Reentrancy and Scoping for Multitenant Rule Engines

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

Published in:
WEBIST 2017 - Proceedings of the 13th International Conference on Web Information Systems and Technologies

DOI:
10.5220/0006283400590070

Publication date:
2017

License:
Unspecified

Document Version:
Submitted manuscript

Reentrancy and Scoping for Multitenant Inference Engines

[Research Paper]

Kennedy Kambona, Thierry Renaux & Wolfgang De Meuter
Vrije Universiteit Brussel, Pleinlaan 2
Brussels, Belgium
{kkambona, trenaux, wdmeuter}@vub.ac.be

ABSTRACT

Evolutions in web technologies have enabled more and more applications to be deployed in the Cloud leading to the increasing adoption of multitenancy. Multitenant systems can share one application instance across many tenants and clients distributed over multiple devices. These systems need to manage the shared knowledge base reused by the various users and applications they support. Rather than hard-coding all the shared knowledge and ontologies, developers often encode this knowledge in the form of rules to program server-side business logic. In such situations, a modern inference engine can be used to accommodate the knowledge for tenants of a multitenant web system.

Existing inference engines, however, were not conceptually designed to support and to cope with the knowledge of the rules of multiple applications and clients at the same time. They are not fit for multitenant web systems since one has to manually hard-code the modularity of the knowledge for the various applications and clients, which quickly becomes complex and fallible.

We present Serena, a middleware supporting multitenant reactive web applications. Serena augments an event-driven web server with a Rete-based inference engine. The distinctive feature of Serena is the notion of reentrancy and scoping in its inference engine, which is the key solution in making it multitenant. We validate our work through a simulated case study and a comparison with a similar common-place approach, showing that our flexible approach improves computational efficiency in the engine.

CCS Concepts

•**Information systems** → *Language models;* •**Security and privacy** → *Software security engineering;*

Keywords

Inference engines; scoping; reentrancy

1. INTRODUCTION

Middleware 2016

DOI: TODO

Traditionally, software systems were conceptually designed to run in isolation. With cheaper networking hardware and the subsequent rise of the internet, various web technologies have evolved to support dynamic, data-driven and *reactive* applications supporting a massive number of users and client devices. Consequently, modern software systems are increasingly being deployed in the Cloud.

A distinguishing characteristic of Cloud-based platforms is Utility Computing [4] with the *pay-as-you-go* model that improves cost reduction through resource sharing. Utility Computing provides a way in which modern software systems can simultaneously support multiple clients and at the same time share resources through *multitenancy.* A multitenant application is installed on a single instance and serves all clients, or 'tenants' from that instance [24]. They exhibit some advantages, including reduced maintenance and increased scalability as they pertain to economies of scale. However, a large number of providers have limited support for multitenancy at the application level – only focusing on process isolation. Partitioning and securing multitenant application behaviour at this level (also known as native multitenancy) is complex and requires a huge development effort [16].

In this paper, we focus on knowledge-intensive multitenant applications running over the web, connected to different clients sending massive amounts of events and data. These systems are required to manage the shared knowledge base reused by the various tenant applications they support. In order to reason about the data and extract higher-level knowledge it is vital that the value of the sent data be extracted efficiently, its massive and intermittent nature notwithstanding. Rather than hard-coding all the shared knowledge and ontologies, developers often encode this knowledge in the form of rules to program server-side logic e.g. as business rules [17].

In such situations, a modern inference engine can be used to accommodate the knowledge for tenants of a multitenant web system. To this end, we have augmented an event-driven web server with a traditional inference engine constituting Serena, a rule-based multitenant middleware. This enables us to realise the computationally-intensive process of receiving and reactively processing data in order to detect complex events, together with accompanying data relevant to notify clients. In Serena clients can install logic reactive rules that define the complex events they are interested in and dynamically upload data. The rules specify which data to match, who to notify and what information is sent with the notifi-

cation.

Conventionally, inference engines were not conceptually designed to work in the multitenant environment. These rule-based systems (such as production systems [22]) are intrinsically non-reentrant: they are characterised by a flat design space where activations could be observed from all asserted facts without discriminating their sources. Further distinctions between clients and their data sources need to be hard-coded within the rules, which quickly become complex and fallible as the number of clients and the relationships between them increase, or when the relationships become complex to enforce using rule semantics. In a multitenant application, failure to properly make these distinctions can cause unintended rule activations in other clients. Inference engines therefore require orchestration within rules to discriminate or distinguish between instances of different entities. The Serena middleware provides techniques for users and developers to specify *scoped rules* that detect patterns in real-time data and to realise grouping structures in knowledge-intensive multitenant applications.

Scoped rules enable rule creators to distinguish between events pertaining to different clients, while keeping this logic cleanly separated from the application logic. As such, the basic purpose of the rule is not muddied with the logic required for distinguishing clients. This leaves the logical intent of a rule easy to understand for a rule creator. At the same time, scoping enables us exploit a number of performance optimizations in the server's inference engine during the matching process. Our approach of encoding the physical, structural or other logical organizations of multitenant applications eases the computational workload of the inference algorithm, thereby decreasing the engine's overall response time.

In brief, the main contributions in this work are:

- A reactive, rule-based middleware for multitenant architectures supporting knowledge-based applications (Section 3)
- An meta-extension to the Rete algorithm for inference engines to improve reentrancy by incorporating techniques from bit-vector encoding that discriminate data matches as defined in the rules (Section 4.2)
- An extension to the rule-based syntax in the middleware to support scope-based reasoning in multitenant systems (Section 4.3)

We begin by introducing the motivation and proceed to enumerate some requirements in Section 2. We then present Serena middleware and its architecture in Section 3. Next we discuss the scoping mechanism in Section 4. We penultimately evaluate our approach in Section 5 and finally discuss the related work in Section 6 and the conclusion and future work in Section 7.

2. DATA-DRIVEN MULTITENANCY

In this section we motivate the need for a data-driven solution in a multitenant inference engine. To highlight the requirements that such a system should meet, we present a scenario of a service provider in the Cloud for monitoring security systems, similar to a Cloud Access Security Broker (CASB) [10]. In this case, the provider is a service that monitors and logs requests in a university-wide security access system.

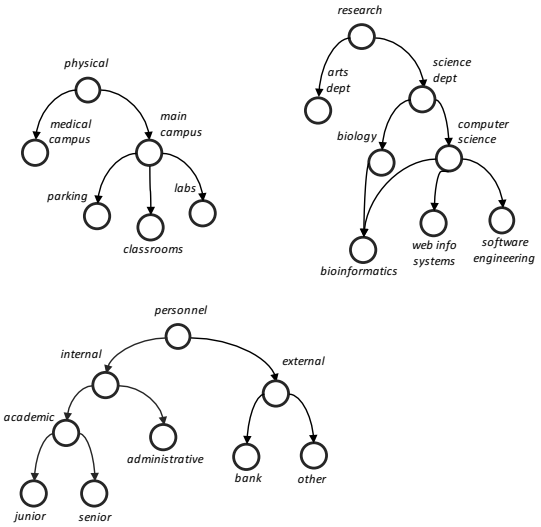


Figure 1: Example structures in a university – Clients are grouped by three hierarchical structures: one based on physical location, one on department, and one on type of personnel. The hierarchies can be arbitrary DAGs and groups can have multiple parents.

2.1 Motivating Example: University Services Access Control

Universities in Brussels have passed a resolution that requires monitoring accesses of students and staff all over their campuses and report access requests that deviate from policies in place. The universities have installed proximity ID-card scanners at most major access points, and students/staff scan their issued ID cards to gain access to various locations in the campuses. Some of the security monitoring policies that the security team design are illustrated below:

1. All students at all levels have access to classrooms during class times on weekdays
2. Only registered student and staff cars are allowed entry to underground parking on their campuses
3. External staff are allowed access only if they have a preauthorised access code, issued by higher level administrative staff

A common university structure consists of different students and staff: research, administrative or external/outsourced; physical structures' hierarchy; and research department hierarchies. A simplified structure for a university is shown in Figure 1. As a result, specific departments and units are allowed to define custom access policies:

4. Biology department students are allowed access to all labs in the (sub)departments in the weekends if accompanied by senior academic staff
5. Only campus bank employees and consultants have access to the bank back office during working hours

For this scenario, we have enumerated around 40 security access policies. The final model of the contains 3 universities and 61 faculty, administrative and physical groupings –

with students, staff and devices belonging to one or multiple groups. Whenever an access request is made by a student or staff the security system of the university sends the data to the monitoring service. According to the policies defined, the service logs the request, computes whether the access is within the defined security policies and displays the results in a dashboard. For instance in policy 1, when a student on a university accesses a classroom during class times the monitoring dashboard would show a status to indicate whether the access is acceptable or otherwise.

2.2 Requirements

The security monitoring service is an representative example of a reactive multitenant application. We particularly target the dynamic design of knowledge-intensive, data-driven applications that continuously stream data back and forth between clients and the server. Accordingly, we exclude multitenant service compositions and static workflow systems as most approaches are process-driven and deviate from these targets. The scenario illustrates some of the requirements that such multitenant middleware should satisfy:

- *Data-driven middleware for instantaneous processing of intermittent data streams* – The middleware should be responsive to new inputs sent by tenants by processing them in real-time or near real-time fashion allowing the end-user application to react to the data. For instance, in the motivating example, the monitoring service provider should be able to process access requests from a large number of clients and devices promptly according to custom policies to provide immediate feedback.

- *Runtime support for the definition and real-time detection of customisable constraints* – The middeware should reduce the complexity of writing code that can efficiently detect real-time events from a continuous stream given a large number of criteria or constraints. This is a challenge to system developers because the intent of the developer is transcended by the accidental complexity [5] of the implementation. In the example, the university security should be able to easily express and upload their own current and future policy constraints for detection of access violations using an expressive syntax.

- *Metadata architecture for multitenant partitioning* – The middleware should be able to model the structures of tenants and possible compositions or relationships between them dynamically through metadata definitions that will discriminate or partition the data residing in the multitenant system. This implies that the internal structures of tenants should be reflected in the runtime in order for it to process the requests within the confines of each client's configuration: in this case the policies of each university. In addition, the internal model should be able to support other software applications from other tenants e.g., other institutions or businesses.

- *Support for concurrent, push-based communications with tenants* – Modern multitenant applications need to be able to handle a number of persistent client connections to reactively send data and to receive feedback. For web developers manually configuring and integrating communication between the server and the clients takes considerable effort through a multitude of standalone, highly specialised connectivity libraries as opposed to having a single comprehensive connectivity framework that internally handles issues such as connection setup and maintenance, session maintenance, initialising event queues, etc. The ultimate goal in this case is to let the middleware and not the application developer fight and babysit the boilerplate processes.

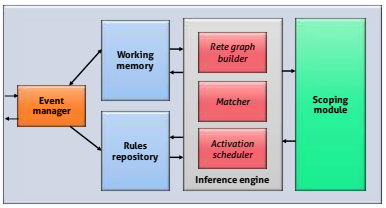


Figure 2: Architecture of the Serena server middleware – The event manager receives events and sends notifications from the inference engine

3. SUPPORTING DATA-DRIVEN MULTITENANCY WITH SERENA

We present the Serena middleware which 1) eases the dynamic definition of requirements by utilizing a rule-based approach, 2) efficiently processes intermittent data giving instantaneous feedback by incorporating a forward-chaining inference engine, and 3) flexibly supports multitenancy by adopting concepts from group theory to model tenant structures. Serena also manages sending messages between the server and its tenants by abstracting the underlying infrastructure that supports push-based communication. We dissect the inner workings of the Serena middleware by first illustrating its architecture and we later explain its execution semantics.

3.1 Serena Architecture

The architecture of Serena is illustrated in Figure 2. The middleware connects clients to the server by means of bidirectional websockets. The middleware server is written as a Node.js package that consists of five main components:

The Working Memory contains the data store that collects and maintains fact assertions from activations and events.

The Inference Engine is the heart of the Serena middleware, and resides on the multitenant server. It serves as the interpreter for the rules, evaluating received data according to the rules. It is composed of 3 submodules:

The Rete graph builder receives the rules from the rules repository, parses the rules, and builds the data flow graph based on a modified Rete algorithm [13] augmented with scopes.

The matching component is tasked with finding consistent bindings in the working memory. Facts can be matched when the conditions of the rules are satisfied. Scope constraints are checked using an encoded representation of the structure of the tenants. The matching component builds an instantiation or activation for every set of facts that satisfy a rule and places them in the queue of the activation scheduler.

The activation scheduler takes the set of all rule instantiations and executes or *fires* them given an activation strategy.

The Rules Repository is tasked with the responsibility of storing rules, and therefore manages the addition and removal of rules from the tenants. It is also involved in building the Rete graph, and in determining which rules to activation within the inference engine.

The Scoping Module internally represents the structure of the tenants, and builds an efficient encoding mechanism for scopes. This encoding affects the matching strategy in the inference engine, and is used by the event manager to determine the recipients of notifications.

The Event Manager receives and queues event data from tenants and pushes queued notifications to the correct recipients whenever a particular rule has been instantiated by the activation scheduler. It also maintains the connection sessions between the tenants and the client library.

On the client side Serena provides a library that initialises and maintains the (re)connections to the server middleware. It further manages sending of messages and reception of notifications pushed from the server through the Event Manager.

3.2 Serena Execution Runtime

The Serena middleware runtime is based on one of the most the widely-used models of knowledge representation known as the production systems model [22]. The distinguishing feature of production systems is the use of data-sensitive rules rather than sequenced instructions as the basis of computation.

Rule-based systems usually consist of a number of unordered rules referencing a global working memory. Similarly native multitenant architectures serve multiple clients that share a dedicated instance, accessing global resources. To support and cope with the knowledge of rules applicable to multiple clients and applications, inference engines and multitenant architectures require features for structural decomposition at the application level. Both models can benefit from modular design and structural abstractions as the systems they support grow in size and complexity.

We outline how the Serena middleware embraces this approach, exemplified using the example scenario. We first begin by explaining the semantics of rules in Serena.

3.2.1 Rule-based syntax

The university policies from the scenario in Section 2.1 can be easily expressed in a rule-based format. We illustrate such a rule to be added by a university security staff using a customised Javascript JSON Rules [14] syntax in Listing 1 for the classroom policy 1. The rule object can be generated from a web-based graphical UI for intuitive rule definitions.

Listing 1: Rule for classtime access

```
1 {rulename: "classtime-access",
2  conditions:[
3   {$s: {type:"student", name: "?name"}},
4   {$d: {type:"accessdevice", name: "?dev",
      ↪ location:"classroom"}},
5   {type:"accessreq", id: "?reqid", person: "?name",
      ↪ time: "?t", device: "?dev"},
6   {type:"$test", expr:"(hourBetween(?t, 8, 20) &&
      ↪ (isWeekday(?t) == true) )"}
7  ],
8  actions:[
9   {assert: {type: "accessrep", reqid:"?reqid",
      ↪ allowed: true}}
10 ]
11 }
```

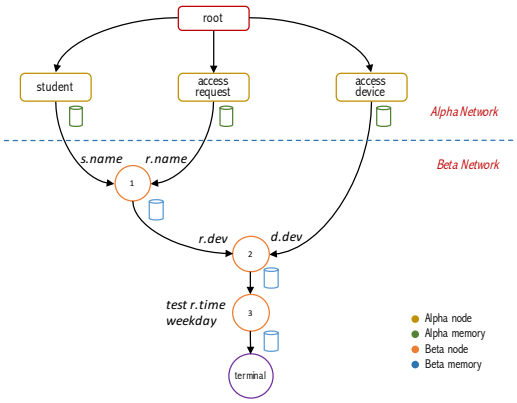


Figure 3: The Rete graph for classtime access rule – Once a token reaches the terminal node the rule is activated.

A rule consists of a name, the left-hand side (LHS) with conditions for event detection, and a right-hand side (RHS) for a reaction after detection. The LHS of the definition (lines 2-6) captures the access request from a person on a proximity device within the specified time periods (line 6). In the rule the '?' operator denotes a variable binding (e.g. ?*nam* in lines 3 & 5). When all the conditions specified in the LHS are satisfied, then the actions defined in the RHS are activated. Here, we assert that the access request has been granted (line 9).

In Serena clients can dynamically add rules to the multitenant server through the middleware's client library. The rules are appended to the existing inference engine's graph and define the real-time detection constraints for that client. In general, the inference engine will process and detect any events that clients are interested in and once activated will notify the relevant client(s). As we show in Section 4.5, a client provides a handler that will be invoked once the rule has been activated.

Such rules are then added into the inference engine. Inference engines perform pattern-matching, a technique that reasons over the data to detect constraints that need to be fulfilled. Most current inference engines are based on the Rete algorithm [13]. Rete compiles rules (such as the one in Listing 1) into a data-flow graph that filters facts (data) as they propagate through nodes performing the actual matching process through the *match-execute cycle*. The matching process searches for consistent bindings between the facts and the existing rules. Efficient matching is achieved through exploiting 1) structural similarity – sharing of the nodes when building the graph, and 2) temporal redundancy – caching of intermediate matched data *tokens* between cycles of incoming results, at the price of higher memory usage.

3.2.2 The Rete Algorithm

In Figure 3 we show the Rete graph built in Serena after addition of the `classtime-access` rule from Listing 1. Facts enter the graph from the *root* node. In the upper alpha

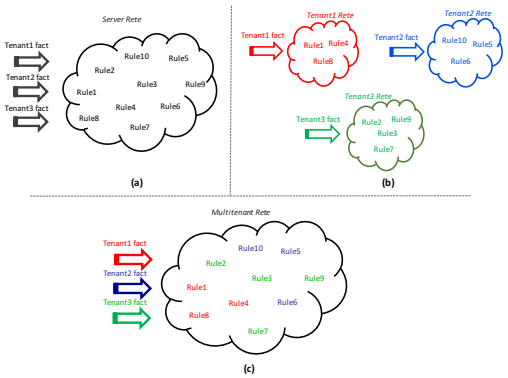


Figure 4: Conceptualizing a multitenant inference engine showing (a) naïve, (b) module-based and (c) scoped engine approaches

network, single-input *alpha nodes* perform generated type selection and intra-condition tests with an *alpha memory* node holding the results. The leftmost alpha node `student` filters facts of that type and stores them in its alpha memory.

The beta network is built in the lexical order of the condition elements forming a left-associative binary tree. Two-input *beta nodes* perform inter-condition tests or join operations on their left and right inputs according to the corresponding conditions. A *beta memory* is associated with each beta node and holds the intermediate join results. The leftmost beta node in Figure 3 performs joins for a `student`'s name and the name of the person performing the `access request`, creates a *token* of both facts in the result and sends it to the next node. It also serves as left input for successive nodes in the beta network. The second beta node receives the token and performs joins of facts from a proximity `device` with the device of the `accessrequest`. For any beta node the right input is always an alpha memory node.

The final beta node in a condition sequence represents the full activation of a rule and is named a *terminal node*. In this case the rule `classtime-access` will be instantiated once a token reaches this node.

The Need for Reentrancy.

In Rete rules are technically shared in their entirety within the network. Structural similarity promotes sharing of nodes performing the same test but corresponding to different rules.

We stated in the previous section that clients can add rules dynamically using the Serena middleware. Adding rules in a multitenant setting is not, however, without its risks when using the naïve approach of having a single inference engine on the multitenant server (Figure 4 (a)) for all tenants. For example, a separate client in another university can develop a rule similar to the one in Listing 1. This will reuse the same graph and as a result both universities receiving notifications of access granted in their dashboards whenever a student in either university enters a classroom: an undesirable result.

The common solution provided by most other engines (as discussed in Section 6) is to spawn a separate engine instance or *module* for each tenant or client (Figure 4(b)). This resolves the problems of unintended and spurious activations, but at the cost of undermining the strengths of the Rete algorithm. By defying the two main tenets of Rete – structural similarity and temporal redundancy – match time and memory usage rapidly increase.

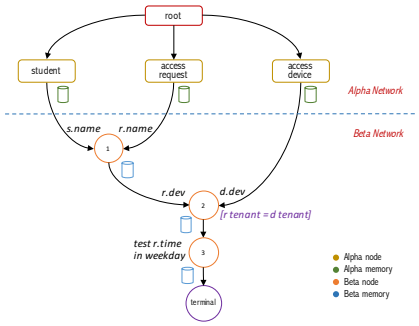


Figure 5: Rete graph for policy 1 – Beta node 2 has an additional tenant check

A more involving approach would require that every rule from tenants have additional discriminatory conditions. This however increases rule complexity, becomes tedious, and hides the rule's logical intent by trying to manually enforce discrimination within the rules of all current and future tenants. It also impacts the resulting Rete network: additional join nodes are created, as we show in the next subsection.

Support for multiple tenants can be improved by making the Rete algorithm *reentrant* such that it can purely handle multiple inference states simultaneously for different sets of client rules, as illustrated in Figure 4(c).

Reentrant Rete.

In order to enforce multitenancy during the inference engine's match cycles, it imperative that an efficient representation be used to represent the hierarchy and to quickly determine the relationships between the data being processed at runtime. To elaborate using Figure 3, an access request that is asserted reaches the second join node causing a left activation. This will initiate a scan on the entire alpha memory for in its right input, to find compatible access devices for a request. Furthermore, in a multitenant setup the scan performs an additional "*is the token's request that we want to match with the access device originating from the same university?*"-check from the same alpha memory of the beta node 2 (depicted in Figure 5 as the join test in brackets – usually this check will result in a separate join node, which we omit here for the sake of clarity). This check tries to find a consistent binding for a device of the same exact university to avoid unintended activations with data from other tenants. The same check is also performed when facts from a different tenant are asserted into the monitoring service.

The numbers of these checks increase markedly when the multitenant inference engine supports relationships within and between tenants. For instance, policy 4 from the motivating example specified that students from a department in the university can have special access times to their departmental labs. The resulting Rete graph for the policy is shown in Figure 6, which results in more join tests and join nodes.

As identified in [21], a major bottleneck in Rete and a number of its variants is such combinatorial join tests. Therefore the group representation needs to be able to perform these checks in a zealously efficient manner. We next show the direction that Serena takes to solve this.

4. SCOPING THE INFERENCE ENGINE

Serena's approach is to embrace the concepts of physical or

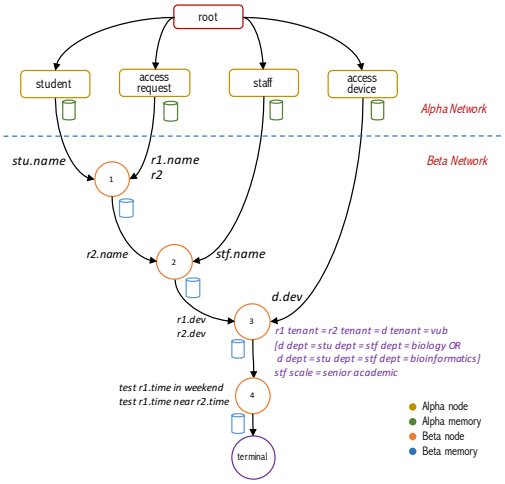


Figure 6: Policy 4 Rete graph with departmental checks – To discriminate data from different clients additional tests are needed.

logical *groups* of tenant clients and their relationships, common in multitenant applications [15]. Examples of groups include research groups in a university, branches in an organization, hobby categories in forums, area zones when monitoring distributed sensor networks, user lists in Twitter or 'campaigns' in Noisetube [28]. The middleware only requires tenants to send their group hierarchies as a list of pairs and it converts and encodes the hierarchies into an efficient representation.

4.1 Group Representation of Tenant Structures

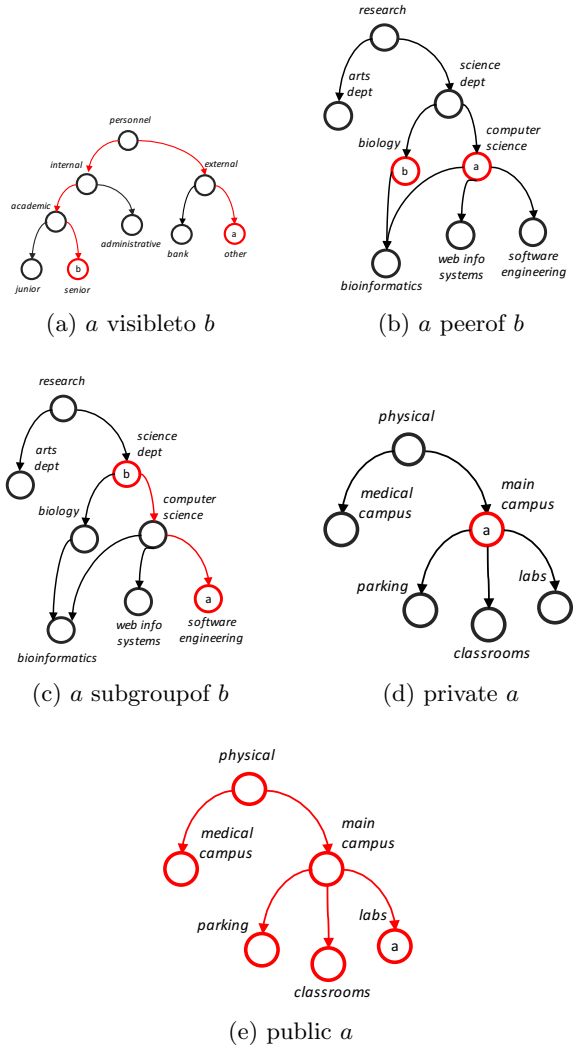
Serena models groups internally with the aim of using these representations to enforce data discrimination in the inference engine. Serena rules can therefore be applicable to a single tenant, to multiple (individual) tenants or to a group of tenants.

We describe a structural representation that uses the notion of a group as a primitive. We showed in Figure 1 how we can conceptually structure the tenants and subtenants into of the monitoring service in groups and subgroups. Serena represents the *group hierarchy* as an acyclic graph with the groups as the nodes with the clients connected to different groups at different levels in the graph.

One characteristic is that groups usually have an aspect of relationships between them – research groups can belong to (sub)departments, hobbies can be categorised into hierarchies of interest groups and sensor area zones can be contained in levels of administrative units. We therefore appropriate the term *scopes* to represent the common relationships between groups in the hierarchy and designate that as a *scope hierarchy*.

Serena adds scopes as (a series of) edges in the group hierarchy. Serena supports the following scope operations shown in Figure 7 on the tenant groups.

- ***visibleto***: In this scope we only capture data from clients in groups that share at least one parent with the specified group. An example is capturing the data that pertains to *senior* academic researchers collaborating with *other* personnel within the same university (Figure 7a). Another example is capturing data from the same university as the specific *bioinformatics* department, excluding data from other universities.



(a) a visibleto b (b) a peerof b

(c) a subgroupof b (d) private a

(e) public a

Figure 7: Scopes supported in Serena – The scopes shown are in relation to the group hierarchy from Figure 1

- ***peerof***: Only data items that originate from peers will be considered in this scope. The peers include groups that are at the same level in the hierarchy. For instance a researcher would want to create a rule with this scope that applies to members in *computer science* and *biology* departments. The *peerof* scope is depicted in Figure 7b.

- ***subgroupof***: Only the data items added by the group or any of its subgroups are included in the scope. This scope is ideal for a departmental rule for *computer science* that will only apply to members of that department or subdepartments (*web info systems* and *software engineering*). See Figure 7c. Its dual is ***supergroupof***.

- ***private***: The private scope will exclusively source data from the specified group and none else (not even its subgroups or parent group). This scope is well suited for data that applies to an exact group, like in Figure 7d where we

can target ID scanning devices that are specifically at the campus entrances and not those in its subgroups such as the campus parking.

- ***public***: Here we capture all data from all defined groups in the hierarchy. The universities could, for example, collaborate in sharing security information between them so they can be interested in data from the devices/student/staff in all the groups and their subgroups. The public scope is depicted in Figure 7e.

4.2 Encoding the Tenant Group Hierarchy

To enforce reentrancy and to efficiently process the various scopes within the match-execute cycle of the inference engine, the Serena middleware internally converts the scopes discussed in Section 4.1 into a more efficient encoding. Our vision is to use an encoding method that rather than performing computationally expensive scope checks such as path traversals in a hierarchical structure, performs (near) constant-time operations to entirely determine data relationships in the structure. This is vital because during the match-execute cycle, Rete performs combinatorial processing in its join nodes as the dataset increases: therefore tenant group path traversals will dramatically affect the performance per cycle. The basic idea is that we precompute the scope check, store and maintain them efficiently as an encoding that will be used to expeditiously process scope constraints.

We base our encoding on the *transitive closure*, a significant component modelling most relationships in knowledge and representation systems as identified in [1] that makes our encoding suitable for querying binary relationships – precisely the kinds of operations that the inference engine performs when performing a scope check between left and right inputs. We next outline the encoding process.

The Group Hierarchy as a Poset – Initially, Serena captures the hierarchy as a *poset*. The example hierarchy in Figure 1 can be represented as a poset $(P, \leqslant)$[1] with the binary relation $\leqslant$ defined as '*is a part of*'[2]. In the poset P have an element (a, b) iff a is part of b, so elements include *(junior, academic)*, *(bioinformatics, biology)*, *(internal, personnel)* and *(computer science, science dept)*. With P we can perform well-defined operations such as bounds (LUB, GLB) and extrema (maximals, minimals)[1]. For instance the maximals in the group hierarchy of Figure 1 are *physical*, *research* and *personnel*. Even so, when processing poset operations the engine would still have to traverse the elements of the poset recursively.

The Groups as a Lattice – We can convert the groups poset to a lattice L as outlined in Appendix A.2. This leads to the hierarchy depicted as the hasse diagram in Figure 8. Other distinct hierarchies can have their own top-level element same as $\top$. A lattice represents the group hierarchy in a form that is more efficient to encode and compute than the earlier poset representation.

Encoding the Lattice – With L, Serena performs a customised *bit-vector encoding* method outlined in Appendix C that is based on the method by Aït-Kaci [2]. The result is a binary matrix encoding M_ϑ of the group hierarchy, shown in Figure 9 for our example, with the following properties:

i) The labels on the rows of M_ϑ represent the groups in L; similarly for columns. The first row represents $\top$ and the last row represents $\bot$.

ii) An entry $M_{\vartheta(a,b)}$ has a `1` if `group` a = `group` b or if `group` a is an ancestor of `group` b in L, and `0` otherwise

iii) An entry $M_{\vartheta(b,a)}$ has a `1` if `group` a = `group` b or if `group` a is an descendant of `group` b, and `0` otherwise

iv) A is a maximal iff the row $M_{\vartheta(a,*)}$ has a `1` only at $M_{\vartheta(a,a)}$ and at $M_{\vartheta(a,\top)}$

v) A is a minimal iff the column $M_{\vartheta(*,a)}$ has a `1` only at $M_{\vartheta(a,a)}$ and at $M_{\vartheta(\bot,a)}$

[1]For definitions, see Appendix A.1

[2]In most cases the general $\leqslant$ relation '*is subgroup of*' suffices

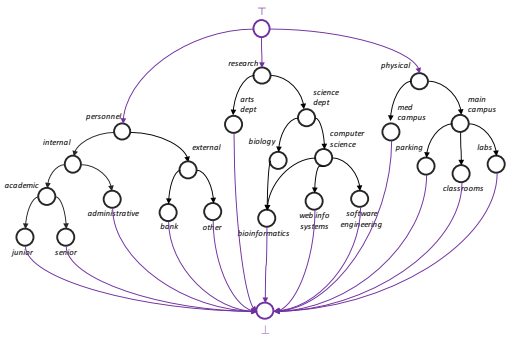


Figure 8: Hasse diagram of the group hierarchy as a lattice – The lattice is the basis of encoding the group hierarchies in a matrix.

	⊤	pers	rese	phys	inte	scie	main	acad	comp	biol	admi	labs	clas	semi	soft	bioi	⊥
⊤	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
pers	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rese	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
phys	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
inte	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
scie	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
main	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
acad	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
comp	1	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
biol	1	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0
admi	1	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
labs	1	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0
clas	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
semi	1	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0	0
soft	1	0	1	0	0	1	0	0	1	0	0	0	0	0	1	0	0
bioi	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	1	0
⊥	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 9: The group hierarchy matrix encoding M_ϑ – The groups are labels in the rows and the columns of the matrix

Additionally, the encoding process generates the *level*[1] of each group, which we store as an integer. We also store indexes for all the maximals in the matrix. We now show how the encoding is used to perform scoping within the inference engine.

Scoping with $\mathbf{M}_\vartheta$ – The encoding with M_ϑ is the basis of performing scope operations in the inference engine. To facilitate this Serena adds scope tests at appropriate nodes when building the Rete network, and performs scoping operations in the beta betwork's beta join nodes during matching.

- ***visibleto*** – To perform a scope check of a `visibleto` b the middeware checks if the result of $M_{\vartheta(a,*)} \wedge M_{\vartheta(b,*)}$ is a maximal in M_ϑ as per property (iv).

- ***peerof*** – To check if a `peerof` b it calculates if $Level(a) = Level(b)$ from the process of encoding M_ϑ.

- ***subgroupof*** – A scope check of a `subgroupof` b is true if the result of $M_{\vartheta(a,*)} \wedge M_{\vartheta(b,*)} = M_{\vartheta(b,*)}$ as per property (ii). Conversely, b is a `supergroupof` a.

- ***private*** – To find out if a `private` b it can check if $M_{\vartheta(a,*)} \wedge M_{\vartheta(b,*)} = M_{\vartheta(a,*)}$ as per property (ii) and (iii).
- ***public*** – For a scope check of a `public` b then middleware calculates if $M_{\vartheta(a,*)} \wedge M_{\vartheta(\top,*)} = M_{\vartheta(\top,*)}$ as per properties (ii) and (i).

With these operations, the middleware can perform scope operations efficiently. It retrieves the values in the matrix and performs binary operations (as defined in Appendix C) from the encoding in near-constant time.

4.3 Defining Scoped Rules

The Serena middleware allows clients to upload rules to the server. To expose scoped rule definitions, Serena follows a similar direction as Allen's work in [3] that proposes rule extensions for temporal interval constraints. Similarly, we present *scope-based constraints* by extending the normal rule syntax with scope-based definitions that specify structural constraints on the groups and the relationships between them, which we simply call *scopes*. The scopes supported are as in Section 4.1.

We illustrate with an example. We show how to define policy 4 of biology students lab accesses in the weekends from Section 2.1 using scope constraints in Listing 2.

Listing 2: Rule for biology dept. weekend lab access

```
1  {rulename: "biology_weekend_access",
2   conditions:[
3    {$stu: {type:"student", name: "?stuname"}},
4    {$stf: {type:"staff", name: "?stfname"}},
5    {$d: {type:"accessdevice", name: "?dev",
       ↪ location:"labs"}},
6    {type:"accessreq", id: "?reqid1", person:
       ↪ "?stuname", time: "?t1", device: "?dev"},
7    {type:"accessreq", id: "?reqid2", person:
       ↪ "?stfname", time: "?t2", device: "?dev"},
8    {type:"$test", expr:"(hourBetween(?t, 8, 20) &&
       ↪ (isWeekend(?t1, ?t2) == true) && isNear(?t1,
       ↪ ?t2) )"}
9   ],
10  scopes:[ "biology supergroupof ($stu & $stf & $dev)",
      ↪ "$stf private senior"],
11  actions:[
12   {assert: {type: "accessrep", reqid:"?reqid1",
       ↪ allowed: true}}
13  ],
14  notify:[ "subgroupof administrative"]
15 }
```

The rule is similar to Listing 1, with an additional `scopes` section (line 10) where the bound condition variables in line 3, 4, and 5 are referenced to check whether the student, staff and device facts are all tagged to belong to the `biology` department or its subdepartments using the scope check `supergroupof`. The `&` operator makes it easier to define multiple scope constraint checks in one line. The additional scope check in line 10 enforces the constraint that the staff member has to be in the `senior academic` group. The rule will therefore detect the constraints of policy 4, which was to capture lab accesses made in the weekends by a student that is accompanied by a senior academic staff member in the biology department and any of its subdepartments.

4.4 Scoped Constraints in the Rete Graph

Within the inference engine, the rule in Listing 2 will be built as shown in Figure 10. The main difference is in the

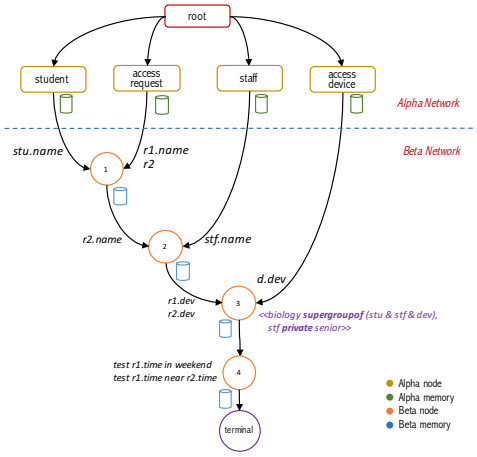


Figure 10: The scoped Rete graph for policy 4 – The expensive tests are replaced with scope tests performed by efficient encoding operations

beta join node 3 where we now have in place a more compact and efficient way to discriminate the tokens that the node is supposed to process. The scoping module will use M_ϑ to perform the binary operations from the scope checks in the figure denoted with angle brackets.

When a token reaches beta node 3, for instance, Serena will perform the `supergroupof` scope check defined in Section 4.2. For example, if the access request is from a student *stu* from the *bioinformatics* subgroup, the engine performs the supergroup check on the token, which in this case succeeds:

$M_{\vartheta(biology,*)} \wedge M_{\vartheta(bioinformatics,*)} = M_{\vartheta(biology,*)}$

```
  1010010001000000
 &1010010011000001
 ----------------
  1010010001000000(biology)
```

Similar operations are performed for this `supergroup` check and the next `private` scope check for the `senior` staff member from the academic personnel group using M_ϑ.

4.5 Scoped Notifications

Eventually, once the security access is granted then the effect should be logged and displayed on the the correct university's dashboard as per the requirements of the example. One issue that arises is *who to notify*, and specifically, which group of which tenant should receive the notification.

With the matrix encoding, Serena can support advanced notification mechanisms whenever a scoped rule has been activated. We refer back to Listing 2 where Serena rules expose a *notify* construct in line 14 that specifies the groups to notify once the rule is fired. The *notification scopes* are similar to the matching scopes but in this case they can enforce notification constraints to a group, subgroup, or direct clients. Furthermore, Serena invokes similar binary operations as in Section 4.2 to determine the groups to notify as when performing a scope check during matching.

To illustrate, the `notify` construct of the rule in Listing 2 will notify members of `administrative` group and its subgroups. The `notify` will invoke code in the clients that provide a callback registered in the `onRuleActivated` construct provided by the Serena middleware client library. With the `subgroupof administrative` scope notification defini-

tion Serena checks the entry $M_{\vartheta(*,administrative)}$ and notifies clients in the groups which have an entry of `1` which in this case is only the clients of the group `administrative`.

We next show client code for receiving notifications using JavaScript syntax in a web application. Initially the client `headofsecurity` connects to the server middleware (providing the name and groups s/he belongs) in line 1 of Listing 3. In line 3 we add a rule and the acknowledgement callback to be invoked after it has been added to the server. The client also specifies code that will be executed whenever a notification for a scoped rule is detected, with the arguments of the `rulename`, the `facts` and the `client` that added rule (line 6). The interface of the client `headofsecurity` can thus be updated with the information from the facts that activated the rule in line 9, which shows the staff, student and access request that has been granted.

Listing 3: Adding a Serena client rule and notification

```
1  var serena = new SerenaClient('server.net',
       ↪ 'headofsecurity', 'administrative');
2  var rule4 = // json rule 'biology_weeked_access' here
3  serena.addRule(rule4, function(err, result){
4    // rule was added
5  });
6  serenaClient.onRuleActivated(
7    function(rulename, facts, source){
8      // rule was fired
9      updateDashBoard(facts);
10 });
```

5. EXPERIMENTAL EVALUATION

In this section we evaluate our approach with the University Services Access Control scenario detailed in Section 2.1. We focus on the investigating whether the scoping metadata architecture within the inference engine as presented has significant computational benefits over a traditional inference engine.

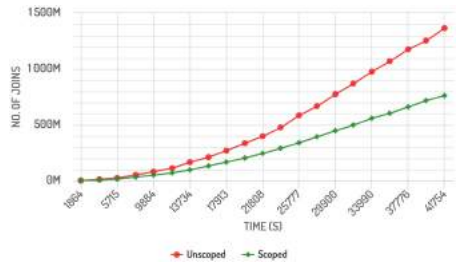
5.1 Setup & Methodology

The example scenario was implemented as a simulation running on an event-driven web server. The final application has a total of 61 groups in hierarchies, 39 access rules, and 73 concurrent clients across 3 sample universities. All clients are connected to the multitenant server concurrently through websocket connections managed by the middleware. The server runs Node.js and has an AMD Opteron Processor 6272 at 2.1Ghz. The maximum RAM allocated to the Node.js simulation was 20GB.

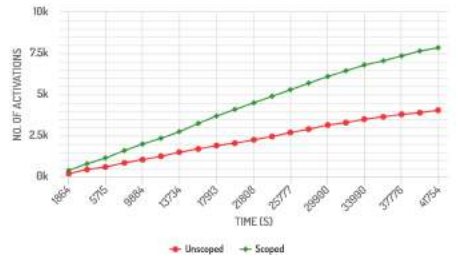
We categorised the general simulation setup into two: one with traditional rules having expression tests to enforce data discrimination (unscoped), and another with Serena's scoped rules using the Serena middleware. Following random assignment experimental design we generated access requests from clients in both categories throttled in ranges of between 1-5 seconds, with a single simulation running for 12 hours. The requests model students and staff from different departments or personnel levels randomly accessing various university locations. We ran 35 iterations of simulations in each category, making a total of approximately 70 simulations and 840 hours runtime. During the simulations we logged the number of joins, RSS/Heap memory used and activations by the server.

5.2 Results & Discussion

Figure 11 shows the cumulative number of join tests performed and rule activations from a single simulation of both scoped and unscoped engines, and in Figure 13 we calculate and chart the sample means of the observed results of all the 70 randomised simulations.



(a) Number of beta join tests computed



(b) Activations

Figure 11: Results in one simulation – We compare the cumulative results of one run of 12 hours

From the graphs in Figure 11 we observe that the traditional unscoped Rete graph built from manually programming data discrimination within the rules suffers a marked increase in the number of joins computed compared to our scoped graph. The unscoped approach spent more time processing the expensive combinatorial join operations in the engine. The scoped engine's scope tests use the encoding, leading to better performance, and consequently to a higher number of activations recorded (by approximately 31%) within the same simulation run.

The aggregated results in Figure 13 show evidence of a better overall performance of the scoped engine. Compared to a traditional approach, Serena on average improves the computation of scope tests and total memory consumption, increasing the average number of rule activations of all randomised simulation runs.

The reduced memory consumption is an interesting result:

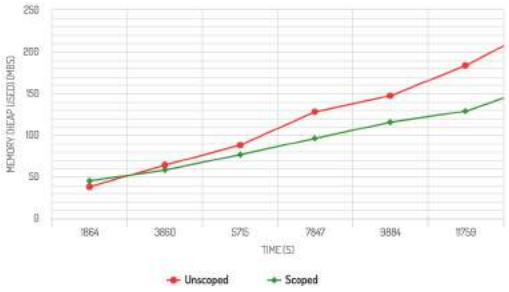
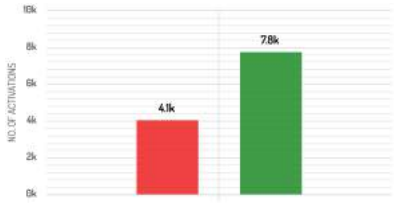
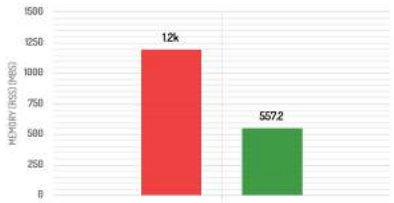


Figure 12: Results of the initial memory consumption – We compare the memory used at the beginning of the run in Fig. 11

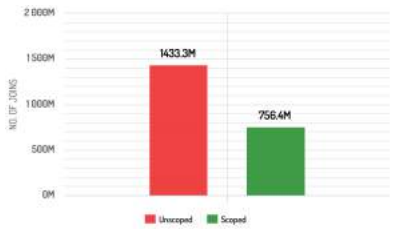
given the matrix encoding of the hierarchy one would expect a higher memory consumption in the scoped approach. This is indeed true when we zoom in on the initial memory consumption of the single run, as illustrated in Figure 12. Eventually, however, the unscoped approach surpasses the scoped engine after about an hour of asserted facts. This is because encoding the structural organisation of the tenants and implementing scoping using metadata enables Serena to perform optimisations in the internal data structures used to tag data from clients. For example: in the traditional approach each fact would contain a dedicated `group` slot object, taking up the same space as other slots like name, age etc. In our approach we utilise the lattice hierarchy labels to tag the data as opposed to creating slots in facts thus saving up space occupied in each asserted fact.



(a) Mean no of activations



(b) Mean memory used



(c) Mean no of beta joins

Figure 13: Aggregated results over all randomised simulations – The results were collected from 70 simulation runs of around 840 hours

From the results we observe that introducing metadata-driven reentrancy in the engine results in simpler rule design and efficient computation in the middleware. This in effect means that the security monitoring system can process a larger number of access requests at a faster rate than the traditional rule engine approach. The requirement is that the tenants need to specify their group structures to fully take advantage of the Serena middleware: the groups to be encoded in the university hierarchy have to be defined and added to the engine so that the benefits provided by the encoding can be fully realised.

6. RELATED WORK

We describe similar rule-based techniques and data-driven methods that support the development of multitenant applications focusing on preventing unnecessary duplication of processes and resources at the application level.

Decomposition in Rule-based Systems.

Modern rule engines such as Drools [25] and Jess [18] are based on the Rete algorithm as well and have their custom enhancements to the algorithm. Some of the engines have support for extensions applicable to web servers, e.g. Drools Guvnor.

Techniques that decompose larger rule bases into groups of rules in other engines all use the concept of *rulebooks* that consist of isolated sets of rules with no relationships between them. Examples are *modules* in Jess and *ruleflows* or *event sources* in Drools. Essentially these give each tenant their own Rete networks, making them fundamentally independent. Serena in contrast facilitates and encodes scoping within a single reentrant Rete network thus fully enforcing structural similarity and temporal redundancy, backed by the same knowledge base.

Schema Sharing in Databases.

A common technique to support multitenancy is by mapping the context of clients into the existing patterns of conventional databases and similar systems. These systems however lack the concept of a 'tenant' and thus do not offer out-of-the-box support for handling the metadata needed for multitenancy [19]. Advanced schema techniques such as Sparse Columns [6], Extension Tables [8] and Multitenant Shared Tables [15] have static and complex configurations and will degrade in performance when implemented in reactive inference engines with incremental processing.

Multitenant Middleware.

Most dedicated middleware for multitenant architectures aim to support multiple tenants at the application level using various techniques. The research in [26] and [11] achieves this through variations of the aforementioned schema sharing techniques. The SaaSAMT approach in [23] supports process-based tenant shareability based on different architectural layers. Most of the research mentioned falters in inherently supporting knowledge-based reasoning systems with incremental processing, and takes little advantage of the heterogenous nature of data in multitenant applications in their approach.

Support for application-level middleware through platforms like the Google App Engine/AppScale [27] use namespaces that partition application data across tenants. Other platforms like GigaSpaces [7] and JSR [9] use similar techniques. The platforms do not intrinsically support the flexibility and expressiveness in customizing tenant behaviour as is offered by the rule-based approach of the Serena middleware. Nevertheless, with some effort they can be utilised as foundations of its runtime.

Distributed Event-based Systems.

Distributed Event-based Systems exchange loosely-coupled data asynchronously between producers and consumers. Consumers subscribe to *topics* or *channels* of interest and receive notifications from producers. Work in [20] and [12] provide custom routing of event notifications from producers to sub-

scribed consumers. Most of existing research, however, focuses on the existence of an overlay of brokers that filter notifications before reaching the respective consumers. In contrast, scoping in Serena is primarily for improving reentrancy in the inference engine during the matching process. Furthermore, Serena provides filtering of rule notifications at the event source to connected clients, which does not require the use of a broker architecture.

7. CONCLUSIONS AND FUTURE WORK

We have described the Serena middleware, a system for reasoning in multitenant architectures with an inference engine based on the Rete algorithm. Our technique is useful in a number of multitenant applications to deal with the problem that much of the heterogeneous knowledge significant when performing reasoning and deductions can be structured hierarchically within a multitenant setup. The technique uses groups and common relationships between them to build an internal representation that captures the scopes present in many multitenant domains by using a hierarchy of groups. The model precisely controls the amount of deductions or computations performed automatically by the middleware as information from tenants flows into the system in a both expressive and computationally effective manner.

As future work we would like to investigate support for dynamic scopes that can be defined by the tenants during execution of the engine, thus affecting the encoding and the state of the inference engine's intermediate memories. In addition, we observe that Rete-based inference engines experience degradation of performance and become lethargic when run for long periods of time. We are looking into ways in which the middleware can support persistence in the engine and its intermediate memories to offload antiquated data.

8. ACKNOWLEDGMENTS

The authors would like to thank colleagues at the Software Languages Lab for their input and inspiration. Thierry Renaux is supported by a doctoral scholarship of the Agency for Innovation by Science and Technology in Flanders (IWT), Belgium.

9. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *SIGMOD Rec.*, 18(2):253–262, June 1989.

[2] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Trans. Program. Lang. Syst.*, 11(1):115–146, Jan. 1989.

[3] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, Nov. 1983.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[5] F. P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, Apr. 1987.

[6] E. Chu, J. Beckmann, and J. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 821–832, New York, NY, USA, 2007. ACM.

[7] U. Cohen. Inside GigaSpaces XAP-Technical overview and value proposition. *New York, NY: GigaSpace Technologies Ltd*, 2004.

[8] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. *SIGMOD Rec.*, 14(4):268–279, May 1985.

[9] L. DeMichiel and B. Shannon. Jsr 342: JavaT M Platform, Enterprise Edition 7 (Java EE 7) Specification, 2011.

[10] E. Fernandez, N. Yoshioka, and H. Washizaki. Cloud access security broker (CASB): A pattern for secure access to cloud services. In *4th Asian Conference on Pattern Languages of Programs, Asian PLoP '15*, Tokyo, Japan, 2015.

[11] J. Fiaidhi, I. Bojanova, J. Zhang, and L. J. Zhang. Enforcing multitenancy for cloud computing environments. *IT Professional*, 14(1):16–18, Jan 2012.

[12] L. Fiege, M. Cilia, G. Muhl, and A. Buchmann. Publish-subscribe grows up: support for management, visibility control, and heterogeneity. *IEEE Internet Computing*, 10(1):48–55, Jan 2006.

[13] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.

[14] A. Giurca and E. Pascalau. JSON rules. *Proceedings of the Proceedings of 4th Knowledge Engineering and Software Engineering, KESE*, 425:7–18.

[15] M. Grund, M. Schapranow, J. Krueger, J. Schaffner, and A. Bog. Shared table access pattern analysis for multi-tenant applications. In *Advanced Management of Information for Globalized Enterprises, 2008. AMIGE 2008. IEEE Symposium on*, pages 1–5, Sept 2008.

[16] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. A framework for native multi-tenancy application development and management. In *The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)*, pages 551–558, July 2007.

[17] D. Hay, K. A. Healy, J. Hall, et al. Defining business rules – What are they really? *Final Report*, 2000.

[18] E. F. Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA, 2003.

[19] D. Jacobs, S. Aulbach, et al. Ruminations on multi-tenant databases. In *BTW*, volume 103, pages 514–521, 2007.

[20] L. Lim and D. Conan. Distributed event-based system with multiscoping for multiscalability. In *Proceedings of the 9th Workshop on Middleware for Next Generation Internet Computing*, MW4NG '14, pages 3:1–3:6, New York, NY, USA, 2014. ACM.

[21] P. Nayak, A. Gupta, and P. S. Rosenbloom. The Soar Papers. chapter Comparison of the RETE and TREAT Production Matchers for Soar (a Summary), pages 621–626. MIT Press, Cambridge, MA, USA, 1993.

[22] A. Newell. Production systems: Models of control structures. Technical report, DTIC Document, 1973.

[23] S. Pal, A. K. Mandal, and A. Sarkar. Application multi-tenancy for software as a service. *SIGSOFT Softw. Eng. Notes*, 40(2):1–8, Apr. 2015.

[24] M. Pathirage, S. Perera, I. Kumara, and S. Weerawarana. A multi-tenant architecture for business process executions. In *2011 IEEE International Conference on Web Services (ICWS)*, pages 121–128, July 2011.

[25] M. Proctor. Drools: A rule engine for complex event processing. In *Proceedings of the 4th International Conference on Applications of Graph Transformations with Industrial Relevance*, AGTIVE'11, Berlin, Heidelberg, 2012. Springer-Verlag.

[26] H. Yaish, M. Goyal, and G. Feuerlicht. An elastic multi-tenant database schema for software as a service. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 737–743, Dec 2011.

[27] A. Zahariev. The Google App Engine. *Helsinki University of Technology*, 2009.

[28] J. Zaman, E. D'Hondt, E. Gonzalez Boix, E. Philips, K. Kambona, and W. De Meuter. Citizen-friendly participatory campaign support. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on*, pages 232–235. IEEE, 2014.

APPENDIX

A. DEFINITIONS

A.1 Posets

A poset $(P, \leqslant)$ is a set P and a binary relation $\leqslant$, such that for all $a, b, c \in P$, the following properties always hold:

1. $a \leqslant a$ (reflexivity)
2. $a \leqslant b$ and $b \leqslant c$ implies $a \leqslant c$ (transitivity)
3. $a \leqslant b$ and $b \leqslant a$ implies $a = b$ (antisymmetry)

A.1.1 Poset Operations

Bounds: Given $A \subseteq P$, an element $b \in P$ is called an *upper bound* of A if $a \leqslant b$ for all $a \in A$. b is a *least upper bound* or LUB if $b \leqslant a$ whenever a is an upper bound of A. The dual of the least upper bound is known as the *greatest lower bound* or GLB[3].

Extrema: The *maximal* of a poset P, abbreviated $\lceil P \rceil$, is an element $m \in P$ that is not greater than any other element in P according to $\leqslant$. More formally,

$$\lceil P \rceil = \forall b \in P, b \leqslant m \tag{1}$$

If there is one unique maximal element in P, we call it the *maximum*. The dual of the maximal is known as the *minimal*, $\lfloor P \rfloor$ and a unique minimal is known as the *minimum*.

A.2 Lattices

If in a poset P every pair has at least an LUB $\wedge$ and a GLB $\vee$, then the poset P becomes $(P, \leqslant, \wedge, \vee)$, a *lattice* L. One way to transform the poset P in into a lattice is by adding a parent $\top$ to every maximal and a child $\bot$ to every minimal in P.

[3]The LUB $\vee$ of P is also known as the *join* or *suprema* of A. The GLB $\wedge$ is the *meet* or *infima* of A.

A.2.1 The Covering Relation

We say for two elements $a, b \in P$, a is *covered* by b if b immediately follows a in the poset ordering (i.e. a is an immediate successor of b). More formally,

$$a \prec b \text{ iff } a \leqslant b \text{ and } \nexists c \text{ s.t. } a \leqslant c \leqslant b, c \neq a, c \neq b \tag{2}$$

This enables us to depict a lattice in a *hasse diagram*, where a curve goes from b to a iff $a \prec b$.

A.2.2 Lattice levels

In this paper we define the level of an element a in a lattice is the longest distance of a from the maximum of the lattice (in this case, $\top$) to the element, i.e.,

$$Lvl(a) = \begin{cases} 0 & \text{when } a \text{ has no predecessors in } P \text{ and,} \\ max(\{Lvl(b)) \mid b \succ a\}) + 1 & \text{otherwise.} \end{cases} \tag{3}$$

where $\succ$ is the dual of $\prec$.

B. OPERATIONS WITH ϑ

Having L we can define a mapping ϑ from L to another lattice $(S \subseteq, \cap, \cup)$ that for every $a, b \in L$,

$$\vartheta(a \wedge b) = \vartheta(a) \cap \vartheta(b), \tag{4}$$

$$\vartheta(a \vee b) = \vartheta(a) \cup \vartheta(b). \tag{5}$$

If ϑ is invertible, then this makes it easy to calculate $\vee$ and $\wedge$. $\forall\, a, b \in L$,

$$a \wedge b = \vartheta^{-1}(\vartheta(a) \cap \vartheta(b)), \tag{6}$$

$$a \vee b = \vartheta^{-1}(\vartheta(a) \cup \vartheta(b)). \tag{7}$$

C. MATRIX ENCODING

We use the encoding method mentioned in [2] taking ϑ as the transitive closure, with a modification that will enable us to map a lattice L to an encoded matrix M_ϑ.

- Instead of starting with $\bot$, start with $\top$ as the first element. Assign $\vartheta(\top) = 0$.
- Move to the next elements level by level downwards in L, level by level, and calculate the bitcode of each element as a vector.
- The bitcode of an element $a \in L$ is obtained by

$$\vartheta(a) = 2^{i-1} \vee \bigvee_{a \prec x} \vartheta(x) \tag{8}$$

  where i is the number of elements visited since $\top$, and $a \prec x$ are the parents of a.
- An entry in the new matrix M_ϑ for a is the reverse of the bitcode obtained by (8), without the most significant bit.

With this encoding, we can perform operations in Eq (6) and (7) having $\cap$ as the bitwise `AND` and $\cup$ as bitwise `OR` on M_ϑ.