Efficient Matching in Heterogeneous Rule Engines

Kambona, Kennedy; Renaux, Thierry; De Meuter, Wolfgang

Published in:
Advances in Artificial Intelligence

DOI:
10.1007/978-3-319-60042-0_44

Publication date:
2017

License:
Unspecified

Document Version:
Accepted author manuscript

Link to publication

Citation for published version (APA):
Kambona, K., Renaux, T., & De Meuter, W. (2017). Efficient Matching in Heterogeneous Rule Engines. In M. Ali, S. Benferhat, & K. Tabia (Eds.), *Advances in Artificial Intelligence: From Theory to Practice - 30th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2017, Proceedings* (Vol. 10350, pp. 394-406). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 10350 LNCS). Springer. https://doi.org/10.1007/978-3-319-60042-0_44

Efficient Matching in Heterogeneous Rule Engines

Kennedy Kambona, Thierry Renaux[⋆], and Wolfgang De Meuter

Vrije Universiteit Brussel, Pleinlaan 2, Brussels, Belgium
`{kkambona,trenaux,wdmeuter}@vub.ac.be`

Abstract. Modern institutions seeking more complex software solutions to represent knowledge in the Cloud are using rule-based systems that serve several applications or clients. Rule-based systems hosted in the Cloud are thus required to support its heterogeneous nature. However, current systems only focus on techniques that isolate instances of rule engines. This paper builds upon earlier work on scoped rule engines that provide mechanisms for supporting shared heterogeneous contexts. We present the scope-based hashing algorithm (SBH) that enables efficient matching in scoped rule engines based on the Rete algorithm. SBH introduces scoped hash tables in alpha memories that help in avoiding unnecessary join tests that hamper performance. Our experimental results show that SBH offers significant improvements in efficiency during the matching process of a heterogeneous rule engine. Consequently, SBH significantly decreases the response time of rule engines in heterogeneous environments having entities sharing the same knowledge base.

Keywords: Heterogeneity, rule engines, Rete algorithm, scoping

1 Introduction

As the Internet matures, modern institutions are seeking more complex software solutions for their operations in cloud service providers. One area that is gaining momentum is in the provision of complex services for knowledge representation using rule-based definitions, e.g., IBM ODM Decision Server [6] and Amazon IoT Rules [2]. Such rule-based systems (or RBS) are known to use complex event processing for reasoning about events of interest to business applications. A rule-based language is often used for programming definitions of event patterns because rules are intuitively appealing to express [3].

Rule engines were fundamentally designed in the era where isolated computing was prevalent: at the time, rule engines were programmed to encode a localised set of rules and to work on homogeneous data [13]. Rule engines were therefore characterised by a *flat design space* where activations could be observed from all data without discriminating their sources. When deployed to hybrid or heterogeneous environments, RBS need manual interventions to enforce rule and data isolation [13]. Previously, the concept of isolation in modern rule engines

[⋆] Supported by a doctoral scholarship of the Agency for Innovation by Science and Technology in Flanders (IWT), Belgium

was manually enforced through the use of separate rule engines using rulebooks or rule modules to separate instances of rulesets from different clients or event sources. Recent work has provided a solution by introducing scoping in the rule engine through the use of scoped rules [12]. Scoped rules support heterogeneous RBS by exposing formalised mechanisms in which applications can perform data isolation in rule definitions for different clients. Scoped rules keep isolation logic cleanly separated from the application logic: the basic purpose of the rule is not mixed with the logic required for distinguishing client data. The result is that the logical intent of a rule becomes easier to understand by rule creators.

In this work we propose a novel improvement in rule engines that improves matching efficiency in scope-aware RBS. Our technique involves an inventive optimisation to the popular Rete algorithm during the matching process, identified as the most computationally-intensive execution phase of any Rete-based rule engine [7]. The *scope-based hashing* (SBH) approach utilises the scoped hash tables and fact metadata to exclusively and efficiently compute compatible data that is relevant for computing joins in heterogenous environments.

The contributions of this work are as follows. We extend the work on scoped inference engines with a novel algorithm that provides an optimisation to the underlying Rete algorithm. In particular, our approach extends the Rete network within an inference engine with scoped hash tables in the alpha memories that are used to efficiently determine compatible data in heterogeneous contexts. We further justify our scheme by performing an evaluation using a representative heterogeneous application backed by a rule engine utilising our approach, comparing the results with the state-of-the-art.

2 Motivating Example

To motivate the concepts that this work proposes we present a practical scenario of a security monitoring system deployed to serve a number of departments across universities. A sample computer labs policy is defined below:

> "*All main campus master's thesis students in the science faculty are allowed extended access to computer labs in their own departments until 10pm weekdays while in the final stages of their theses (in the months between March and August). Students in the department of computer science are additionally allowed on the weekends between 10-16h.*"

A typical structure of a university is depicted in Figure 1a. The structure shows various entities as groups and users are connected to one or more groups in the hierarchy. Staff and students are issued readable ID cards and are required to scan their IDs on devices strategically placed at access points to gain entry. Every access request on a device is relayed to a central server that logs and processes the request according to the defined policies. Any granted accesses according to the defined security policies should be promptly shown on the dashboard of the security team's interface. In this scenario, we have articulated around 40 such policies across different faculty and departments of universities. A service provider can support several of such universities with their own policies.

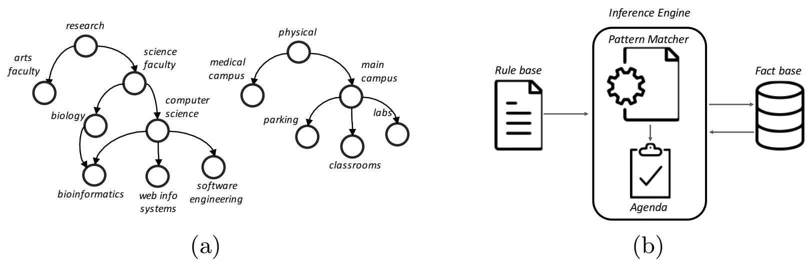


Fig. 1: (a) Typical university group hierarchy – Represented as acyclic graphs of hierarchies of departments for students and physical locations for scanning devices (b) Rule engine architecture – Rules from clients are added to the global rule base and event data as facts in the global fact base.

3 Reactive Rule Engine Architecture

The motivating example represents a typical reactive application that exposes services to clients contributing data intermittently. It quickly becomes apparent that the policies can be suitably captured in traditional rule-based syntax, and the intermittent access requests from different student entities can be easily captured and processed by a rule engine.

In this work we focus on the most computationally expensive phase in an RBS, the matching process. Because the system *reacts* when the data is sent, the approach should provide a data-driven mechanism for reasoning through forward-chaining. In this work we therefore employ the most efficient algorithm used in forward-chaining RBS, the Rete algorithm [9]. In a typical Rete-based system (Figure 1b), rules are added into the rule base and event data from other client devices such as sensors are constructed as facts and added into the fact base. The inference engine contains the pattern matcher and the agenda which employ Rete to determine which rule to fire given the current state of the engine. A Rete inference engine converts rules into an acyclic graph with intermediate memories that cache intermediate results. This eliminates extra work that would otherwise need to be performed during each matching cycle.

4 Heterogeneity in Rule-Based Systems

This work proposes efficient matching in heterogeneous rule engines. Using the motivating example we illustrate how current rule-based systems deal with heterogeneity and contrast it with recent work involving scoped engines.

4.1 Classic vs. Scoped Rule-based Systems

The lab access rule definition. The policy for science faculty students can be represented in classic rules syntax as shown in Listing 1.1 and using scoped rules in Listing 1.2.

They show a customised JSON Rules [10] syntax sent by a client (which can be built intuitively using a graphical interface). The conditions capture the access request, the student making the request and the device scanned. The test expressions confirm the time constraints of the policy. A separate rule can be similarly designed to determine the second part of the policy specifically for computer science students.

In **classic rules** there is need to determine if the student and device originate from the same department, to ensure the rule will not cause unintended activations. This will avoid students from other faculties having access to the labs they are not a member of (e.g., a student from the arts faculty gaining access to a bioinformatics lab). Because the student can belong to any of the sub-departments, the check needs to confirm if the student or device's group is a descendant of a faculty group. Classical approaches use compatibility checks such as those in Line 6 and 7 of Listing 1.1 through *relation facts* that show a group is related to a particular parent group.

Listing 1.1: Classic rule for lab access

```
1  {rulename: "MastersStudentsLabAccess",
2   conditions:[
3    {type:"accessreq", id: "?reqid", badge: "?badgid", time: "?t", device: "?devid"},
4    {$s: {type:"student", name: "?nam", badge: "?badgid", level: "master", group:"?stugrp"}},
5    {$d: {type:"accessdevice", id: "?devid", group:"?devgrp"}},
6    {type:"belongsTo" grp:"?devgrp", parent:"?pgrp"},
7    {type:"belongsTo" grp:"?stugrp", parent:"pgrp"},
8    {type:"$test", expr:"( month(?t) > 1 && month(?t) < 9 && hour(?t) > 10 && hour(?t) < 16  && isWeekend(?t)"}
9  ],
10  actions:[
11   {assert: {type: "accessrep", reqid:"?reqid", allowed: true}}
12  ]
13 }
```

Listing 1.2: Scoped rule for lab access

```
1  {rulename: "MastersStudentsLabAccess-Scoped",
2   conditions:[
3    {type:"accessreq", id: "?reqid", badge: "?badgid", time: "?t", device: "?devid"},
4    {$s: {type:"student", name: "?nam", badge: "?badgid", level: "master"}},
5    {$d: {type:"accessdevice", id: "?devid"}},
6    {type:"$test", expr:"( month(?t) > 1 && month(?t) < 9 && hour(?t) > 10 && hour(?t) < 16  && isWeekend(?t)"},
7   ],
8   actions:[
9    {assert: {type: "accessrep", reqid:"?reqid", allowed: true}}
10  ],
11  scopes: ["($s & $d) subgroupof science", "$d private labs"]
12 }
```

In contrast, **scoped rules** [12] extend rule syntax to support the definition of flexible constructs that ensure data compatibility between instances of data from different entities during matching. Listing 1.2 shows the same rule as Listing 1.1, but with s special `scopes` construct in line 11 instead of conditions with relation facts. The first scope definition specifies that the data for the rule to be matched should be captured from the `science` group and any of its children or subgroups. The second scope definition specifies that only device data from the group `labs` should privately be matched.

Rete graph for lab access rule. The *MasterStudentsLabAccess* rule is designed by a security staff member and sent to the server. The server builds a Rete graph [9] from the rule, which we show in Figures 2 and 3. The graph performs both intra-condition checks such as `student`, `device` etc. It also performs inter-condition tests, e.g., in node 1 that makes sure that an `access request` is matched with the student that made the request, propagating compatible data as tokens to its children. Node 2 checks the same for the `device`.

Algorithm 1 Beta node left activation

```
function betanodeLeftReceive($node$:$n$,$token$: $t$)
    $facts \leftarrow n$.alphaMemory.getFacts()
    for each fact $f$ in $facts$ do
        if $n$.joinTestPassed($t$, $f$) then
            $t_{new} \leftarrow n$.createNewToken($t$, $f$)
            $n$.sendTokenToChildren($t_{new}$)
        end if
    end for
end function
```

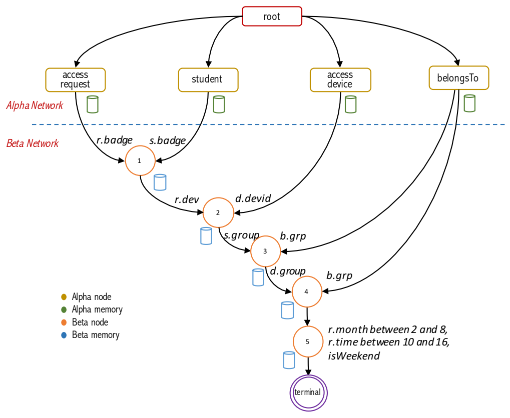


Fig. 2: Left activation and classic Rete graph for lab rule – Relation facts can distinguish between entities in heterogeneous RBS.

Matching in the Rete graph. When a token is received at the left input of any join node, a *left activation* is triggered that issues a request for all the items in its right alpha memory to perform its inter-condition tests – this process is also known as matching and the test is called a *join*. A similar process happens on a right activation. The *beta test* nodes before the terminal node perform processing to check for the time constraints of the policy on the `accessrequest` fact. If a token passes the tests it reaches the terminal node and is added to the agenda for activation of the policy rule, granting access to the student.

For a **classic RBS** (Figure 2), left-activating beta node 3 results in a need to access all the items in the `belongsTo` alpha memory to find relation facts with groups same as its `student`'s group. A similar process happens in beta node 4 for `devices`. This method and other similar approaches for finding compatible data with heterogeneous users sharing the same knowledge base is inefficient and quickly becomes cumbersome: the rule logic becomes difficult to follow and processing is dominated by expensive join computations that are necessary to distinguish incompatible facts from different users or user groups – like during matching in nodes 3 and 4.

In a **scoped RBS** an internal representation of physical or logical organisations of clients is first precomputed, stored and maintained efficiently as an encoding that will be used to expeditiously process constraints used to enforce reentrancy within the inference engine. This is done by constructing a *bit-vector encoding* that allows for near-constant time scope checks during matching, thereby reducing the processing overhead when isolating compatible data matches in heterogeneous contexts. The university group hierarchy in Figure 1a is converted into the matrix encoding M_ϑ shown in Figure 4a through a process that is based on Ait Kaci's method in [1]. In the encoding, there is an entry at $M_{\vartheta(a,b)}$ iff b is an ancestor of a. The engine also automatically adds metadata to all facts added to a client. For instance, if a device sends an access request, the fact is automatically tagged with the device's group when inserted to the rule engine.

Algorithm 2 Betanode Left Activation with Scopes

```
function scopedBetanodeLeftReceive($node : n$, $token : t$)
    $facts \leftarrow n$.alphaMemory.getFacts()
    for each fact $f$ in $facts$ do
        if $this$.scopeCheckPassed($n$,$t$,$f$) then
            if $n$.joinTestPassed($t$, $f$) then
                $t_{new} = n$.createNewToken($t$, $f$)
                $n$.sendTokenToChildren($t_{new}$)
            end if
        end if
    end for
end function
```

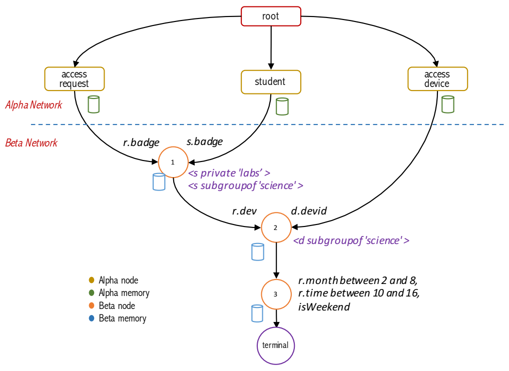
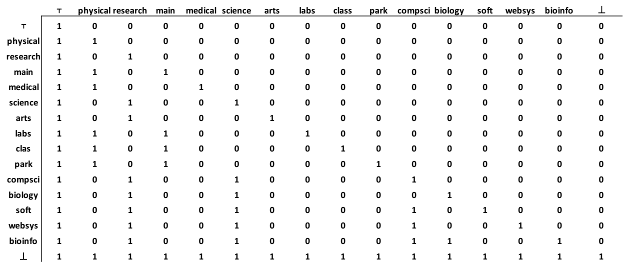
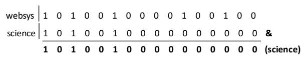


Fig. 3: Scoped left activation and scoped Rete graph – Scoped approach adds scope checks to beta nodes at opportune node locations.



	⊤	physical	research	main	medical	science	arts	labs	class	park	compsci	biology	soft	websys	bioinfo	⊥
⊤	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
physical	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
research	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
main	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
medical	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
science	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
arts	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
labs	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0
clas	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0
park	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0
compsci	1	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0
biology	1	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
soft	1	0	1	0	0	1	0	0	0	0	1	0	1	0	0	0
websys	1	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0
bioinfo	1	0	1	0	0	1	0	0	0	0	1	1	0	0	1	0
⊥	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(a)



websys	1	0	1	0	0	1	0	0	0	0	1	0	0	1	0	0	
science	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	&
	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	(science)

(b)

Fig. 4: (a) Matrix encoding M_ϑ – The rows and columns refer to groups in a hierarchy. (b) Example of a bitwise scope check operation – The operation confirms that `websys` is a part of the `science` faculty.

Figure 3 shows the Rete graph built from the scoped rule. In the graph of a scoped engine, *scope checks or guards* are added to beta nodes 1 and 2. The `<s subgroupof science>` guard can be interpreted as check for any bound fact for a `student` (in the alpha node for the condition in line 4 of rule Listing 1.2) that is a member of any subgroup of the `science` faculty in the university. When an access request is made its token is received in beta node 1 through the `accessrequest` alpha node and a left activation is triggered. The algorithm follows the steps shown in Algorithm 2. The main difference with Algorithm 1 is that before performing the join test, the node first runs a near constant-time scope check in each fact from the alpha memory using the matrix encoding M_ϑ illustrated in Figure 4a. For example, when a student from `websys` requests access on the entrance of the `compsci` labs, the `<s subgroupof science>` check performs a bitwise `AND` for the row encodings of the two groups and compares the result to the row encoding of the `science` group as shown in Figure 4b. If the scope check fails then the data is incompatible and computation moves onto the next fact.

In essence, the processing outlined in Algorithm 2 using scopes is more adept compared to the relation fact technique or other current approaches due to the fast encoded binary tests. However, it is clear that with each left activation, of a beta node scope checks **are still performed on every fact** in the alpha memory, regardless. This makes the approach inefficient. To this end, we present an approach that makes join computations in Rete networks more efficient using the scope-based hashing algorithm (SBH).

5 The Scope-based Hashing Algorithm

Alpha Memory Hashing via Groups. Alpha memories in Rete can be simply viewed as nodes that store facts of a particular type, e.g., the memories of `student` and `device` nodes in figure 3. The purpose of alpha memories is to supply beta nodes with fact items. As event data is added to the engine the cached dataset increases especially in heterogeneous rule engines since multiple users and their devices all contribute data. The result is that although scope-based rule engines offer a reasonable improvement when computing joins, the rule engines still suffer when performing scope checks for **every** data item added in the alpha memory.

We propose an approach that improves this scope-checking process. Our technique constructs a hash table that dynamically assigns buckets based on user groups in the group hierarchy (e.g. the groups in Figure 1a). Each bucket points to a list that holds a set of facts of that group. As facts are added to the system SBH assigns each fact to the correct bucket dynamically. For instance, for a device located at the entry point of the science department the fact will be added to the `science` bucket of the `student` SBH hash table.

Matching with Scope-based Hashing. The matching process stage is where any rule engine performs most of its computation. Matching in scopeful engines involves updating the beta network with scope guards that check compatibility of left and right inputs. SBH introduces a way to efficiently determine which fact items are compatible with an incoming token at a beta node to be subsequently used in the join test of the node.

Take an example of an access request made by a student in `web info systems` on a device located at the `computer scicience` labs. This is a valid access request that should be granted (assuming it is made in the correct timeline). Using the Rete graph of the same policy (figure 5), the fact will trickle down to beta node 1 causing a left activation. A left activation with the SBH technique is shown in Algorithm 3, where `this` refers to the SBH instance.

We describe the algorithm using the scope guard `<$s subgroupof science>` in beta node 1. In this case, `s` is bound to the fact representing the `websys` student. Instead of performing the check with every `student` fact in the alpha memory, SBH retrieves the matrix codes for all the subgroups of `science` via `calculateCodeFromScopeGuards`.

Let n be the total number of elements of a row in the encoding matrix M_ϑ (Figure 4a). `calculateCodeFromScopeGuards` constructs a bit vector V_n with all elements having a `0`. Conceptually, the V_n represents all groups in the hierarchy. At this point, no groups have passed the scope check (all have `0`s as entries in V_n). SBH then performs operations that assign a group element `1` iff it satisfies the scope guard `<s subgroupof science>`. In this case, the method retrieves

Algorithm 3 BetaNode Left Activation w. SBH

```
function scopedBetanodeLeftReceive($node : n$, $token : t$)
 $groupsCode \leftarrow this$.calculateCodeFromScopeGuards($n.scopeTests,\ t$)
 $groups \leftarrow this$.getGroupsFromCode($groupsCode$)
 $scopeFacts \leftarrow n$.alphaMemory.getFacts($groups$)
    for each fact $f$ in $scopeFacts$ do
        if $n$.joinTestPassed($t,\ f$) then
            $t_{new} \leftarrow n$.createNewToken($t,\ f$)
            $n$.sendTokenToChildren($t_{new}$)
        end if
    end for
end function
```

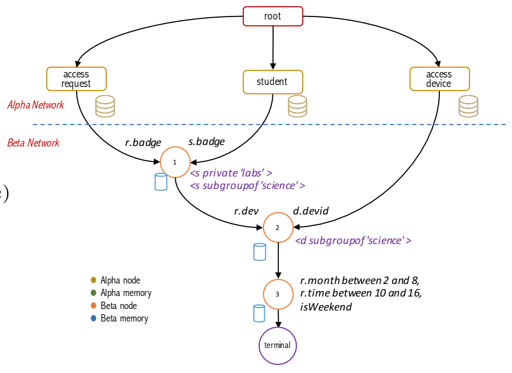


Fig. 5: SBH algorithm and scoped Rete graph with SBH – The alpha memories are replaced with SBH alpha memories that make scope-based computations more efficient

the column vector $M_{\vartheta(*,science)}$= `[0000010000111111]` which represents all the subgroups of `science`.

The next step in Algorithm 3 is to retrieve the corresponding groups in method `getGroupsFromCode` which will then be used to retrieve the items per group in the alpha memory. The method `getGroupsFromCode` retrieves the group names or labels which have a `1` in V_n excluding $\perp$, which in this case V_g=`[science, compsci,biology,soft,websys,bioinfo]`. Retrieving V_g should be relatively easy since the groups have direct mappings to the labels in the matrix.

Next, the `getFacts` method of the alpha memory now accepts V_g as an argument. The method uses the `student` alpha memory's internal SBH table (introduced in Section 5) to efficiently access the fact items that are pertinent to the child beta node, node 1. The alpha memory will thus retrieve the facts residing each of the named groups in V_g from its buckets. Essentially, the facts retrieved are a local subset of all the items in the alpha memory and as such the join computation of node 1 will be performed on a these rather than all the fact items residing in the `student` alpha memory. The rest of the code in Algorithm 3 iterates through all the retrieved items and performs the normal join tests for the node.

Additionally, in reality a number of nodes will have multiple scope expressions in one node: a good example is node 1 which not only has `<s subgroupof science>` but also `<s private labs>`. Furthermore, scope tests can contain complex expressions – to specify "an access device that is in the classrooms of the computer science and biology department or the arts faculty," the expression becomes `<$d subgroupof (science & biology) | $d private arts>`

One option to compute such expressions is to repeatedly call `calculateCodeFromScopeGuards` on each scope test, store multiple vectors of V_g and send these to the alpha memory to retrieve the items needed for a beta node's join computations. A more efficient way that SBH uses is that it performs reductions using bitwise operations given every V_{ni} bit vector result of each scope test i of a beta node. This is used by method `calculateCodeFromScopeGuards` to construct V_n for the scope test,

```
  <d subgroupof (science & biology) | d private arts>
= <(d subgroupof science & d subgroupof biology)
    | d private arts >
= [(0000010000111111 & 0000000000010011) | 0000001000000000]
= 0000000000010011 | 0000001000000000
= 0000001000010011
```

Note that the vector V_{ni} of a scope `<private` u`>` is the unit vector of u. The result V_n is returned from the method `calculateCodeFromScopeGuards`. Next V_g is computed which in this case evaluates to `[arts,bio,bioinfo]`. The SBH algorithm proceeds normally as outlined in Algorithm 3, retrieving the scope facts of groups in V_g and preforming the join tests if the scope check succeeds.

6 Experimental Evaluation

For the evaluation we focus on investigating whether a rule engine with SBH experiences significant improvements in efficiency compared to the current alternative techniques available in contemporary rule engines. The evaluation was based on the complete university security monitoring application staged in an experimental setup as introduced in Section 2.

Setup & Methodology. We performed our evaluation in an experimental setup consisting of a web server running a rule engine based on the Rete algorithm. The server hardware was configured with a AMD Opteron Processor 6272 at 2.1Ghz. The server processes were assigned a maximum of 20GB RAM. To model a practical real-world scenario we designed a user hierarchy of 60 groups in total and 40 typical access policies modelled as rules serving 70 clients concurrently. To simulate practical delays in access requests clients were configured to generate access requests intermittently at intervals of 1-5 seconds and devices received reactive feedback, with a security console receiving push-based updates of accesses to entry points. Each access request was randomised, with a random client belonging to any group(s) making an access request at a device from a random location in the university hierarchy.

We split the experiment into three categories. The first category was running a traditional rule engine without scopes, the second was running a scoped rule engine and the third had a rule engine running the scope-based hashing algorithm. For each category a total of 62 sessions were performed with one session running for a duration of 12 hours. The total experiment therefore spanned 186 sessions and 2332 hours runtime.

Results & Discussion. During the experiment the activation times (comparable to response time) and the memory used were logged and compared. We graphically chart the results using bean charts that show the quartiles as well as the density estimates.

Figure 6a shows the results of the activation times of the unscoped, scoped and SBH rule engines. Rule activation time is the time it takes the engine to perform a matching process, between assertion and rule activation. The chart shows that, on average, the scoped rule engine showed slightly less activation times than

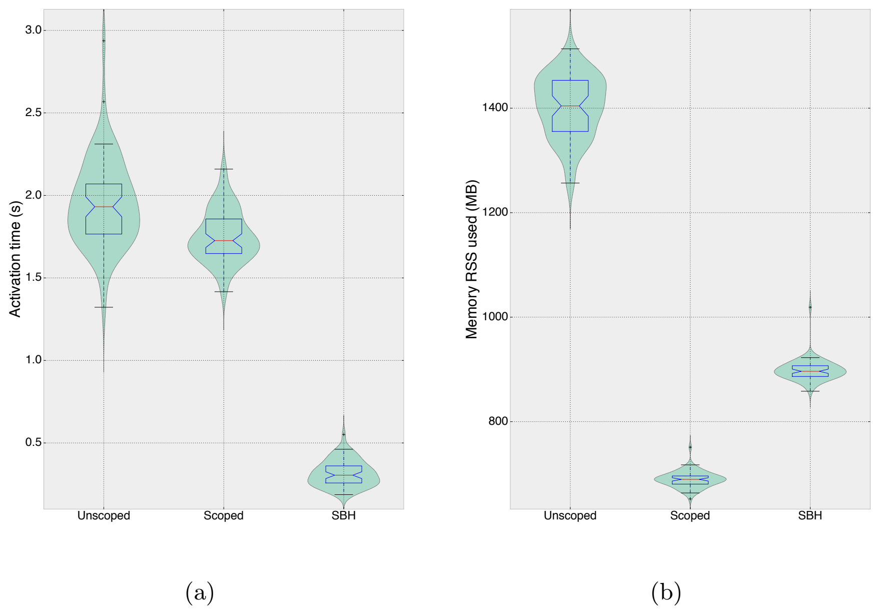


Fig. 6: Bean plots of the results – Results of over 60 runs of random 12-hour sessions show that the SBH approach offers faster activation times than both the scoped and typical unscoped engines. SBH still exhibits less memory usage than traditional unscoped engines, but consumes more memory than a plain scoped approach.

the unscoped engine. Comparing the first two with the SBH approach, it is clear that SBH exhibits an advantage with reduced activation times of up to 80% in some cases. Figure 6b shows the results of the recorded memory consumption (measured by resident set size) averages of each category. The classical approach of managing heterogeneity in rule engines leads to a much higher memory consumption, due to redundant information and inefficiencies brought about by the complexity of enforcing reentrancy. The scoped engine showed a lesser amount of memory consumed by reducing and optimising redundant information used for computing scopes. When we compared the memory usage of the three, the SBH approach is seen to consume up to 30% more than using a pure scoped engine on average, but uses 35% less than a classic unscoped engine. The alpha memory hashing of SBH leads to a more complex node memory structure that needs more space than the conventional structures of node memories.

From the results we observe that for a rule engine in a heterogeneous environment, adopting the SBH algorithm leads to faster execution of the engine's matching process resulting in less activation times. This improves the responsiveness of the RBS as a whole while still having lower space requirements than in unscoped RBS. We therefore find that SBH offers significant efficiency benefits for heterogeneous rule engines over both traditional and purely scoped approaches.

7 Related Work

Rule engines based on the Rete algorithm optionally employ a variety of hashing techniques for faster execution. The approach of beta node indexing in [16] describes creating node indexes for beta nodes to be used in beta memories to improve engine performance. In [15] the double-hash method that creates hash maps for various attribute constraints or types improves the speed of filtering facts within the alpha network. These and other similar techniques [14] are orthogonal to the approach we present here and their approaches can thus be implemented together with with SBH. The SBH algorithm presented delegates to the normal execution of the rule engine once a scope check passes, therefore the basic semantics of the rule engine execution is preserved.

There exists research that introduce multitenancy to conventional DBMSes since they do not offer mechanisms to support extensibility and data sharing required in the heterogeneous multitenant context. Work in the multitenacy domain has mapped multiple single logical database schemas to one multitenant physical database schema with shared tables [8, 11]. Most of these approaches utilise structures such as pivot tables that index and map logical multitenant schemas onto physical ones. Additionally, advanced techniques for the multitenant setup such as Chunk Folding [4] and XOR Delta [5] also exist. All these approaches only focus on statically optimising data schemas of largely persistent or static data sets of multiple tenants and do not employ advanced techniques for efficient reactive incremental processing at runtime.

8 Conclusions and Future Work

Modern rule engines are increasingly deployed to support heterogeneous multitenant setups and other similar multiuser environments. We have described the scope-based hashing algorithm SBH, which is a novel optimisation to the popular Rete algorithm in forward-chaining rule engines. SBH extends the Rete network within a scoped engine with scoped hash tables in the alpha memories that are used to efficiently optimise the expressions that compute the compatibility of inputs of a join node in a heterogeneous RBS. From the evaluation we conclude that SBH offers a significant improvement in efficiency during the matching process for heterogeneous data in the rule engine. The cost that comes with this is a relatively higher memory consumption compared to scoped engines without SBH. As future work we would like to extend the SBH algorithm to right activations of beta nodes during the matching process. This intrinsically implies that the beta memories should be hashed and brings about research questions about the semantics of hashing token compositions.

References

1. Aït-Kaci, H., Boyer, R., Lincoln, P., Nasr, R.: Efficient implementation of lattice operations. ACM Trans. Program. Lang. Syst. 11(1), 115–146 (Jan 1989), http://doi.acm.org/10.1145/59287.59293

2. Amazon Web Services, Inc: Rules for AWS IoT. http://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html (Apr 2015), (Accessed on 12/10/2016)
3. Anderson, J.R.: The architecture of cognition. Psychology Press (2013)
4. Aulbach, S., Grust, T., Jacobs, D., Kemper, A., Rittinger, J.: Multi-tenant databases for software as a service: schema-mapping techniques. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 1195–1206. ACM (2008)
5. Aulbach, S., Seibold, M., Jacobs, D., Kemper, A.: Extensibility and data sharing in evolving multi-tenant databases. In: 2011 IEEE 27th International Conference on Data Engineering. pp. 99–110. IEEE (2011)
6. Dettori, P., Frank, D., Seelam, S.R., Feillet, P.: Blueprint for business middleware as a managed cloud service. In: Cloud Engineering (IC2E), 2014 IEEE International Conference on. pp. 261–270. IEEE (2014)
7. Doorenbos, R.B.: Production matching for large learning systems. Ph.D. thesis, University of Southern California (1995)
8. Fiaidhi, J., Bojanova, I., Zhang, J., Zhang, L.J.: Enforcing multitenancy for cloud computing environments. IT Professional 14(1), 16–18 (Jan 2012)
9. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial intelligence 19(1), 17–37 (1982)
10. Giurca, A., Pascalau, E.: JSON rules. Proceedings of the Proceedings of 4th Knowledge Engineering and Software Engineering, KESE 425, 7–18 (2008)
11. Grund, M., Schapranow, M., Krueger, J., Schaffner, J., Bog, A.: Shared table access pattern analysis for multi-tenant applications. In: Advanced Management of Information for Globalized Enterprises, 2008. AMIGE 2008. IEEE Symposium on. pp. 1–5 (Sept 2008)
12. Kambona, K., Thierry, R., De Meuter, W.: Reentrancy and scoping in multitenant inference engines. In: 13th International Conference on Web Information Systems and Technologies (WEBIST) (2017)
13. Nalepa, G.J.: Architecture of the HeaRT hybrid rule engine. In: International Conference on Artificial Intelligence and Soft Computing. pp. 598–605. Springer (2010)
14. Scales, D.J.: Efficient matching algorithms for the SOAR/OPS5 production system. Tech. rep., DTIC Document (1986)
15. Tianyang, D., Jing, F., ZHANG, L.: An improved rete algorithm based on double hash filter and node indexing for distributed rule engine. Transactions on Information and Systems 96(12), 2635–2644 (2013)
16. Xiao, D., Zhong, X.: Improving rete algorithm to enhance performance of rule engine systems. In: Computer Design and Applications (ICCDA), 2010 International Conference on. vol. 3, pp. V3–572. IEEE (2010)