

Student Research Project
University of Applied Sciences Rapperswil

Indexing of Access Control Policies

Stefan Oberholzer, soberhol@herasaf.org

Feb 2010

Supervisor: Prof. Dr. Josef M. Joller

Abstract

Security policies are used to define whether someone is allowed to access a resource with a certain action under certain conditions or not. XACML is an approved OASIS standard that defines how such policies should look like. Until now almost no work is published in the area of optimization of the evaluation process. This thesis is about increasing the performance by using an index over XACML policies. As there can be thousands of policies deployed on a single Policy Decision Point (PDP) it makes sense to have some index over the policies to identifies the applicable policies as fast as possible.

In this thesis we present an index able to return at least the applicable subset of all policies deployed on the PDP. This may bring a major time saving for the evaluation.

To create the index we developed a way how policy targets can be dualized to n-dimensional data. The index also supports updates in an efficient way. That means that the index tree is partially updatable.

The index structure is flexible enough to be optimized for primary and secondary storage structures and fully relies on existing structures of data collections.

Due to the fact that access control in general, and especially XACML, gain more and more importance it is our belief that indexing of access control policies is a feature every XACML implementation must support.

Acknowledgements

I want to thank my supervisor Prof. Dr. Josef M. Joller for the support he gave during the work. I also want to thank Florian Huonder for his great help in reviewing already early drafts.

My further thanks go to the whole HERAS^{AF} team for the ideas I got in the chats I had with them.

In addition I want to thank my family and friends for their understanding that I had only a few times to spend with them during this thesis.

Contents

I. Introduction	5
1. Motivation	5
2. Related work	5
II. Database index structures	6
3. Use cases in terms of policies	6
4. Primary Storage Index structures	7
4.1. Binary Trees	7
4.2. Skip list	7
4.3. T-Tree	7
4.4. kD-Tree	8
4.5. Rangetree	8
5. Secondary Storage Index structures	8
5.1. B-Tree	9
5.2. R-Tree	9
5.3. K-D-B Tree	10
5.4. Index Fabric (index structure for semistructured data)	10
5.5. Grid file	10
5.6. GiST (Generalized Search Tree)	10
III. Policy Indexing	11
6. XACML policies	11
6.1. XACML Target definition	11
6.2. XACML Match functions	12
6.3. Data types	13
6.3.1. Types mapped to integer	13
6.3.2. Types mapped to a Floating point	14
6.3.3. Types mapped to String	14
7. Required Index structure	14
8. Mapping data types to a one dimensional space	16

9. Target Functions in the one Dimensional Space	16
9.1. Type-match functions	16
9.1.1. Regular expression match functions	17
9.1.2. urn:oasis:names:tc:xacml:1.0:function:x500Name-match	17
9.1.3. urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match	17
10. Mapping targets to the n-dimensional space	18
10.1. Step 1: Normalizing the policies	18
10.2. Step 2: Building the hyperrectangle	18
11. Normalizing targets	18
11.1. Covering all RequestElements	19
11.2. Splitting targets by MatchingGroups	19
IV. Resulting Policy Index	21
12. Index base	21
12.1. Requirements to the index structure	21
12.2. Evaluation over primary storage structures	21
12.2.1. Binary Tree	21
12.2.2. Skiplist	22
12.2.3. T-Tree	22
12.2.4. kD-Tree	22
12.2.5. Rangetree	23
12.3. Evaluation over secondary storage structures	23
12.3.1. B-Tree	23
12.3.2. R-Tree	23
12.3.3. K-D-B-Tree	24
12.3.4. Grid file	24
12.4. Conclusion and selection of structure as basis for policy index	24
13. The policy-index-tree	25
13.1. The big picture	25
13.2. Algorithms	25
13.2.1. Find policies	26
13.2.2. Insert Policy	27
13.2.3. Remove Policy	29
13.3. Limiting behavior	31
13.3.1. storage	31
13.3.2. Read	32
13.3.3. Insert	32
13.3.4. Remove	32

13.4. Implementation hints	32
13.4.1. Tree independant	32
13.4.2. String Dimensions	33
13.4.3. Regex Optimization	33
13.4.4. Adjected Bounds	33
13.4.5. Any match	33
13.4.6. Multivalued types	33
13.4.7. Specialized urn:oasis:names:tc:xacml:1.0:function:x500Name-match	33
13.4.8. Choosing the dimension	34
14. Conclusion	34
V. APPENDIX	35
A. Glossary	36
B. Example Code	36
B.1. Index.java	36
B.2. Node.java	37
B.3. ListEntry.java	40
C. Thesis Appendix	43
C.1. Thesis Overview	43
C.1.1. Target	43
C.1.2. People	43
C.1.3. Dates	43
C.2. Project Plan	43
C.3. Risk Management	44
C.4. Time Evaluation	44
C.5. Personal Statement	44

Part I.

Introduction

In this section we will give a short introduction about this thesis. In the first part of this introduction we explain the motivation of introducing an index over access control policies. In the following section we will give a short overview about the related work we could identify.

1. Motivation

The eXtensible Access Control Markup Language[?] (XACML) is a standard specifying how policies can be written in a uniform way and how these policies must be evaluated. This evaluation is done in a policy decision point (PDP). As a single PDP may contain a lot of policies and the evaluation time is a critical factor it makes sense to index the policies to find the relevant policies as fast as possible.

When the HERAS^{AF} XACML implementation was implemented a simple index was created. This index had the drawback that it could not be updated within a sensible space of time. In this thesis a improved index structure will be specified.

2. Related work

There are only a few papers about indexing or optimizing policies.

The XEngine [?] paper describes an engine that internally changes and restructures the information in the policies to get a better evaluation performance. All values are replaced by numbers representing the values. Policies are also restructured and then the evaluation is done against the restructured policies.

As we have not been able to find any paper on indexing policies, this work seems to be one of the first about policy indexes.

Part II.

Database index structures

In the first part multiple data structures will be analyzed. The knowledge of these structures will hopefully help to find an appropriate indexing structure for XACML policies. Regarding indexing there was a lot of research in the area of databases.

In a first part we describe what the focus of the analysis is. In the second part, binary storage index structures are listed. Afterwards we will discuss secondary storage index structures.

3. Use cases in terms of policies

In this chapter we will discuss the requirements for indexing structures that might be used to hold policies.

A Policy Decision Point (PDP) [?] will be responsible for policies of multiple applications or resources, each having multiple policies. Companies allowing their customers to configure privacy policies on their own can even have millions of policies.

The most accesses to the policy repository will be read-access because one single policy is not going to change often. As an example we can take a policy, describing privacy policies of customers. A customer will describe once who may access his data and who not. Until something changes in his environment, the customer is not going to change his own privacy policies. As there will be a lot of policies deployed in one PDP, changes in the policy repository as a whole will happen often, as a result of the big amount of policies deployed. Because of this and the fact that creating an index for a big number of policies will logically take a long time it is not feasible to simply reinitialize the whole index structure upon a change. Therefore insert, update and delete operations must be supported as well but the impact to the read-access should be as small as possible.

To reach this goal, based on the facts from above, optimistic locking is obviously the best choice. Optimistic locking creates new versions of the changed parts in the index. As long as the changes are not in effect, read-access will be against the old version. After the change is done, read-access still in progress uses the old version while newly started read-accesses obtain the newest version. During the change the changing parts are locked for other operations changing data in this part of the index.

A single update on a policy set will mostly consist of multiple changes that must be deployed atomically to be sure that each request will also be evaluated against a valid and consistent policy set. Because optimistic locking is used, an update can be treated like a remove followed by an insert of a new value.

How the policy set on a PDP can be changed atomically is not part of this research project.

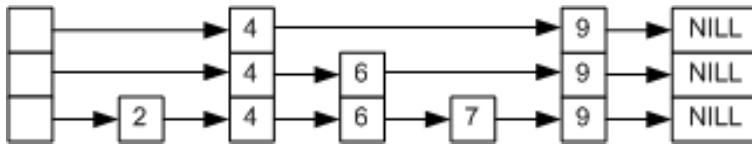


Figure 2: Skip list

4. Primary Storage Index structures

In primary storage index structures the index as well as the indexed data is stored in the primary storage.

The only important thing for primary index structures is the time required for the different kinds of access.

4.1. Binary Trees

In a binary-tree each node has two sub-nodes. One Sub-nodes leads to a sub-tree containing all values smaller than the actual node and the other node contains a sub-tree with all bigger values. The values are stored in all nodes so each node contains a key-value pair. There are multiple types of binary trees, well known are the binary search tree and AVL-Tree. Balanced binary search trees have a worst case time for search insert and delete of $O(\log n)$ [?].

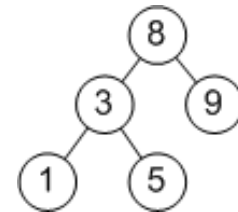


Figure 1: Binary-tree

4.2. Skip list

Skip lists are a probabilistic alternative to balanced trees [?]. Even though they have a bad worst-case performance. The chance that the tree performance is bad is very low. It is easy to implement and can be configured to be very space efficient.

In general the skip list is like a double linked list with the difference that each entry can have a height. On every height level only the policies with the same or a higher level are linked together. The search begins with the highest level. Every time the next entry would be bigger then the searched value the search continues on the next lower level in the same entry. An example of a skiplist can be seen in Figure 2.

4.3. T-Tree

A T-Tree is an index structure for in-memory databases. Its target is to have both, low storage requirements and low access time. So in contrary of a binary-tree not only the access time is important but also the whole structure should require as less memory as possible. The main structure is like in the binary-tree but a node may contain multiple values. So the left sub-tree contains only values smaller than the smallest value in the node, while the right subtree contains only values bigger than the biggest value in the node [?].

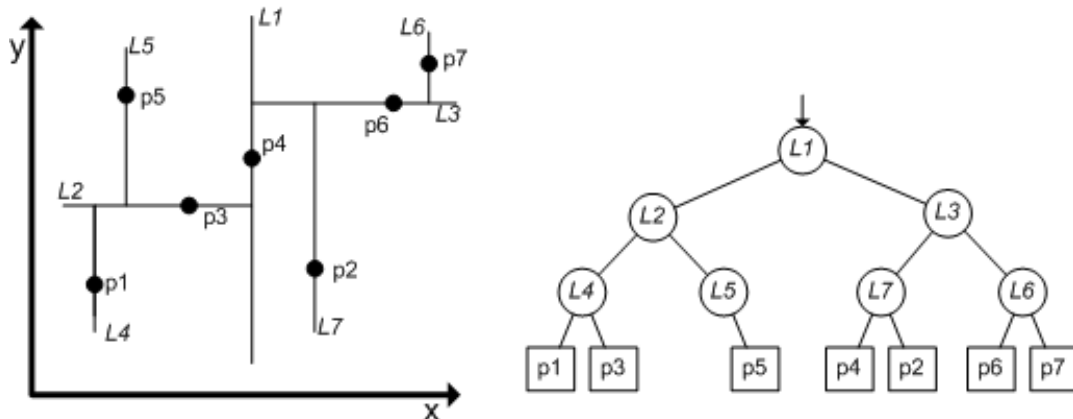


Figure 3: kD-Tree example

4.4. kD-Tree

kD-trees are designed for multidimensional range searches over values. In the one dimensional case a kD-tree is just a binary search tree where the find algorithm searches the smallest value and reports all values until the node values are no longer in the search range. In the multidimensional case each level of hierarchy splits the values on the median. E.g. the root node splits the points on the x-axis into 2 sub-trees. The root node of this sub-tree splits its sub-nodes on the y-axis into two new sub-trees and so on. An example of a kD-tree can be seen in Figure 3. To search a range the tree must be traversed down with a left and a right bound query. If a sub-tree lies fully in the search range the data in the tree is within the search range and therefore returned [?].

4.5. Rangetree

A rangetree is optimized for range queries on spatial data¹. The dimensions are splitted up in multiple trees. The root tree contains all values of the first dimension. The leaves contain another trees for the next dimensions. Each node contains all values contained under it and also a tree into the next dimension to search values. To search in the subtree a normal range search with left and right bound query is done. For all nodes and leaves that are fully contained in the search the contained subtree is also searched.

5. Secondary Storage Index structures

When an index is getting to big to be stored in the primary storage it has to be stored in the secondary storage. In this case minimizing the number of storage accesses is

¹Spatial data are geo information

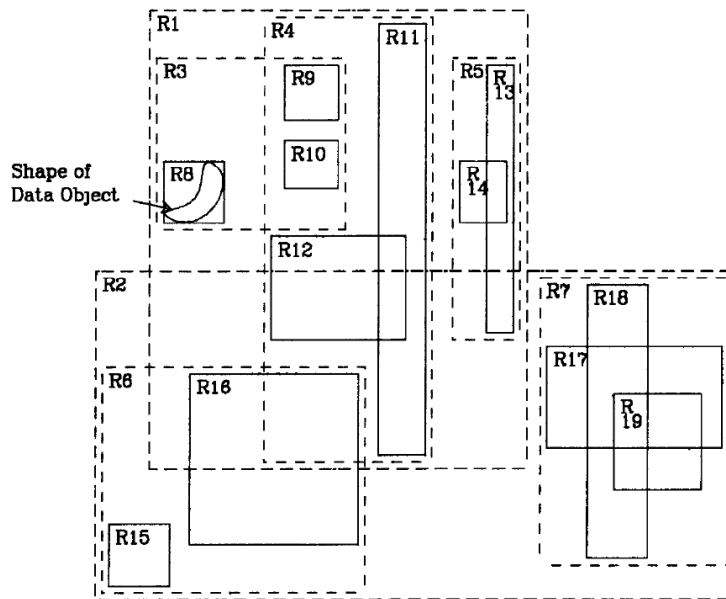


Figure 4: R-Tree data example

the most critical operation regarding the performance. Because of this secondary storage index structures must minimize the storage access too.[?]

5.1. B-Tree

The B-Tree is the most used indexing structures for database systems. In a B-tree any node has multiple children. The amount of children is configured in a way that the size of the node is the size that can be read in one operation from secondary storage. The rest is like in the binary-tree. Nodes are optimized to be stored on the secondary storage [?].

5.2. R-Tree

The R-Tree is a sub-form of the B-Tree that is optimized for spatial data. The nodes in an R-Tree represent a region while its subnodes are subregions of the parent node. Nodes at the same level can intersect each other so it may be necessary to read several leaves to get all areas containing a single searched point [?].

Figure 4 shows an example of data indexed by an R-Tree. Boxes with dashed lines denote areas indexed by a node. Areas with normal lines are minimal boxes around indexed data.

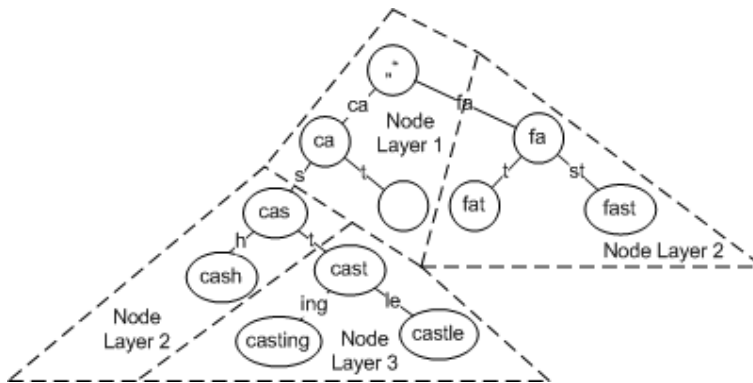


Figure 5: Index fabric

5.3. K-D-B Tree

The K-D-B Tree is a mix between a B-Tree and a kD-Tree. It is optimized for multi-attribute indexes. The data is stored in the leaves. Each Node represents an area. Child-nodes are sub-areas of the area of the parent node. The nodes themselves are organised as a kD tree.[?].

5.4. Index Fabric (index structure for semistructured data)

Index Fabric is an index structure for semi-structured data. It also allows named queries to handle often used queries more efficient. The paths to the values are encoded as strings. Each node contains a patricia trie². See Figure 5 for an example.

5.5. Grid file

The grid file structure is created to minimize the access to the secondary storage. It stores the data on the secondary storage in an n-dimensional grid. The index over the areas in the grid is held in another file. So it always requires only two file access operations. How this data is handled once it is in the memory is only rarely specified and multiple possible solutions are mentioned [?].

5.6. GiST (Generalized Search Tree)

GiST is a generalized search tree that allows to be adapted easily to different tree structures optimized for secondary storage [?].

²tree with string fragments as node name, having the next tree or data as leafs.

Part III.

Policy Indexing

Table 1 shows the notation used in this part.

Symbol	Description
PT^i	Policy target. i is the identifier of the policy.
$VirtualT_n^i$	Virtual policy target. i is the identifier of the policy it belongs to. n is the identifier of the VirtualTarget
$TG^{j,i}$	TargetGroup. i is the identifier of the policy. j denotes the type of the Target group. The possible values are: subject, action, resource, environment
$MG_k^{j,i}$	MatchingGroup. i and j are specified as by the TargetGroup. k denotes the index in the MatchingGroup element.
$ME_{(k,m)}^{j,i}$	MatchingElement. i , j and k are like in the MatchingGroup it belongs to. m denotes the index of the MatchElement in the MatchingGroup.

Table 1: Notation

6. XACML policies

To understand how an index for XACML policies should look like, we first look at the XACML structure to see the boundaries we are in. In this section we will take a look at the definition of policies in the XACML specification [?].

The XACML specification specifies policy sets, policies and rules. A PDP may contain multiple policies or policy sets for the evaluation. Further a policy set may also contain multiple policy sets or policies. A policy consists of rules. It is easy to see that the policies form a tree with a special policy set, the PDP, as root. From here on we will use the term policy for policy sets and policies.

As this tree may consist of thousands of policies at the same level, an index can speed up the evaluation time significantly.

Each policy or rule contains a target element (PT^i , where i is the identifier of the policy) specifying constraints that must be fulfilled to determine if that specific policy or rule is applicable to a certain request. Due to the fact that the target is the only element that says if a request is applicable or not, it is the part that shall be indexed.

6.1. XACML Target definition

A target can contain the following elements:

- Subjects ($TG^{subject,i}$)
- Resources ($TG^{resource,i}$)
- Actions ($TG^{action,i}$)
- Environments ($TG^{environment,i}$)

As each of these elements has the same structure we will just identify them by the names subject, resource, action, environment and only describe the **Subjects** element as a representant of all other elements.

If the **Subjects** element is empty (i.e. `<Subjects />`), the policy has no restriction about this, meaning it is applicale for all subject attributes. For each element that occurs in the target it has to match if the policy shall match the request ($TG^{subject,i} \wedge TG^{resource,i} \wedge TG^{action,i} \wedge TG^{environment,i}$). In the following parts these elements will also be named **TargetGroupType**.

The **Subjects** element consists of multiple **Subject** elements (we will call them **MatchingGroup** elements ($MG_k^{j,i}$)). If one of these elements is applicable, the whole **Subjects** element is applicable for the request. In the following these elements will be named **MatchingGroup**.

The **Subject** element itself contains multiple **SubjectMatch** elements (we will call them **MatchElement** ($ME_{k,m}^{j,i}$)). The **SubjectMatch** element consists of muliple **AttributeValue** elements, a selector for the values in the request and the id of the function that is used to determine if the values match. If all **SubjectMatch** elements in the **Subject** element are applicable to the request, the whole **Subject** element matches the request. In the following these elements will be named **MatchElement**.

To describe it in boolean algebra we can define a **TargetGroup** as ($TG^{j,i} = MG_0^{j,i} \vee MG_1^{j,i} \vee \dots \vee MG_n^{j,i}$)

6.2. XACML Match functions

The matching functions are predicate functions. This means that all of them have a boolean return value. They take the value to match to as first argument and a bag, conaining all elements of the corresponding request element, as second argument. So the following function categories, defined by XACML, are allowed:

- urn:oasis:names:tc:xacml:2.0:function:*-type-equal
- urn:oasis:names:tc:xacml:2.0:function:*-type-greater-than
- urn:oasis:names:tc:xacml:2.0:function:*-type-greater-than-or-equal
- urn:oasis:names:tc:xacml:2.0:function:*-type-less-than
- urn:oasis:names:tc:xacml:2.0:function:*-type-less-than-or-equal
- urn:oasis:names:tc:xacml:2.0:function:*-type-match

The match functions are the only ones that behave a little bit special. In the category of `*-type-match` functions are mainly functions matching the values by using regular expression. Exceptions are the `x500Name-match` and `rfc822-match` functions. The exact definition of these functions and the data types can be found in the XACML specification [?].

6.3. Data types

In this section we describe how the different data types will be treated. XACML defines 16 data types. Dualizing this data types to a string, integer or floating point number allows the handling if them easier and in a more uniform way.

- String
- Integers
- Floating point numbers

Theoretically, everything could be reduced to an `integer`, even `floating point numbers`. The possible granularity is given by the amount of usable bytes in the implementation. This would lead to too big numbers, we will not do this, and take `strings` and `floating points` as the only data types allowed beside the `integer` type. For example if we map all possible instances of a string with a maximal length of 256 bytes, we would have $< \textit{NumberOfAllowedChars} >^{256}$ possibilities.

6.3.1. Types mapped to integer

In this subsection we will list all types we will dualize into an `integer` and also show how this can be done.

`http://www.w3.org/2001/XMLSchema#integer` This is already an integer and therefore no mapping is needed.

`http://www.w3.org/2001/XMLSchema#boolean` The mapping of a `boolean` to an `integer` is `false` as 0 and `true` as 1.

`http://www.w3.org/2001/XMLSchema#time` As basis for time dualization, it is first transformed into GMT. The `integer` representation of the time is then the integer value of the milliseconds since 00:00 GMT.

`http://www.w3.org/2001/XMLSchema#date` It is transformed into the milliseconds since January 1, 1970, 00:00:00 GMT. To have the date only, the day component is always 00:00:000 o'clock.

`http://www.w3.org/2001/XMLSchema#dateTime` In the case of date, it is transformed into the milliseconds since January 1, 1970, 00:00:00 GMT.

urn:oasis:names:tc:xacml:2.0:data-type:dayTimeDuration It is transformed into the milliseconds of the `dayTimeDuration` as integer dualization.

urn:oasis:names:tc:xacml:2.0:data-type:yearMonthDuration It is transformed into the milliseconds of the `yearMonthDuration` as integer dualization.

6.3.2. Types mapped to a Floating point

http://www.w3.org/2001/XMLSchema#double This is the only type that can be represented as floating point type because it is hardly possible to find another well fitting representation.

6.3.3. Types mapped to String

For all types in this section, we will just take the `String` representation they have in their XML form. We will treat the following types as `String`:

- `http://www.w3.org/2001/XMLSchema#string`
- `http://www.w3.org/2001/XMLSchema#anyURI`
- `http://www.w3.org/2001/XMLSchema#hexBinary`
- `http://www.w3.org/2001/XMLSchema#base64Binary`
- `urn:oasis:names:tc:xacml:1.0:data-type:x500Name`
- `urn:oasis:names:tc:xacml:1.0:data-type:rfc822Name`
- `urn:oasis:names:tc:xacml:2.0:data-type:ipAddress`
- `urn:oasis:names:tc:xacml:2.0:data-type:dnsName`

7. Required Index structure

As described in the previous section, targets define to which requests a certain policy or rule applies to. Every value will be treated as point, while each `MatchElement` has a different axis. So `n` `MatchElements`, will build a `n`-dimensional hyperrectangle. We can treat a request like multiple points in the whole space of possible values. Figure 6 shows 2 informal described requests and their values in the two dimensional space.

Like request targets can be defined as an hyperrectangle or sets of hyperrectangles (if the target does not build only one hyperrectangle) and points in the same `n`-dimensional space. Figure 7 shows 3 policies in the two dimensional space. On the subject-id axis we have the possible attributes subject-id attributes. The bill-cost axis just represents an integer space.

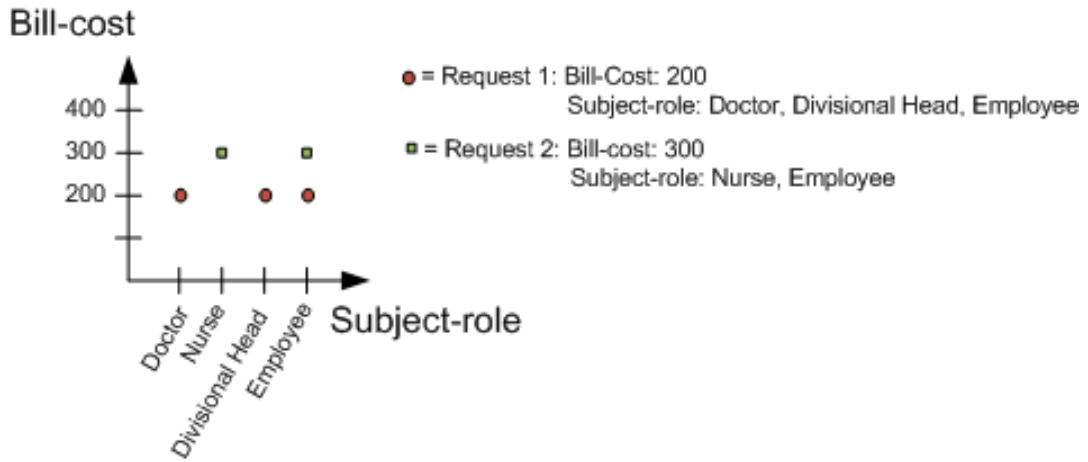


Figure 6: Informal Request example

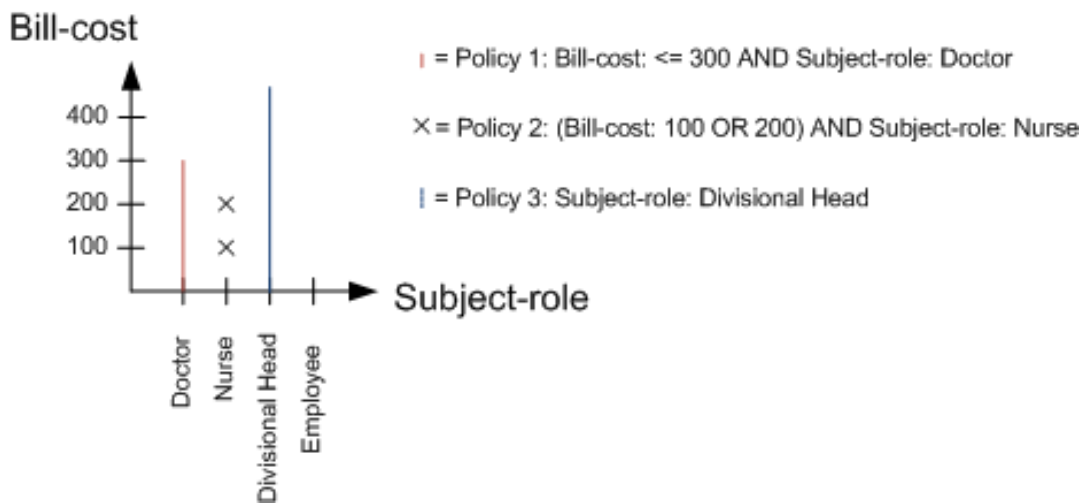


Figure 7: Informal Policy example

If any of the points of the request lies upon an hyperrectangle that is described by a policy then that policy matches the request. So if we take the example in Figure 7 and Figure 6, policy 1 and 3 would match to request 1, while no policy would match to request 2.

8. Mapping data types to a one dimensional space

In section 6.3 we mapped all data types specified by XACML to three base data types. In this section, we will describe how these types can be mapped to a vector.

For `integer` and `floating point` values this is quite easy, as they have a fixed order and can directly be mapped to the points in the dimension. As both, `integer` and `floating points` are finite because of the limitations of the used programming language and system, all possible values can be mapped. Due to the fact that mapping all possible values would take too much storage space it is up to the index structure to minimize the number of points that must be stored.

`String` values can be mapped to a vector by restricting them to a limited size and than map all possible strings like in a dictionary (e.g. `aa < ab < ba`). As in the case of `integers` mapping all possible values would take too much storage space. So it is up to the index structure to minimize the number of points that must be stored.

9. Target Functions in the one Dimensional Space

In section 6.2 we listed the allowed types of match functions. In this section we will describe how they can be mapped to points in the one dimensional space.

The matching values of the `equals` functions are just points in the one dimensional space.

The different forms of `greater-than` and `less-then` functions can be represented as lines. Starting by the defined point and than going to the maximum (given by the programming language or system) of the described direction. Figure 7 shows an example of such functions. On the subject-role axis `equals` functions are used. On the bill-cost axis for doctor subject-roles, a `less-than-or-equals` function is used.

So only the matching functions contain some complexity. This will be described in the following subsections.

9.1. Type-match functions

First we will describe how the regular expression match functions can be treated. Afterwards we will show how the `urn:oasis:names:tc:xacml:1.0:function:x500Name-match` and `urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match` can be substituted with a regular expression function.

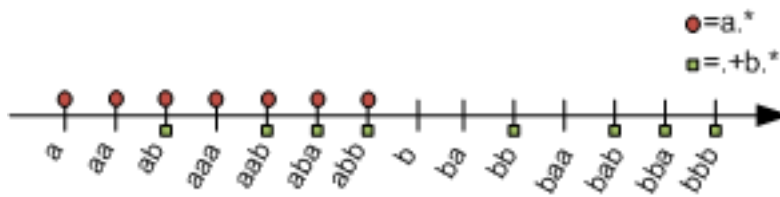


Figure 8: Regular Expression Match example

9.1.1. Regular expression match functions

As the the maximal allowed size for strings is limited (it may be configurable) like described in section 6.3, regular expressions can be treated in the way that the target is defined by all substrings matching the regular expression.

Figure 8 shows an example of this. To have a small example we take an alphabet only containing the character a and b and a maximum string size of 3.

9.1.2. urn:oasis:names:tc:xacml:1.0:function:x500Name-match

The `urn:oasis:names:tc:xacml:1.0:function:x500Name-match` function can be transformed into a `regex-match` function. To do this we will order all values in the attribute value as described in the `urn:oasis:names:tc:xacml:1.0:function:x500Name-equal` function.

Now we can split the X500Name into its attributes and put them together to a regular expression allowing any character between the attributes. So the regular expression would look like this: $regex = \backslash.*attr_1\backslash.*\dots\backslash.*attr_n$ while $attr_i$ are the single attributes.

9.1.3. urn:oasis:names:tc:xacml:1.0:function:rfc822Name-match

This function can be transformed into a regular expression too. The XACML specification specifies this function in the way that the attribute to match can be define as the local-part (e.g. `foo` for the `rfc822Name` `foo@bar.ch`), the domain part (e.g. `bar` in the `rfc822Name` `foo@bar.ch`) or the whole `rfc822Name`. The local-part is case-sensitive, so if this name is part of the attribute value, it can just be taken as it is into the regular expression followed by the `@` symbol. The domain part is case-insensitive. So for this we will have to define every character with the upper case and lower case character. For example `@foo.bar` is transformed into $\backslash S@[fF][oO][oO]\backslash.[bB][aA][rR]$. If the domain and the name are listed, they are just appended to each other. For example `name@foo.bar` will be transformed to $name@[fF][oO][oO]\backslash.[bB][aA][rR]$.

10. Mapping targets to the n-dimensional space

In the previous chapter we described how functions and data types can be mapped to the one dimensional space. In this chapter we will show how the whole target can be mapped to the n-dimensional space.

This can be done in two steps. Normalizing the policies followed by building the hyperrectangle in the n-dimensional space.

10.1. Step 1: Normalizing the policies

Normalizing the target can be done in two steps.

- All policies must contain MatchGroups in every TargetGroupType targeting the same request elements. To reach this we will normalize the MatchingElements as described in section 11.1.
- Every TargetTgroupType must only contain a single MatchingGroup. This can be done by normalizing the targets and building multiple virtual policies. This is described in section 11.2.

10.2. Step 2: Building the hyperrectangle

Because of the normalization in the first steps, all targets now consist of AND connected MatchElements over the same elements in the request. Now we can treat the area of MatchElement as the hyperrectangle only restricted by the dimension the MatchElement applies to. The matching area of all MatchElement is described by the intersection of all areas. An example of this with two dimensions can be seen in Figure 7.

By doing this the result is a set of hyperrectangles describing the space of all request element combinations a single policy applies to. So this can be linked with the policy the hyperrectangles belong to. By adding all hyperrectangles of the other policies and linking them to the corresponding policies, we can add all policies to the same n-dimensional space.

If a request has to be evaluated, we can just build all points in the n-dimensional space by taking all combinations of the elements, that occur in the n-dimensional space, and check if any one of these lies in one of the hyperrectangles. By taking the policies linked by the matching hyper-rectangles we can get the matching policies.

11. Normalizing targets

In the previous chapter we described some restrictions necessary to build an n-dimensional space for all possible values in the request elements that are part of any target definition. In this section, we describe how targets can be normalized to a set of hyperrectangles.

11.1. Covering all RequestElements

As the targets are represented by hyperrectangles, the target has to have an interval of accepted values for every dimension in the space. To do this, we define a `MatchElement` for each dimension, the target does not describe any required values.

Once we have this we have to append the required `MatchElements`. The created `MatchElement` has a designator pointing to the request element type it is created for. Any attribute value and the `MatchAllFunction` we define in the following.

This `MatchAllFunction` should match to all values of the dimension it is created for. This means that it does not reduce the number of requests the policy will match to.

As XACML defines the match functions as predicate functions, taking an `AttributeValue` as first argument and a bag containing the values of the request as second argument the `MatchAllFunction` takes this arguments too. As both arguments are not required they will be ignored and the function just returns true.

11.2. Splitting targets by MatchingGroups

As mentioned in section 10 building the combined n-dimensional space of a single policy target is very hard if there are multiple `MatchingGroup` in the same `TargetGroupType`.

To solve this problem we build multiple virtual targets for each policy containing more then one `MatchingGroup` per `TargetGroupType` element. The union of all virtual targets $VirtualT_n^i$ together build all possible combinations of the `MatchingGroup` elements. Because of this they can fully substitute the original target. We can see this in Figure 9.

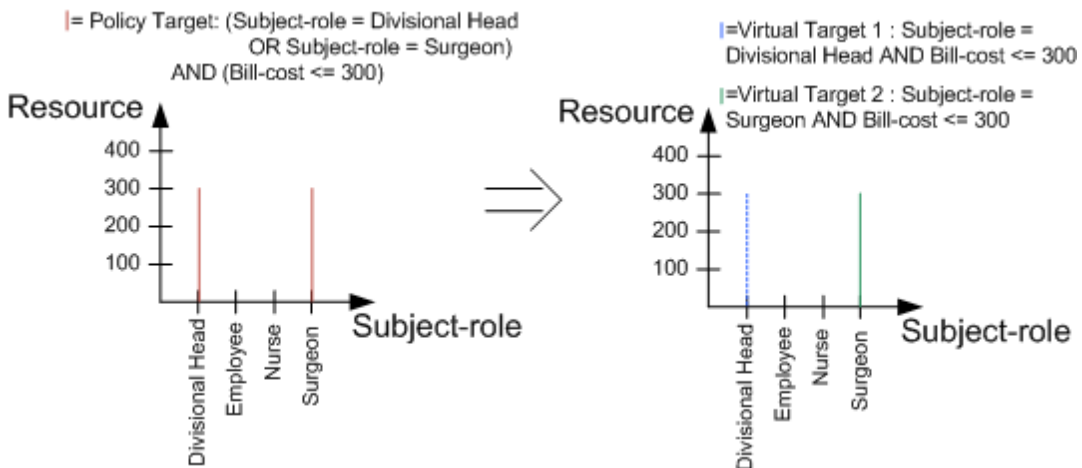


Figure 9: Target split example

The following formula shows a short example of how policy targets can be splitted up into multiple policy targets. In the example we have 2 subject matching groups

and 3 action matching groups.

$$TG^{subject,1} = MG_0^{subject,1} \vee MG_1^{subject,1} \quad (1)$$

$$TG^{action,1} = MG_0^{action,1} \vee MG_1^{action,1} \vee MG_2^{action,1} \quad (2)$$

$$TG^{resource,1} = MG_0^{resource,1} \quad (3)$$

$$TG^{environment,1} = MG_0^{environment,1} \quad (4)$$

$$PT^1 = TG^{subject,1} \wedge TG^{action,1} \wedge TG^{resource,1} \wedge TG^{environment,1} \quad (5)$$

$$VirtualT_0^1 = MG_0^{subject,1} \wedge MG_0^{action,1} \wedge MG_0^{resource,1} \wedge MG_0^{environment,1} \quad (6)$$

$$VirtualT_1^1 = MG_0^{subject,1} \wedge MG_1^{action,1} \wedge MG_0^{resource,1} \wedge MG_0^{environment,1} \quad (7)$$

$$VirtualT_2^1 = MG_0^{subject,1} \wedge MG_2^{action,1} \wedge MG_0^{resource,1} \wedge MG_0^{environment,1} \quad (8)$$

$$VirtualT_3^1 = MG_1^{subject,1} \wedge MG_0^{action,1} \wedge MG_0^{resource,1} \wedge MG_0^{environment,1} \quad (9)$$

$$VirtualT_4^1 = MG_1^{subject,1} \wedge MG_1^{action,1} \wedge MG_0^{resource,1} \wedge MG_0^{environment,1} \quad (10)$$

$$VirtualT_5^1 = MG_1^{subject,1} \wedge MG_2^{action,1} \wedge MG_0^{resource,1} \wedge MG_0^{environment,1} \quad (11)$$

$$PT^1 \equiv VirtualT_0^1 \vee VirtualT_1^1 \vee VirtualT_2^1 \vee VirtualT_3^1 \vee VirtualT_4^1 \vee VirtualT_5^1 \quad (12)$$

Line 1-4 define the concrete content of each of these `TargetGroupTypes`. Line 5 defines a policy with these `TargetGroupTypes`. In the lines 5-11 we define all possible combinations of these `MatchingGroups`. Finally in line 12, we define the policy as union of all of this virtual targets.

Part IV.

Resulting Policy Index

12. Index base

In part II we described how the index will be used and also introduced often used index structures. In this section we will first make a short list of the requirements mentioned in section 3. Afterwards we will add a short overview of the advantages and disadvantages of the introduced index structures if they are used for policies. In the end we will make a conclusion and decide which one can be taken as basis structure for the policy index.

12.1. Requirements to the index structure

- n-dimensional (spatial) data must be indexed.
- The most important part is the read performance.
- Update and delete must be possible.
- It should not take too much storage space. This to hold the index as long as possible in the primary storage.

12.2. Evaluation over primary storage structures

12.2.1. Binary Tree

Advantages

- Fast in searching a value.
- Updateable.
- Acceptable storage requirements.
- n-dimensional data can be stored by chaining multiple trees. So every leaf node contains the tree for the next dimension.
- The intervals could be stored as placeholder values, indexing the difference between two node and can be appended to the leaves.

Disadvantages

- Not able to store intervals. They must be appended.

12.2.2. Skiplist

Advantages

- Low storage requirements.
- Updateable.
- Easy to implement.
- By inserting a placeholder for intervals between all entries, ranges can be indexed easily.
- n-dimensional data can be stored by chaining multiple Skiplists. So every entry contains the skiplist for the next dimension.

Disadvantages

- No good, guaranteed, worst-case performance for read access.

12.2.3. T-Tree

Advantages

- Good read access performance (but worse than in the binary tree).
- Low storage requirements.
- n-dimensional data can be stored by chaining multiple trees. So every tree leaf contains the tree for the next dimension.
- The intervals could be stored as placeholder values, indexing the difference between two nodes can be appended to the leaves.

Disadvantages

- Not optimized for spatial data.

12.2.4. kD-Tree

Advantages

- Optimized for multi dimensional search.
- Fast in searching a value.
- Acceptable storage requirements.

Disadvantages

- Can only index points. As the points are not strictly ordered, it is hard to generate intersections and create interval points.

12.2.5. Rangetree

Advantages

- Can store n-dimensional data.

Disadvantages

- Optimized for range queries over points.
- Read access for points is slow if only the edges of the areas are stored because all leaves have to be passed to identify all matching areas.
- Not optimized for updates.

12.3. Evaluation over secondary storage structures

12.3.1. B-Tree

Advantages

- Fast read access.
- Updateable.
- n-dimensional data can be stored by chaining multiple trees. So every leaf node contains the tree for the next dimension.
- The intervals could be stored as placeholder values, indexing the difference between two nodes can be appended to the leaves.

Disadvantages

- Not optimized for spatial data.

12.3.2. R-Tree

Advantages

- Good read performance on spatial data.
- Indexes areas.
- Updateable.

Disadvantages

- Finding all areas matching to a point requires a search over multiple leaves.

12.3.3. K-D-B-Tree

Advantages

- Good read performance.
- Optimized for spatial data.

Disadvantages

- Can only index points. Indexing all intersection points, with the areas the tree is splittet into, is hard.

12.3.4. Grid file

Advantages

- Optimized for read access.

Disadvantages

- In node structure very unclear.
- Rectangles in one dimension have to be of the same size. So a policy has to be splitted up in more parts than it would be necessary.
- Requires less read access than B-Tree.

12.4. Conclusion and selection of structure as basis for policy index

None of this structures is optimized for read access of a single point in spatial range data. As we have not been able to find a solution to adapt the range-tree, kD-tree, R-Tree and K-D-B tree to be able to index ranges by only requiring a single read access in an acceptable time, we will not take them as basis for the index.

The T-Tree reduces the read performance of a binary tree to have less storage requirements, so we will not take this tree as basis as well.

The skiplist is very easy to expand to search for a point in ranges. As it has no guaranteed worst-case performance, we will only take the idea of the linked-list below and index this list with a balanced binary tree.

As the binary tree to index the linked-list can just be replaced by a B-tree we will take the B-tree as structure for secondary storage.

13. The policy-index-tree

In this section we will introduce the policy-index-tree. First we will give a graphical overview and describe the big picture. Afterwards we will list the algorithms for read, insert, delete and update. In the end of this chapter we will introduce some improvements to optimize string indexes, regular expression match functions etc.

13.1. The big picture

Basically the index consists of double-linked-lists and an indexes over the values in these lists. In the first dimension there is a linked list containing all values and all open intervals between these values. There is also an interval from minus infinity to the smallest value and one from the biggest value to infinity. For each entry in the linked list another linked list for the next dimension is appended. In case of the last dimension the linked list contains references to the policies indexed by the entry.

The easiest way to show the behavior is in the one dimensional space. This is shown in figure 10. Here three policies are indexed. The lines show the range of values their targets accept. Every value points to its representation in the double-linked-list while all values in the double linked list point to policies.

Let's take two examples, one with a value existing in the binary tree and one that does not exist. If we search with the value 5 the binary tree is searched. The node with 5 is retrieved. By taking the link to the double-linked-list we will find the policy P2.

In the case we search for 5.5 we will also find 5 as a leaf node. As the value is greater than 5 we will take the value linked in the double-linked-list and then take the value right from the retrieved entry. Due to the fact that the next bigger end of a range is indexed by an own value there are no further checks required if the value lies in the range of the interval. The policy P2 can be returned without any more checks.

In the case of an index over more dimensions the same structure for the next dimension would be referenced by the double-linked list and the search will have to continue in the next dimension until all dimensions have been searched or an empty entry in the double-linked-list is returned.

As the binary tree is just used to retrieve the correct entry in the double-linked list it can be replaced by a B-tree if the index will get too big and must be stored on the secondary storage. The linked list itself can be fragmented into multiple files if needed.

13.2. Algorithms

In this subsection the used algorithm for read, insert and delete are explained. Update will be done by a delete operation followed by an insert.

We will show all algorithms for the case of a binary tree as index structure. We take the binary tree because it is a simple implementation allowing us to focus on

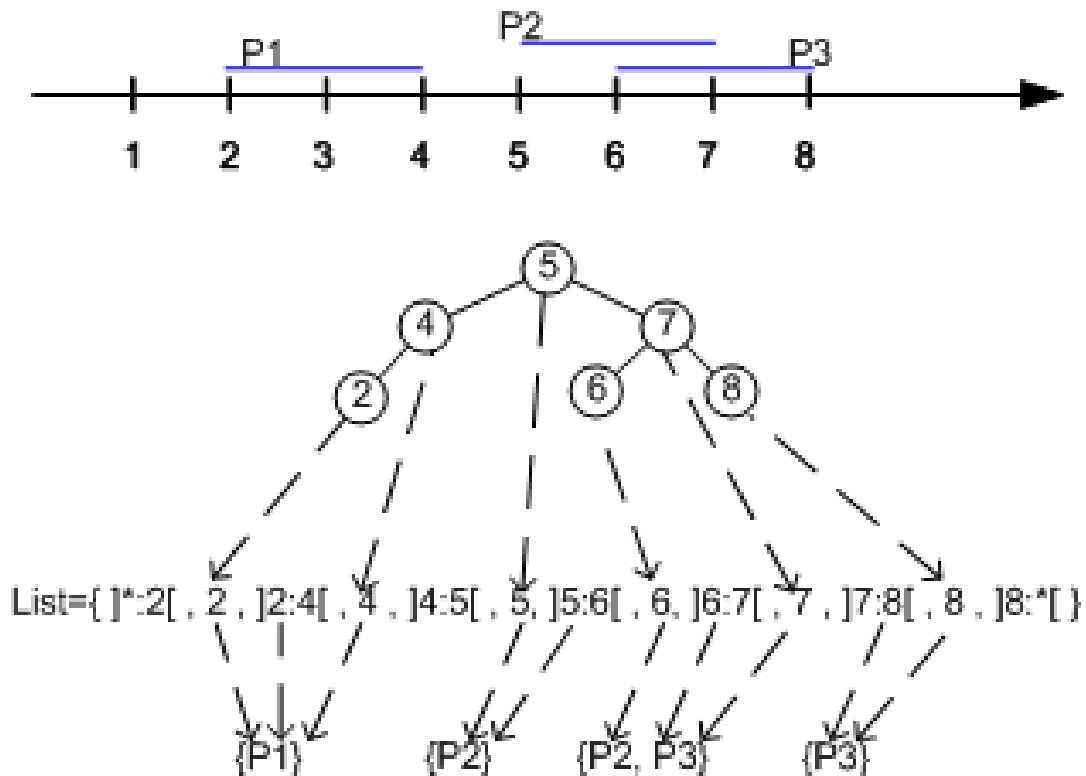


Figure 10: One dimensional example

the important parts. In a real implementation an AVL-tree or B-Tree will be better.

13.2.1. Find policies

Given the tree in picture 10 it can easily be shown how matching policies can be found. We will show this by the example of a search for the policies matching the value 3.

First, the tree is searched until the searched value is found or a leaf is reached. In the case of a search for the value 3 the node 2 is returned. As the node two is returned it is known that the interval right of the list entry linked by the node has to be taken. As the next bigger node would be returned (in this case the node 4) if the value is bigger than the interval. So this needs no check and the list-entry on the right side of the one linked in the node 2 can be returned. If the content of the linked list entry is a policy the last dimension is reached. If this is link to a root node of another tree then the search has to be continued in the next dimension until the last dimension is reached.

If the request returns more than one value, this has to be done for every value.

findPolicies

1. Input: Request = r, Searched Request element id = elemId
2. foreach (v = element value of matching element in r)
3. Search leave l where search matches or no more leaves exist
4. Compare element value with searched value
 - a) If l.value = value -> entry = l.listEntry.
 - b) If l.value < value -> entry = l.listEntry.right
 - c) If l.value > value -> entry = l.listEntry.left
5. react to entry.content
 - a) if entry.content = Set of Policies -> add content to return-values
 - b) if entry.content = next-index-dimension -> content.search(r,Request-element-id-of-next-index-dimension)
6. foreach end
7. return return-values

We have to check all element values matching to the target because in the next dimensions the target might behave differently. So the result has to be a combination of all points a single request has in the n-dimensional space. In the steps 4b and 4c, the value can be taken without checking it. This is because all edge points of the policy targets are indexed. So the interval is from one target to the next target in this direction. Everything in between can be handled equally.

13.2.2. Insert Policy

To insert a value the linked list and the index tree must be expanded. As a first step it must be checked whether the lower and the upper bound of the target area is already existing. If not, the missing bounds have to be inserted in the tree and the linked list has to be expanded. The idea to expand the linked list is to substitute an interval containing the inserted value with an interval less than the value. Figure 11 shows an example where the value 3 is inserted by substituting the interval]2 : 4[with]2 : 3[, 3,]3 : 4[. The value node is then linked with the middle linked-list entry.

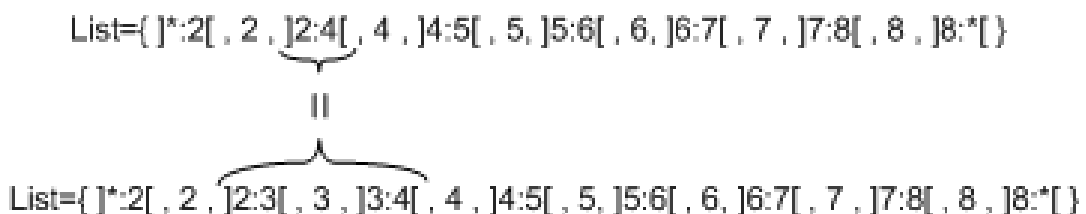


Figure 11: Linked list expansion

Once all nodes exist the policy can be expanded by starting by the upper bound and inserting it in all entries until the lower bound is reached.

insertPolicy

1. Input: policy = p, rootNode = n, dimensionNr = d
2. lowerValue = p.getMatch(d).lowerBound
3. upperValue = p.getMatch(d).upperBound
4. n = **insertValue(lowerValue, n, p.id)**
5. n = **insertValue(upperValue, n, p.id)**
6. listEntry = n.searchNode(upperValue)
7. **appendPolicy(listEntry, p, lowerValue, d)**

insertValue

1. Input: value = v, rootNode = n, policy-id
2. If rootNode = null
 - a) n = new Node(v)
 - b) n.listEntry = new ListEntry(v, policy-id)
 - c) n.listEntry.appendLeft(new ListEntry)
 - d) n.listEntry.appendRight(new ListEntry)
 - e) return n
3. Search leave l where value matches or no more leaves exist
 - a) If (l.value = value)
 - i. l.listEntry.append(policy-id)
 - b) If(l.value < value)
 - i. l.listEntry.appendRight(l.listEntry.right.deepClone())
 - ii. l.listEntry.appendRight(l.listEntry.right.deepClone())
 - iii. l.listEntry.right.right.value = v
 - iv. l.listEntry.right.right.append(policy-id)
 - v. entry = l.listEntry.right.right
 - c) If(l.value > value)
 - i. l.listEntry.appendLeft(l.listEntry.left.deepClone())
 - ii. l.listEntry.appendLeft(l.listEntry.left.deepClone())

- iii. `l.listEntry.left.left.value = v`
- iv. `l.listEntry.left.left.append(policy-id)`
- 4. if `l.value != value`
 - a) `l = n.insertNode(v)`
 - b) `l.listEntry = entry`
- 5. return `n`

By appending the `deepClone` twice and setting the value to the middle, the interval that existed before is splitted into two intervals and the value that still represents all values the interval had before. The intervals have no values so they must not be changed when the interval range changes. The `policy-id` is appended to the edge node to make it easier to remove a policy. If this list contains only the removed policy the node can be removed in the other case there are still other policies having the node as edge boundary.

appendPolicy

1. Input: `listEntry l`, `policy = p`, `lowerBound = v`, `dimension = d`)
2. Check `l.content`
 - a) if `l.content = list-of-Policies` -> `l.content.add(p)`
 - b) if `l.content = next-dimensional-tree-root` -> **`insertPolicy(p, l.content, d + 1)`**
 - c) if `l.content = empty`
 - i. if `policy.getTargetDimensions = d` -> `l.content.add(p)`
 - ii. else `l.content = insertPolicy(p, null, d + 1)`
3. if (`l.value` is not set OR `l.value != v`) -> **`appendPolicy(l.left, p, v, d)`**

As the policy has to be added to every entry between the bounds it is enough to just append it until the lower bound is reached.

13.2.3. Remove Policy

To remove a policy it has to be checked whether its boundaries are still required. If not, the value and at least one of the intervals can be removed. In all list-entries between the boundaries the policy has to be removed too.

removePolicy

1. input: rootNode = n, policy p, dimension = d
2. lowerValue = p.getMatch(d).lowerBound
3. upperValue = p.getMatch(d).upperBound
4. leave = n.searchNode(upperValue)
5. l = leave.listEntry
6. **removeFromList(l, p, d, lowerValue, true)**
7. lowerLeave = n.searchNode(lowerValue)
8. if leave.listEntry.policyIds.size = 0 -> n = removeNode(n, lowerValue)
9. if lowerLeave.listEntry.policyIds.size = 0 -> n = removeNode(n, lowerValue)

removeFromList

1. Input: entry = l, policy = p, dimension = d, lowerValue = v, upperBound
2. l.removep.id
3. if l.policyIds.size = 0 AND (upperBound OR l.value = v)
 - a) if upperBound
 - i. l.right.removeLeft().removeLeft();
 - ii. **removeFromList(l.right.left, p, d, v, false)**
 - iii. return
 - b) if not upperBound
 - i. l.left.removeRight().removeRight();
 - ii. **removeFromList(l.right.left, p, d, v, false)**
 - iii. return
4. if l.policyIds.size > 0
 - a) Check content
 - i. l.content = set-of-policies -> l.content.remove(p)
 - ii. l.content = root-node-of-next-dimension -> removePolicy(l.content, p, d + 1)
5. if(l.value != v) -> **removeFromList(l.left, p, d, v, false)**

The differentiation between the lower and the upper bound is required to be sure that the values in between the bounds of the policy target are removed and not the values outside.

13.3. Limiting behavior

In this section the limiting behavior of the index solution is showed. In all examples n is the number of policies and d is the number of dimensions. To get the limiting behavior, a balanced binary tree as tree structure and a standard double-linked list is taken.

13.3.1. storage

In a first step the behavior in the one dimensional case is shown. Afterwards this is expanded to show the behavior in the n -dimensional case.

A policy has two bounds in each dimension. So the amount of indexed points is $2n$. Each of this bounds is indexed in the binary tree. As a binary tree requires storage of $O(n)$ the storage required by the binary tree is $O(n)$. The linked list contains intervals between all points so the required storage by the double-linked list contains $2*(2n) + 1$ entries. Taking this together, the storage requirement in the one dimensional case is $O(n)$.

In the two dimensional space, the most storage is required if the policies are enclosing each other. So the first policy has the biggest rectangle. The indexed space of the second policy lies fully in the space of the first policy and so on. The case with 3 policies can be seen in figure 12.

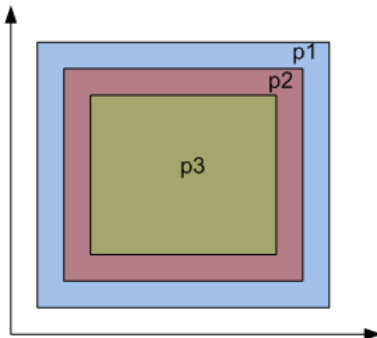


Figure 12: Worst-case storage requirements area example

Starting in the first dimension there are areas with 1 policy in the next dimension some with 2 and in the middle there are again 3 policies in the next dimension. So the storage requirements are the first dimension with $O(n)$ storage requirements and then the cases for the next dimensions $\sum_{i=0}^n (O(i)) = O(n^2)$. So the storage requirement for two dimensions is $O(n^2)$. As in the step from one dimension to two dimension, for each case in the two dimensional example an expansion to the next dimension can be done. By doing this for more dimension a worst case storage requirement of $O(n^d)$ can be shown.

13.3.2. Read

Getting a value out of a balanced binary tree is $O(\log n)$. As reading a value is only getting the matching node in the binary tree and then taking the linked value or one of its neighbors the read access for one match value is $O(\log n)$. A request can contain k values that have to be applied to the target. So the overall read access in the one dimensional case has a complexity of $O(\log(n) * k)$. As shown in Figure 12 every dimension can contain as many policies as the first dimension, so the worst case complexity is the same for every dimension. Thus read access will require $O(d * (\log(n) * k))$

13.3.3. Insert

First we will take a look at the one dimensional case. Inserting the upper bound takes $\log(n)$ because a binary tree is used. Copying the list-entry takes as much time as its content is. So in the worst case this is n . Afterwards all entries have to be updated. In the n -dimensional case copying a list-entry takes as much time as the storage used in a node. Summing everything together, the resulting complexity for an insert operation is $O(n^d)$ in the worst-case scenario.

13.3.4. Remove

Searching the boundary nodes in one dimension requires $O(\log n)$ time. Removing an entry from a linked list can be taken in $O(1)$ and removing a node from a binary can be done in $O(\log n)$. Because of this we can ignore the case where a node has to be deleted simply because its complexity is not bigger than finding a node in the binary tree. The worst case in removing a policy is when the policy applies over the full space. So all $2n+1$ entries in the linked list have to be touched. In the multi dimensional case also all entries of the linked-lists for the next dimensions have to be touched. As shown in section 13.3.2 there are n^d such linked list entries. So the worst case complexity to remove a policy is $O(n^d)$.

13.4. Implementation hints

In this section some hints about the implementation of the index will be given. The algorithms in the chapter before have been defined as simple as possible. By inserting some complexity it will be possible to save some storage space or improve the read access performance.

13.4.1. Tree independant

If the index should be used for big amounts of data it will make sense to hold the index in the secondary storage. In this case the B-tree can be replaced by a B-tree. If a B^+ -tree is taken as implementation it will make sense to implement the leaves differently and introduce the double-linked-list in the leaves of the B^+ -tree.

13.4.2. String Dimensions

For String dimensions it will make sense to use patricia tries instead of binary trees. If the index should be stored in the secondary storage the solution of the index fabric can be taken to persist parts of the patricia trie on the secondary storage.

13.4.3. Regex Optimization

All index but the ones with a fixed string and wildcharacters at the end (e.g. *ab.**) require a lot of single entries or small intervals in one dimension to index all possible strings matching the index. As large intervals have only a few bounds but multiple small intervals have more bounds, it makes it easier only to have a wildcharacter at the end.

In case of wildcard characters at the beginning, the regular expression and all strings of the dimension can be taken in it's reverse order.

13.4.4. Adjected Bounds

Using the described algorithm two nodes with values always have an interval node in between. If the values of two nodes are next to each other (e.g. a node with the integer value 6 is next to a node with the integer 7) the interval value in between can be removed. If a node with no interval value on its left or right side must be removed the value of the node can be deleted and the policy, of which it was an edge, can be removed from the content. If the value is removed, the node can now act as interval node.

13.4.5. Any match

In the previous part we introduced an `AnyMatchFunction`. In practice it will make sense to not use an implementation of this function but to assume it exists implicitly. So if a policy has no target in a dimension the smallest value of the linked list can be searched and the policy can just be appended to all entries of the linked-list.

13.4.6. Multivalued types

To map multivalued types two ways can be used. The first one is to order the values and just build a string with a separator. The second way is to build it to multiple dimension. Every value has it's own dimension and the whole type value is a hyperrectangle in this dimensions.

13.4.7. Specialized `urn:oasis:names:tc:xacml:1.0:function:x500Name-match`

If it is known that all targets of all policies match over the same attributes in the `x500Name` the `x500Name` can be treated like a multivalued type with each attribute as value.

13.4.8. Choosing the dimension

As seen in section 13.3 the behavior of the index can be improved by ordering the dimensions correctly. So it will make sense to allow the user to define which dimensions should be indexed and in which order the dimensions should be indexed.

14. Conclusion

The created index is really good for read operations. Even though update can be very slow if there are a lot of intersections between the policies. This means that the policies have to be created carefully and with respect to the index. As the index splits up the policies dimension by dimension the first dimension should be the one resulting in as less intersections in the next dimension as possible. If this is not done the index is nearly impossible to update in the case of a high number of policies.

All in all we believe that the index is very good if a lot of read operations apply and only a few updates happen. If the policies can not be designed in a way that that the policies differ in a few dimensions another index should be chosen.

Part V. APPENDIX

A. Glossary

PDP	Policy Decision Point
XACML	eXtensible Access Control Markup Language
OASIS	Organization for the Advancement of Structured Information Standards
HERAS ^{AF}	Holistic Enterprise-Ready Application Security Architecture Framework

B. Example Code

To test if the algorithms work, a small proof of concept was made. To make the implementation easier a simple binary tree implementation is used. This is just a quick and dirty implementation so an implementation for productive usage should be structured in a better way.

In the following subsections, the important classes are listed. The classes not listed are only stub implementations.

B.1. Index.java

```
import java.util.HashSet;
import java.util.Set;

public class Index {
    public static Node indexRoot;

    public static void insertPolicy(Policy pol) {
        insertPolicy(indexRoot, pol, 0);
    }

    public static Set<Policy> getPolicies(Request r) {
        return indexRoot.getPolicies(r, indexRoot);
    }

    @SuppressWarnings("unchecked")
    public static void insertPolicy(Node root, Policy pol, int matchIndex) {
        PolicyTargetMatch match = pol.targetMatchList.get(matchIndex);

        if (root == null) {
            if (root == null) {
                root = createRootNewNode(pol, match);
                indexRoot = root;
            }
        } else {
            extendIndex(root, pol, match.lowerBound);
        }
        ListEntry listEntry = extendIndex(root, pol, match.upperBound);
        boolean nodeMatchesValue = true;
        while (nodeMatchesValue) {
            if (matchIndex == pol.targetMatchList.size() - 1) {
                if (listEntry.content == null) {
                    listEntry.content = new HashSet<Policy>();
                }
                ((Set<Policy>) listEntry.content).add(pol);
            } else {
```

```

        if (listEntry.content == null) {
            listEntry.content = createRootNewNode(pol,
                pol.targetMatchList.get(matchIndex + 1));
        }
        insertPolicy((Node) listEntry.content, pol, matchIndex + 1);
    }

    if (listEntry.value != null && listEntry.value == match.lowerBound) {
        nodeMatchesValue = false;
    }
    listEntry = listEntry.left;
}
}

private static Node createRootNewNode(Policy pol, PolicyTargetMatch match) {
    Node root;
    root = new Node(match.lowerBound);
    root.listEntry = ListEntry.createInitialEntries(pol, match.lowerBound);
    root.indexedElementId = match.elementId;
    return root;
}

private static ListEntry extendIndex(Node root, Policy pol, Integer val) {
    Node node = root.getBestMatchingNode(val);
    if (node.value == val) {
        node.listEntry.append(pol);
    } else {
        ListEntry entry = node.listEntry;
        int compValue = node.value.compareTo(val);
        node = Node.insertNode(root, val);
        node.indexedElementId = root.indexedElementId;

        if (compValue < 0) {
            node.listEntry = entry.insertRight(pol, val);
        } else {
            node.listEntry = entry.insertLeft(pol, val);
        }
    }
    return node.listEntry;
}

public static void remove(Policy pol) {
    indexRoot = Node.removePolicy(indexRoot, pol, 0);
}
}
}

```

B.2. Node.java

```

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

```

```
public class Node implements Serializable {
    private static final long serialVersionUID = 1L;

    public Integer value;

    public Node leftChild;
    public Node rightChild;

    public ListEntry listEntry;
    public String indexedElementId;

    public Node(Integer val) {
        this.value = val;
    }

    @SuppressWarnings("unchecked")
    public Set<Policy> getPolicies(Request r, Node n) {
        Set<Integer> vals = r.getValues(n.indexedElementId);
        Set<Policy> result = new HashSet<Policy>();
        for (Integer val : vals) {
            Node leaf = n.getBestMatchingNode(val);
            ListEntry entry;
            if (leaf.value == val) {
                entry = leaf.listEntry;
            } else if (leaf.value < val) {
                entry = leaf.listEntry.right;
            } else {
                entry = leaf.listEntry.left;
            }
            if (entry.content instanceof Set) {
                result.addAll((Set<Policy>) entry.content);
            } else if (entry.content instanceof Node) {
                result.addAll(((Node) entry.content).getPolicies(r,
                    (Node) entry.content));
            }
        }
        return result;
    }

    public Node getBestMatchingNode(Integer val) {
        if (this.value > val) {
            if (this.leftChild != null) {
                return this.leftChild.getBestMatchingNode(val);
            }
        }

        if (this.value < val) {
            if (this.rightChild != null) {
                return this.rightChild.getBestMatchingNode(val);
            }
        }
        return this;
    }

    public static Node insertNode(Node root, Integer val) {
        Node closest = root.getBestMatchingNode(val);
        Node insertedNode = new Node(val);

        if (closest.value.compareTo(val) < 0) {
            closest.rightChild = insertedNode;
            return insertedNode;
        }
    }
}
```

```
    }
    if (closest.value.compareTo(val) > 0) {
        closest.leftChild = insertedNode;
        return insertedNode;
    }
    return closest;
}

public static Node removePolicy(Node root, Policy pol, int dimension) {

    PolicyTargetMatch match = pol.targetMatchList.get(dimension);
    Node upperNode = root.getBestMatchingNode(match.upperBound);
    Node lowerNode = root.getBestMatchingNode(match.lowerBound);

    upperNode.listEntry.remove(pol, match.lowerBound, dimension, true);
    if (upperNode.listEntry.edgePolicyId.size() == 0) {
        root = root.remove(upperNode.value);
    }
    if (upperNode.value != lowerNode.value
        && lowerNode.listEntry.edgePolicyId.size() == 0) {
        root = root.remove(lowerNode.value);
    }
    return root;
}

private Node remove(Integer value) {
    if (this.value == value) {
        if (this.leftChild == null) {
            // Case left and right == null
            if (this.rightChild == null) {
                return null;
            }
            // left == null, right != null
            return this.rightChild;
        }

        if (this.rightChild == null) {
            // right == null, left != null
            return this.leftChild;
        }
        Node minNode = rightChild.getMinReplaceNode();
        minNode.leftChild = this.leftChild;
        minNode.rightChild = this.rightChild;
        return minNode;
    }
    if (this.value < value) {
        this.rightChild = rightChild.remove(value);
        return this;
    }
    this.leftChild = leftChild.remove(value);
    return this;
}

private Node getMinReplaceNode() {
    if (this.leftChild == null && this.rightChild == null) {
        return this;
    }
    if (this.leftChild == null && this.rightChild != null) {
        return this.rightChild.getMinReplaceNode();
    }
    return leftChild.getMinReplaceNode();
}
```

```
}  
  
public String toString() {  
    return "Value: " + value + ", leftChild: (" + leftChild  
        + "), rightChild: (" + rightChild + ")";  
}  
}
```

B.3. ListEntry.java

```
import java.io.ByteArrayInputStream;  
import java.io.ByteArrayOutputStream;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.io.Serializable;  
import java.util.HashSet;  
import java.util.Set;  
  
public class ListEntry implements Serializable, Cloneable {  
  
    private static final long serialVersionUID = 1L;  
    public ListEntry left;  
    public ListEntry right;  
    public Object content;  
    public Set<String> edgePolicyId = new HashSet<String>();  
    public Integer value;  
  
    public ListEntry insertRight(Policy pol, Integer val) {  
        ListEntry newListEntryWithValue = this.right.clone();  
        newListEntryWithValue.append(pol);  
        linkRight(newListEntryWithValue);  
        linkRight(this.right.clone());  
  
        newListEntryWithValue.value = val;  
        return newListEntryWithValue;  
    }  
  
    private void linkRight(ListEntry newRightEntry) {  
        ListEntry oldRightEntry = this.right;  
        oldRightEntry.left = newRightEntry;  
        newRightEntry.right = oldRightEntry;  
        this.right = newRightEntry;  
        newRightEntry.left = this;  
    }  
  
    public ListEntry insertLeft(Policy pol, Integer val) {  
        ListEntry newListEntryWithValue = this.left.clone();  
        newListEntryWithValue.append(pol);  
        linkLeft(newListEntryWithValue);  
        linkLeft(this.left.clone());  
        newListEntryWithValue.value = val;  
        return newListEntryWithValue;  
    }  
  
    private void linkLeft(ListEntry newLeftEntry) {  
        ListEntry oldLeftEntry = this.left;  
        oldLeftEntry.right = newLeftEntry;  
        newLeftEntry.left = oldLeftEntry;  
    }  
}
```

```

    this.left = newLeftEntry;
    newLeftEntry.right = this;
  }

  public void append(Policy pol) {
    edgePolicyId.add(pol.id);
  }

  /**
   * Everything except the policies are deep cloned
   */
  @SuppressWarnings("unchecked")
  @Override
  protected ListEntry clone() {
    try {
      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ObjectOutputStream oos = new ObjectOutputStream(baos);
      oos.writeObject(this);
      ByteArrayInputStream bais = new ByteArrayInputStream(baos
        .toByteArray());
      ObjectInputStream ois = new ObjectInputStream(bais);
      ListEntry deepCopy = (ListEntry) ois.readObject();
      if (deepCopy.content instanceof Set) {
        ((Set<Policy>) deepCopy.content).clear();
        for (Policy pol : (Set<Policy>) this.content) {
          ((Set<Policy>) deepCopy.content).add(pol);
        }
      }
      return deepCopy;
    } catch (Exception e) {
      e.printStackTrace();
    }
    return null;
  }

  @SuppressWarnings("unchecked")
  public void remove(Policy pol, Integer lowerBoundValue, int dimension,
    boolean upperBound) {
    this.edgePolicyId.remove(pol.id);
    if (this.edgePolicyId.size() == 0) {
      if (this.left.left == null) {
        if (upperBound && this.right.right != null) { // there is no ←
          lower bound
          this.right.right.left = this.left.right;
          this.left.right = this.right.left;
          return;
        }
        if (this.right.right != null){ //If this.right.right == null, ←
          it's the only value left and it will be removed anyway.
          this.right.right.left = this.left;
          this.left.right = this.right.right;
          return; //It must be the lower bound, because there is no ←
          value on the left
        }
      }
      else { // no change to the upper bound is required, as policy ←
        is already removed from right node.
        this.left.left.right = this.right;
        this.right.left = this.left.left;
      }
    }
  }

```

```
    } else {
        if (this.content instanceof Set) {
            ((Set<Policy>) this.content).remove(pol);
        } else {
            this.content = Node.removePolicy((Node) this.content, pol,
                dimension + 1);
        }
    }
    if (!lowerBoundValue.equals(this.value)) {
        left.remove(pol, lowerBoundValue, dimension, false);
    }
}

public static ListEntry createInitialEntries(Policy pol, Integer val) {
    ListEntry middleEntry = new ListEntry();
    middleEntry.edgePolicyId.add(pol.id);
    middleEntry.value = val;

    middleEntry.left = new ListEntry();
    middleEntry.right = new ListEntry();
    return middleEntry;
}

public String toString() {
    if (this.value == null) {
        return "null";
    } else {
        return this.value.toString();
    }
}
}
```

C. Thesis Appendix

C.1. Thesis Overview

C.1.1. Target

- Policy indexing
- Description how XACML policies can be indexed
- Worst case analysis of the algorithm

C.1.2. People

Josef Joller The responsible lecturer. He is Professor at the University of Applied Sciences Rapperswil and is supporting Stefan Oberholzer during his thesis.

Stefan Oberholzer The author of the thesis. He is Master Degree Student at the University of Applied Sciences Rapperswil.

C.1.3. Dates

Project start: 14. September 2009
Project end: 14. Februar 2010

C.2. Project Plan

Phase	Start	Ende
Read in DB indexes	14.09.2009	25.10.2009
Read related work	26.10.2009	22.11.2009
Create index	22.11.2009	29.01.2010

The rest of the time is reserved for unforeseen circumstances and test preparation.

C.3. Risk Management

Risk	Impact	Action	Costs of the action (in man hours)	Highest losses (in man hours)	Chance to happen (in percent)	Weighted losses (in man hours)	Priority
R01: Crash of the infrastructure	Loss of data	Regular backup	1	48	1	0.5	0
R02: Injury or illness	Thesis stops	-	0	252	1	25	0
R03: Bad quality of literature	A lot more research work to do	Exact literature preparation	10	40	10	4	0
R04: Complexity of the research underestimated	A lot more research work to do	-	0	40	15	6	0
Total hours for the actions			12				
Total hours for standby						65.5	

C.4. Time Evaluation

Overall the thesis took 290 hours. As it can be seen in Figure 13 most of the time was used to create and describe the policy index.

C.5. Personal Statement

As it was my own idea to write a thesis about indexing policies I was very motivated to write this thesis.

It was my target to create an index leading to a very good read access time. As an index only make sense on appropriate data it was clear, that the policies have to be created in a way optimized for the index. I believe that the insert time is acceptable if the dimensions are in the correct order and the policy targets are created in a way optimized for this algorithm. So I believe that an implementation of this index would be usable in practise.

To manage the high work load I created a plan when I have to do what for which module. Unfortunately seminar module took much more time then expected. As I

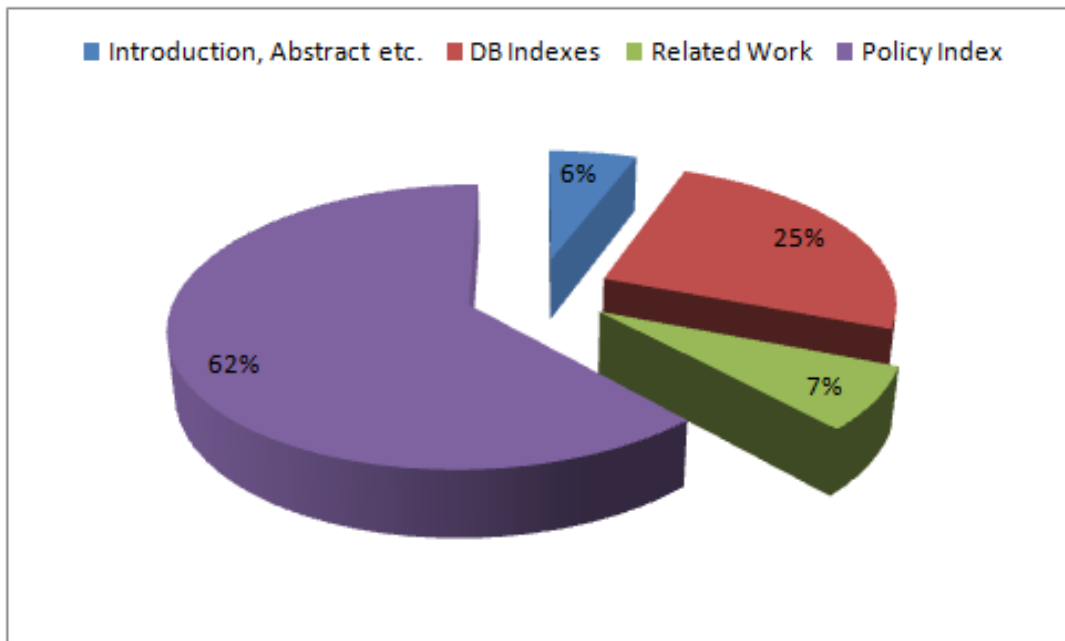


Figure 13: Work per part

had to succeed in this module because failing would have meant that my study fails i had to stop working on the thesis and finish the module. As a result of this I was very short in time and had to do the work in the thesis in a very short time period. I think I only managed to get to a result because I had a good idea about how an index over policies could look like and because I didn't stop thinking about the index while I was working for the module. So most of the ideas how policies can be indexed already existed when I was able to continue with this work.