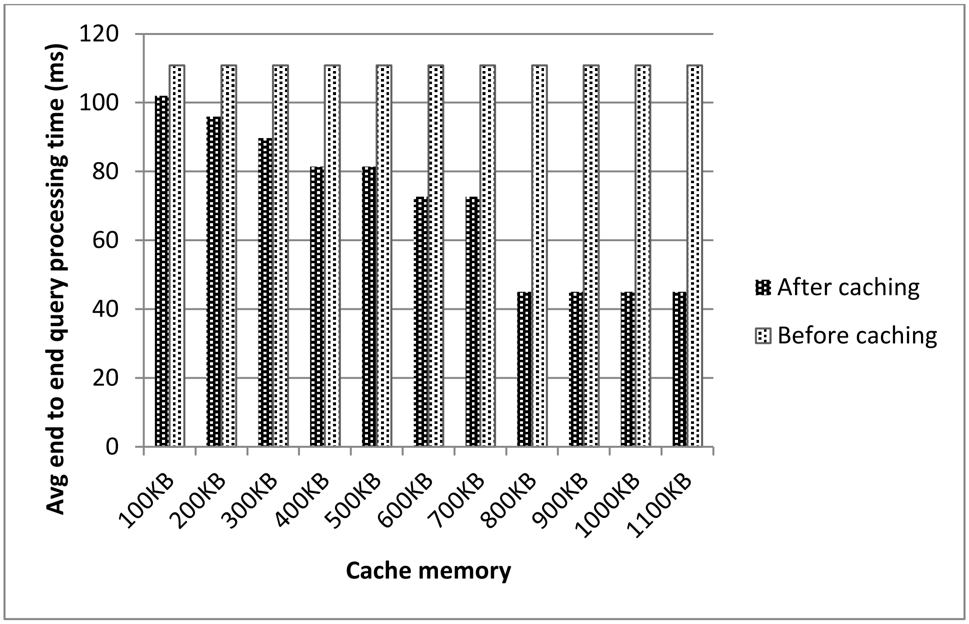


Memory usage for views created from rule set 2

Figure 14



(a) Rule set 1

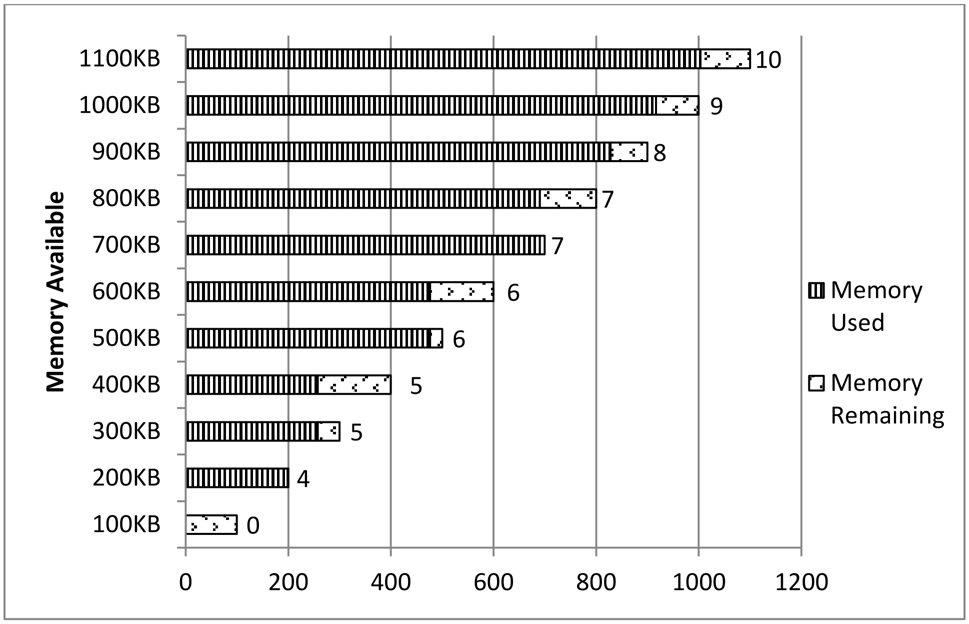


(b) Rule set 2

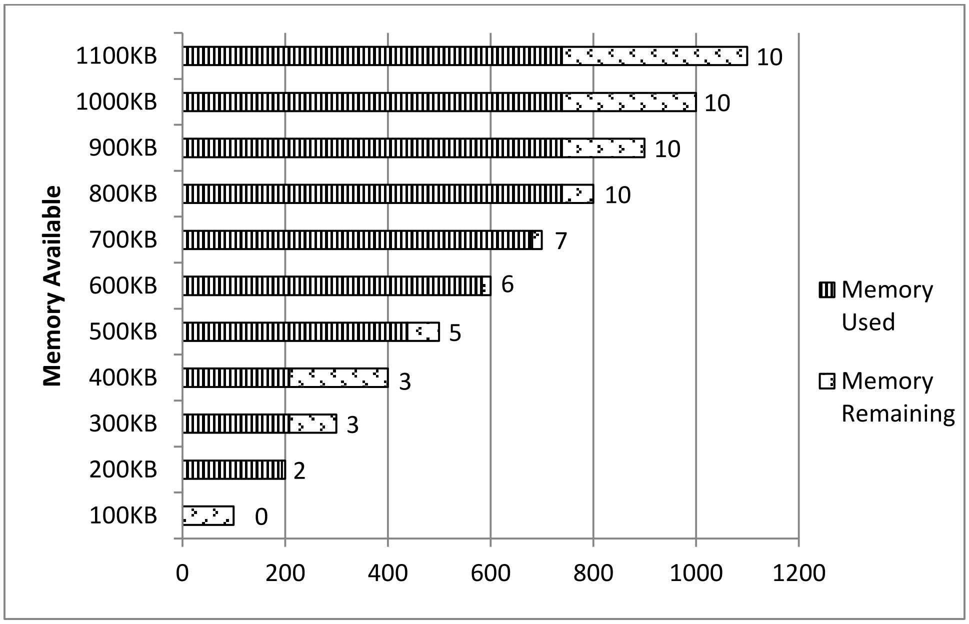
Average end to end query processing time before and after caching for views created from

Figure 15

Figure 15 shows the average end to end query processing time before and after caching views for two sets of views constructed from the two rule sets. We use the query evaluation times for query set 3 in this experiment. When the views are loaded on the fly one at a time to answer each query, the average end to end query processing time is greater because it includes the average loading time of the views. After caching the views, the average end to end query processing time decreases dramatically because the views are already in memory so, the loading time is eliminated. The average end to end query processing time for all the views without using caching remains constant. It is the value calculated when the views are being loaded into memory one at a time to answer the queries.



Memory usage for views using rule set 1



Memory usage for views using rule set 1

Figure 16

The average end to end query processing time improves as larger amount of cache memory is available because higher number of views are cached and it decreases the average. The total benefit increases with the increase in cache memory space. The difference between the before caching value and the after caching value in figure 15 gives the benefit obtained by caching the views.

For the benefit only solution, we cache the views that give the most benefit keeping the cost in the affordable range. The benefit is the amount of time saved by caching the view in memory.

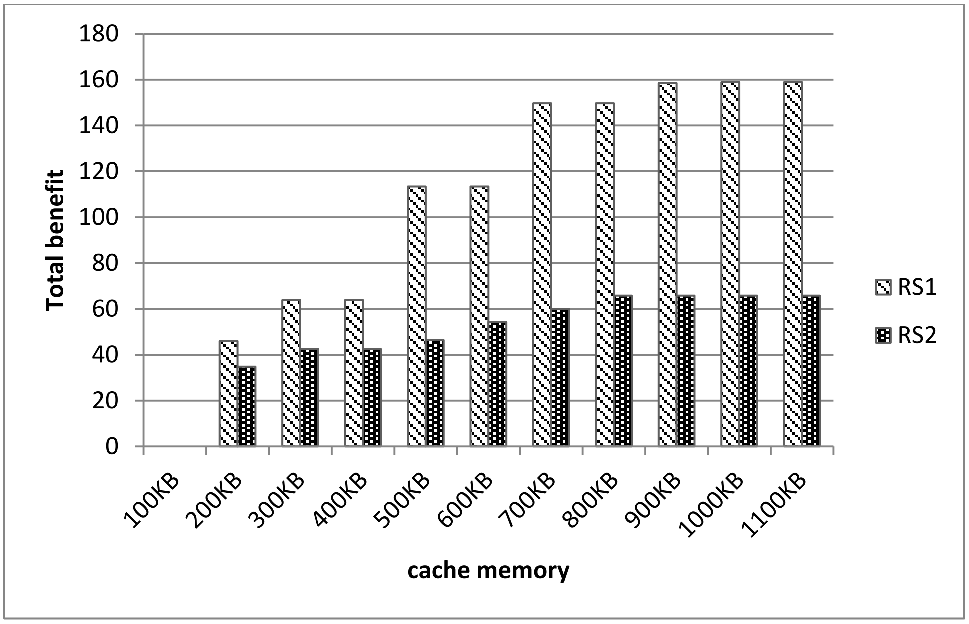


Figure 17: Benefit obtained by caching views in rule set 1 and rule set 2

Figure 16 shows the memory usage for different variations of available cache memory for two sets of views constructed from two rule sets. It also shows the number of views being loaded into memory in each instance. The used memory in the figure indicates that a certain number of views are loaded into memory and the free space is because no other view can be loaded into this space.

The benefit obtained by caching the views constructed from two rule sets for varying availability of cache memory is shown in figure 17. The benefit attained by rule set 1 is much higher when compared to rule set 2. This is because, the views in rule set 2 are small and their cost is minimal when compared to rule set 1. Larger views tend to contribute more towards the benefit because the time to load the view every single time is eliminated from the average.

The average end to end query processing time for varying cache memory space for both the rule sets are shown in figure 18.

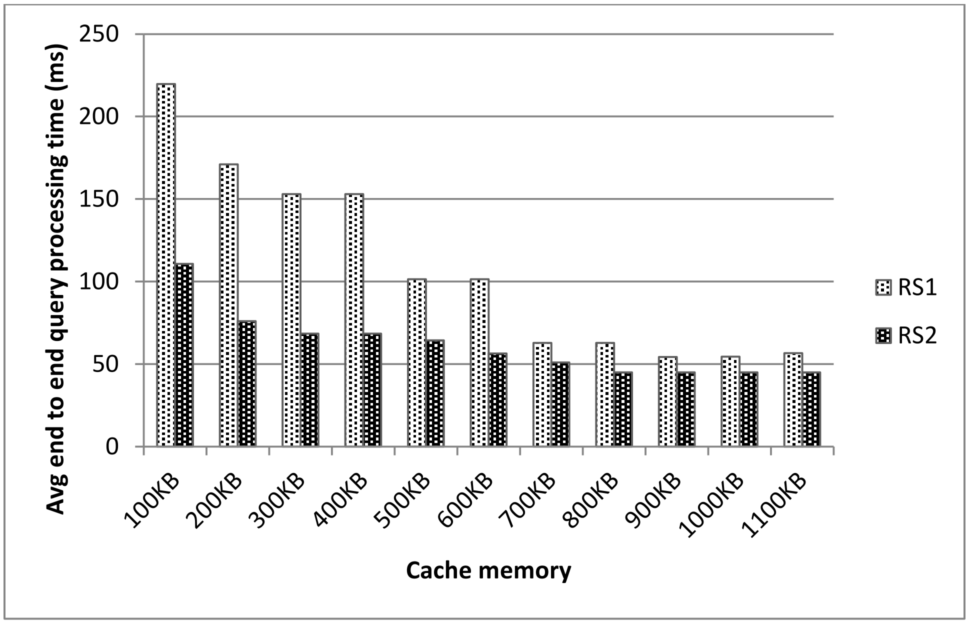
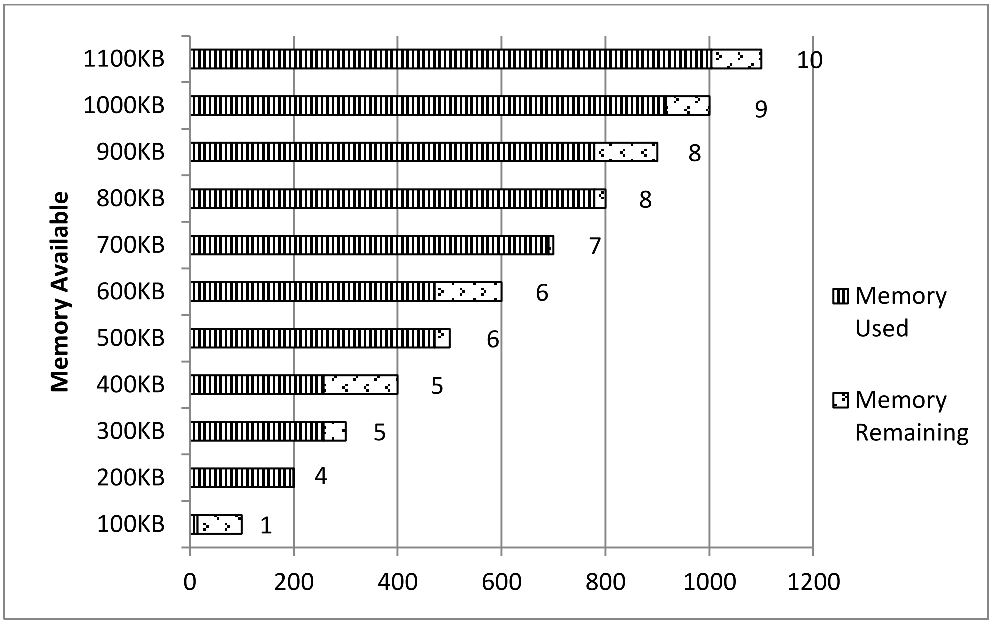
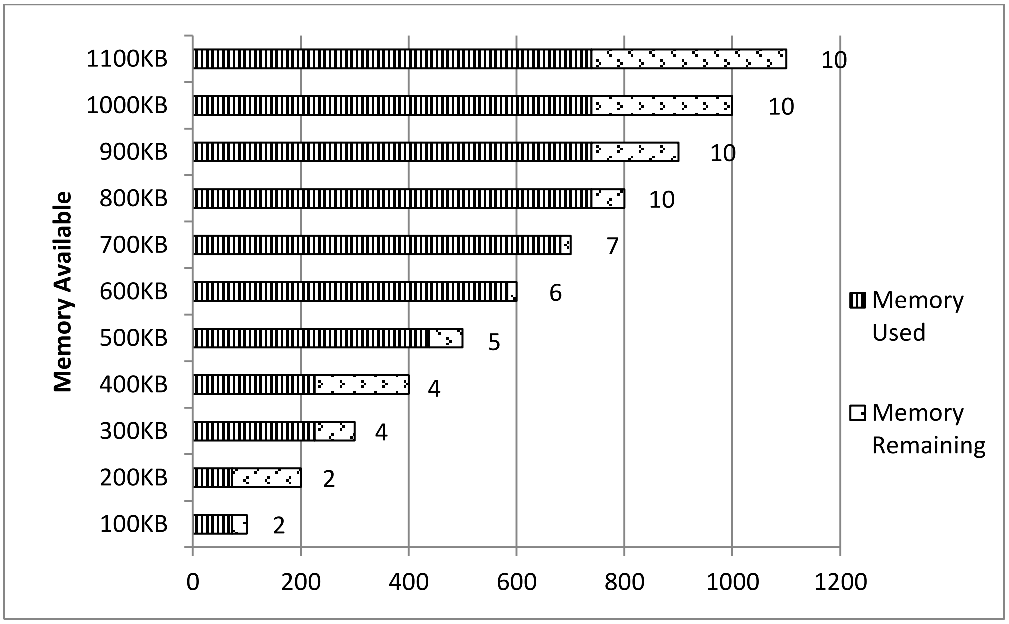


Figure 18: Average end to end processing time for two rule sets

After testing the cost effective and benefit only methods, we move on to the next case that is the decision criteria. Here we calculate the cost benefit ratio for the views and select the views with the best cost benefit ratio. We use the same set of views (constructed from two rule sets) that we used for the previous experiments.



Memory usage for views using rule set 1



Memory usage for views using rule set 2

Figure 19

The query distribution for each of the views is known when we know the query pattern. The total benefit is given by the product of the loading time of the view that is cached and the number of queries that are answered by the query i.e. the time that is being saved by caching the view. The cost benefit ratio gives the benefit obtained per unit cost. We use the greedy method to load the views with the maximum cost benefit ratio keeping the total cost below the maximum available memory. Figure 19 gives the memory usage for the different views created from the two rule sets.

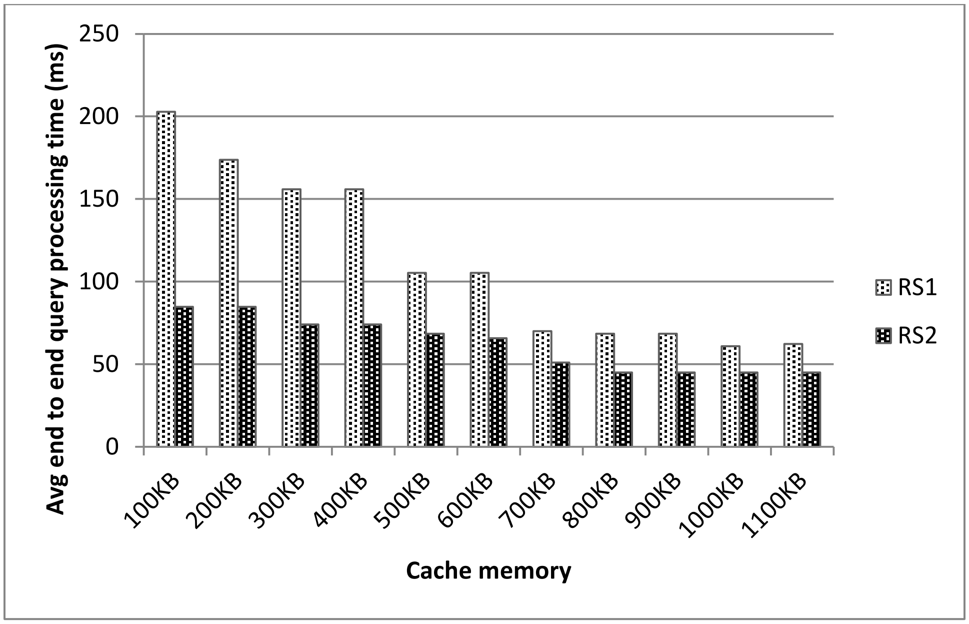


Figure 20: Average end to end query processing time for varying memory availabilities

Figure 20 and 21 show the average end to end query processing time and the total benefit attained by this method respectively. As before, the average end to end query processing time decreases as higher number of views are cached in memory. And, the total benefit increases.

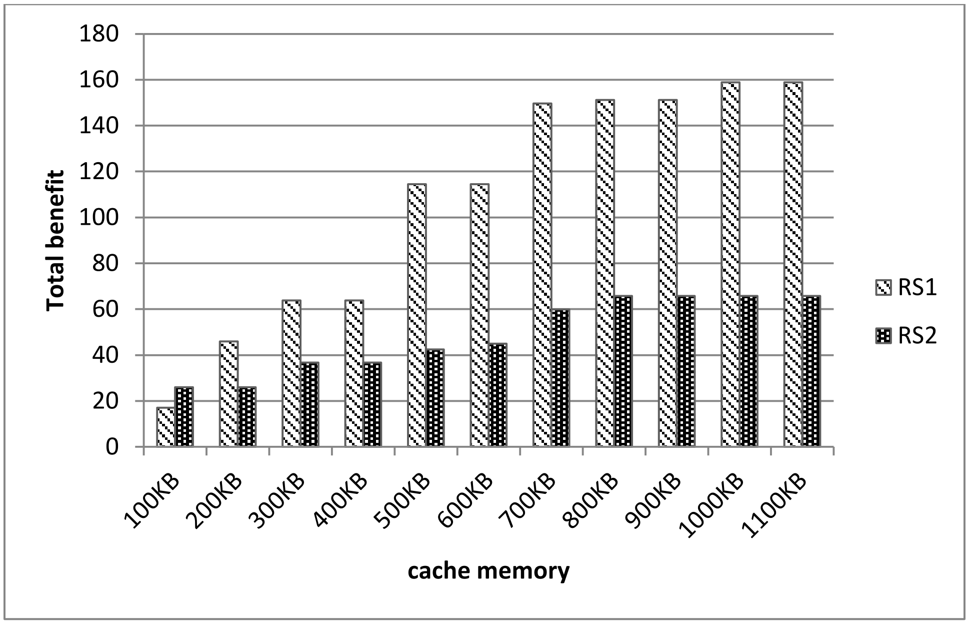


Figure 21: Total benefit obtained by caching views for various cache memory availabilities

The baseline approach is the traditional pre – processing approach that is quite popular. From our experiments we can say that the query performance in the baseline approach is greatly dependent on the size and the number of documents that are being used. Also, in our experiments we assume that there is enough memory space to load one document at a time. But in most cases, the memory available might not be sufficient to load all the documents required. The larger documents have a higher loading time and the query evaluation time also increases. So we can say that the HyXAC model is advantageous when compared to the pre-processing method, as it eliminates the higher loading times of the documents.

Comparing HyXAC to the traditional view based method: The views created for every role are generally larger when compared to the fine grained views used in HyXAC. We created views for roles 1 and 2 (rule sets 1 and 2). The cost and loading times for these two views are as follows:

	Cost	Loading time
View R1	1184 KB	1896 ms
View R2	821 KB	1439 ms

Table 4

The views created by HyXAC are fine grained and have lesser cost and loading times. The cost and loading time of the views created using rule set 1 are as follows:

RS1	View 1	View 2	View 3	View 4	View 5	View 6	View 7	View 8	View 9	View 10
Cost (KB)	32.6	86.9	58.1	214	219	88.4	138	121	30.7	15
Loading time ms	42	135	86	361	368	144	220	162	39	17

Table 5

The costs of the fine grained views are much smaller than the traditional views and this in turn decreases the loading times as well. Whenever a query from role 1 is to be processed, the view R1 has to be loaded to answer the query. But in HyXAC model we just load the fine grained view that answers the query. The query performance increases because of lesser loading times.

CHAPTER 6

CONCLUSION

In this thesis work, we have attempted to remedy the problems of the various access control models and mechanisms that have been proposed earlier. In particular we discuss the advantages and disadvantages of view based approaches and pre – processing approaches and introduce a new approach called the hybrid approach for XML access control (HyXAC). The key idea of our HyXAC approach is to develop a better access control that is cost effective and dynamic. We achieved this by combining the advantages of the pre – processing approach and view based approach. We used QFilter (a pre – processing method) as the baseline to filter queries and created fine grained views from the document. The important feature of HyXAC is the dynamic materialization of the fine grained views. Resources are allocated on the fly to materialize views. We performed several experiments to demonstrate the effectiveness of HyXAC approach. HyXAC demonstrates various capabilities: (1) it presents dynamic and fine grained view management for optimal query performance (2) it discusses several cost effective solutions for dynamic materialization of views (3) it allows sharing of the fine grained views among different roles eliminating redundancies in storage.

REFERENCES

[1] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou: Structured Materialized Views for XML Queries. VLDB 2007:87-98

[2] A. Balmin, F. Özcan, K. S. Beyer, R. Cochrane, H. Pirahesh: A Framework for Using Materialized XPath Views in XML Query Processing. VLDB 2004:60-71

[3] A. Gabillon, E. Bruno: Regulating Access to XML documents. DBSec 2001:299-314

[4] A. Gabillon: An authorization model for XML databases. SWS 2004:16-28

[5] A. P. Sheth, J. A. Larson, E. Watkins: TAILOR, A Tool for Updating Views. EDBT 1988:190-213

[6] A. Rota, S. Short, M. Ashiqur Rahaman: XML secure views using semantic access control. EDBT/ICDT Workshops 2010

[7] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, D. Srivastava: Answering Queries Using Views. PODS 1995:95-104

[8] B. Choi, G. Cong, W. Fan, S. Viglas: Updating Recursive XML Views of Relations. ICDE 2007:766-775

[9] B. Choi, G. Cong, W. Fan, S. Viglas: Updating Recursive XML Views of Relations. ICDE 2007:766-775

[10] B. Luo, D. Lee, W. Lee, P. Liu: A Flexible Framework for Architecting XML Access

Control Enforcement Mechanisms. Secure Data Management 2004:133-147

[11] B. Luo, D. Lee, W. Lee, P. Liu: Deep Set Operators for XQuery. XIME-P 2005

[12] B. Luo, D. Lee, W. Lee, P. Liu: QFilter: fine-grained run-time XML access control via NFA-based query rewriting. CIKM 2004:543-552

[13] B. Luo, D. Lee, W. Lee, P. Liu: QFilter: rewriting insecure XML queries to secure ones using non-deterministic finite automata. VLDB J. (VLDB) 20(3):397-415 (2011)

[14] B. Mandhani, D. Suciu: Query Caching and View Selection for XML Databases. VLDB 2005:469-480

[15] C. Lim, S. Park, S. Hyuk Son: Access control of XML documents considering update operations. XML Security 2003:49-59

[16] D. Chan An, J. Kim, S. Park: Access Control and Labeling Scheme for Dynamic XML Data. GPC Workshops 2008:329-334

[17] D. Chan An, S. Park: Efficient Access Control for Secure XML Query Processing in Data Streams. CRITIS 2007:161-172

[18] D. Chan An, S. Park: Efficient access control labeling scheme for secure XML query processing. Computer Standards & Interfaces (CSI) 33(5):439-447 (2011)

[19] D. Srivastava, S. Dar, H. V. Jagadish, A. Y. Levy: Answering Queries with Aggregation Using Views. VLDB 1996:318-329

[20] E. Bertino, E. Ferrari: Secure and selective dissemination of XML documents. ACM Trans. Inf. Syst. Secur. (TISSEC) 5(3):290-331 (2002)

[21] E. Bertino, S. Castano, E. Ferrari, M. Mesiti: Specifying and enforcing access control policies for XML document sources. World Wide Web (WWW) 3(3):139-151 (2000)

[22] E. Bertino, S. Castano, E. Ferrari, M. Mesiti: Specifying and enforcing access control policies for XML document sources. World Wide Web (WWW) 3(3):139-151 (2000)

[23] E. Bertino, S. Castano, E. Ferrari: Securing XML Documents with Author-X. IEEE Internet Computing (INTERNET) 5(3):21-31 (2001)

[24] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati: A fine-grained access control system for XML documents. ACM Trans. Inf. Syst. Secur. (TISSEC) 5(2):169-202 (2002)

[25] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati: Securing XML Documents. EDBT 2000:121-135

[26] F. N. Afrati, C. Li, P. Mitra: Answering Queries Using Views with Arithmetic Comparisons. PODS 2002:209-220

[27] F. N. Afrati, R. Chirkova, M. Gergatsoulis, B. Kimelfeld, V. Pavlaki, Y. Sagiv: On rewriting XPath queries using views. EDBT 2009:168-179

[28] G. Sladic, B. Milosavljevic, Z. Konjovic, M. Vidakovic: Access control framework for XML document collections. Comput. Sci. Inf. Syst. (COMSIS) 8(3):591-609 (2011)

[29] G. Sladic, B. Milosavljevic, Z. Konjovic: Extensible Access Control Model for XML Document Collections. SECRYPT 2007:373-380

[30] I. Fundulaki, M. Marx: Specifying access control policies for XML documents with XPath. SACMAT 2004:61-69

[31] I. Fundulaki, S. Maneth: Formalizing XML access control for update operations. SACMAT 2007:169-174

[32] J. A. Blakeley, P. Larson, F. Wm. Tompa: Efficiently Updating Materialized Views. SIGMOD 1986:61-71

[33] J. Patel, M. Atay: An efficient access control model for schema-based relational storage of XML documents. ACM Southeast Regional Conference 2011:97-102

[34] J. Simeon, M. Fernandez: Galax: http://galax.sourceforge.net/

[35] J. Wang, S. L. Osborn: A role-based approach to access control for XML databases. SACMAT 2004:70-77

[36] J. Zheng, A. Lo, T. Özyer, R. Alhajj: Efficient Mechanism for Handling Materialized XML Views. ICEIS 2006:151-162

[37] K. Selçuk Candan, W. Hsiung, Songting Chen, J. Tatemura, D. Agrawal: AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering. VLDB 2006:559-570

[38] L. Bouganim, F. Dang Ngoc, P. Pucheral: Dynamic access-control policies on XML encrypted data. ACM Trans. Inf. Syst. Secur. (TISSEC) 10(4) (2008)

[39] L. Chen, E. A. Rundensteiner, S. Wang: XCache: a semantic caching system for XML queries. SIGMOD 2002:618

[40] L. Huai Yang, M. Lee, W. Hsu: Efficient Mining of XML Query Patterns for Caching. VLDB 2003:69-80

[41] L. Wang, E. A. Rundensteiner, M. Mani, M. Jiang: HUX: a schemacentric approach for updating XML views. CIKM 2006:816-817

[42] M. Duong, Y. Zhang: An Integrated Access Control for Securely Querying and Updating XML Data. ADC 2008:75-83

[43] M. Kudo, N. Qi: Access Control Policy Models for XML. Secure Data Management in Decentralized Systems 2007:97-126

[44] M. Kudo, S. Hada: XML document security based on provisional authorization. ACM Conference on Computer and Communications Security 2000:87-96

[45] M. Murata, A. Tozawa, M. Kudo, S. Hada: XML access control using static analysis. ACM Conference on Computer and Communications Security 2003:73-84

[46] M. Murata, A. Tozawa, M. Kudo, S. Hada: XML access control using static analysis. ACM Trans. Inf. Syst. Secur. (TISSEC) 9(3):292-324 (2006)

]47] N. Bruno, N. Koudas, D. Srivastava: Holistic twig joins: optimal XML pattern matching. SIGMOD 2002:310-321

[48] N. Qi, M. Kudo, J. Myllymaki, H. Pirahesh: A function-based access control model for XML databases. CIKM 2005:115-122

[49] N. R. Adam, V. Atluri, E. Bertino, E. Ferrari: A Content-Based Authorization Model for Digital Libraries. IEEE Trans. Knowl. Data Eng. (TKDE) 14(2):296-315 (2002)

[50] P. Ayyagari, P. Mitra, D. Lee, P. Liu, W. Lee: Incremental adaptation of XPath access control views. ASIACCS 2007:105-116

[51] P. Ayyagari, P. Mitra, D. Lee, P. Liu, W. Lee: Incremental adaptation of XPath access control views. ASIACCS 2007:105-116

[52] R. Busse, M. Carey, D. Florescu, M. Kersten, I. Manolescu, A. Schmidt, F. Waas: XMark an XML Benchmark Project, http://www.xml-benchmark.org/

[53] R. Chirkova, C. Li, J. Li: Answering queries using materialized views with minimum size. VLDB J. (VLDB) 15(3):191-210 (2006)

[54] R. Halder, A. Cortesi: Observation-Based Fine Grained Access Control for XML Documents. CISIM 2011:267-276

[55] R. Pottinger, A. Y. Halevy: MiniCon: A scalable algorithm for answering queries using views. VLDB J. (VLDB) 10(2-3):182-198 (2001)

[56] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman: Role-Based Access Control Models. IEEE Computer (COMPUTER) 29(2):38-47 (1996)

[57] S. Jo, Y. Kim, H. Kouh, W. Yoo: Access Control Model for Secure XML Documents. ACIS-ICIS 2005:352-357

[58] S. Mohan, A. Sengupta, Y. Wu: Access control for XML: a dynamic query rewriting approach. CIKM 2005:251-252

[59] T. Sasaki, T. Fukushima, D. Park, M. Toyama: Fine-grained access control in hybrid relational-XML database. ICDIM 2008:599-604

[60] T. Yu, D. Srivastava, L. V. S. Lakshmanan, H. V. Jagadish: Compressed Accessibility Map: Efficient Access Control for XML. VLDB 2002:478-489

[61] W. Fan, C. Yong Chan, M. N. Garofalakis: Secure XML Querying with Security Views. SIGMOD 2004:587-598

[62] W. Fan, F. Geerts, X. Jia, A. Kementsietsidis: Rewriting Regular XPath Queries on XML Views. ICDE 2007:666-675

[63] W. Xu, Z. M. Özsoyoglu: Rewriting XPath Queries Using Materialized Views. VLDB 2005:121-132

[64] X. Wu, D. Theodoratos, W. H. Wang: Answering XML queries using materialized views revisited. CIKM 2009:475-484

[65] X. Yang, C. Li: Secure XML Publishing without Information Leakage in the Presence of Data Inference. VLDB 2004:96-107