Resource Information Service for Cloud Datacenters

Radko Zhelev
Institute on Parallel Processing
Bulgarian Academy of Sciences
Sofia, Bulgaria
zhelev@acad.bg

Vasil Geogriev
Faculty on Mathematics and Informatics
University of Sofia "St. Cl. Ohridsky"
Sofia, Bulgaria
v.georgiev@fmi.uni-sofia.bg

Abstract—**Existing Grid systems make different choices about the employed type of data storage used for keeping persistent information about the presence and state of the available grid-wide resources. They usually use a concrete storage type – LDAP [2], RDMBS [7], Round-robin database [5], etc. and build the organization of their distributed infrastructure using the same storage type for all resources employed within Grid. Of course each type of storage brings its specific advantages and disadvantages regarding arbitrary considerations. In this paper we propose a design approach for resource data persistency in Cloud datacenters that does not restrict all the data to be kept in a certain type of storage. Instead, different types of resources can choose an optimal storage for their data with respect to the specific resource semantic requirements. Although data is spread into different locations and fetched via different types of queries, we still implement efficient filtering for cross-resource-type searches. We achieve this by employing the techniques of logical decomposition of Boolean expressions and executing partial filters against the responsible parties.**

Keywords - Cloud Systems; Grid Information Services; Database; Search; Filtering; Query Processing

I. RELATED WORK OVERVIEW

A. Grid Information Services

A Grid Information Service is a software component, whether singular or distributed, that maintains information about the resources in distributed computing environment, and makes that information available upon request. Information Services has an Update Interface for populating resource entities data and Query Interface for retrieving it. Updates are performed within obtainment of status information from actual resources, while Query Interface is used by administrator users, grid applications, job schedulers, etc. The sophistication of the Query Interface determines the ease with which users can write queries and the efficiency of query executions. Relational and object oriented databases support a standardized and powerful access method like SQL [3], which is declarative and queries are optimized so are highly efficient. Hierarchical models such as LDAP use simplified access protocols. There are two main sides of the Grid Information Service discussion. In one view, Grid Resource Information is best served by a hierarchical representation, and on the other side, that a relational or at table representation is more suitable [1].

B. Resource data in existing Grid systems

Existing grid middlewares make use of different, but always one certain storage type for keeping information about system resources. Globus Toolkit [11] adopts hierarchically based infrastructure and uses a LDAP storage with LDAP and Xpath queries [4][9] for retrieving resource information. Condor [12] and its Hawkeye [13] are concentrated on gathering of statistical information about the load and utilization of the available hardware resources - disk space, memory usage, open files, system load, etc. The information is stored in the so called Round-robin database [5], which is very suitable for storing statistical information since it has functionality to consolidate (summarize) old statistical data, gradually reducing the resolution of the data along the time axis. The Relational Grid Monitoring Architecture (R-GMA) [10] used within the European Data Grid, as it's names says, implements GMA [8] using a relational RDBMS database for storing and retrieving of resource information via SQL queries.

C. Analysis and Motivation

Each storage type has its pros and cons and different ones could appear as most appropriate for different use cases. Data about system users is usually kept in LDAP database [17]. Over the years LDAP has become the most popular storage for managing users' data and most of the related user management tools work exactly with LDAP servers. Unfortunately LDAP has certain disadvantages regarding the general searches functionality. The Round-robin database tool is unbeaten for keeping statistical and hardware utilization data, but again it is highly insufficient for more general-purpose persistency like application states, software configurations, etc. Meanwhile there are types of resources that may prefer to use unstructured or semi-structured data, thus not needing standard database functionality at all – for instance, system log entries could better be kept in simple files on the file system or a distributed file system (DFS). There could also be resources that may report only runtime status, without any need for persistency - like currently active connections.

Another major issue introduced in the existing Grid systems is that even if one certain storage type is used, every type of resources naturally needs a custom data structure to be efficiently stored. For instance R-GMA defines every resource type to be related to a certain database table with specific name and columns. Since Query Interface is usually a native database query (SQL or LDAP), one should possess intimate knowledge on the specific data structures for every different resource type. The last functional disadvantage is the inability to perform complex queries that relate information from different resource types. Even in R-GMA, although employing relational database, it is not possible to perform table joins if independent Producer tables reside on different physical location [6] [20].

This research is supported by the National Science Fund Proj. ДДВУ02/22-2010

These considerations lead us to the conclusion that each type of resources should be able to use an arbitrary type of storage for meeting efficiently its specific requirements; Query Interface needs to be decoupled from the physical data structures, allowing more opportunities for data structuring as well as more convenience in the Query Interface; and finally we need well-defined capabilities for cross-resource-type filtering even if data is spread in various locations and shapes.

The paper proceeds as follows. In chapter II we make a short overview of the "Generic Resource Framework" defined in our previous paper [15], on which basis we build our Information Service. In III we present the data store architecture and information management organization. In chapter IV are described the searches capabilities provided by our System. Chapters V and VI present an efficient implementation of those searches. In VII we analyze the functional and performance aspects of the proposed design. And in the conclusion, we summarize the achieved goals and outline the directions for future elaboration.

II. GENERIC RESOURCE FRAMEWORK – BRIEF SUMMARY

A. *Resource Representation*

Our resource framework defines a general abstraction for representing diverse types of resources into a uniform interface regardless of the underlying resource access protocol and related information structure. Its main characteristics are:

- A **Resource state** is represented as a set of State Variables, which are simply name-value pairs.
- Every **Resource** has a **type and id** associated with it. The type of Resource identifies its interface, i.e. every Resource of a given type has a constant list of State Variable names. There may be many Resource instances of the same type, all having the same list of State Variable names, but being in different state, i.e. different values for the State Variables.
- **Resources** may be **arranged hierarchically** - every Resource instance may have one or more sub Resources and in turn may have a parent Resource. This allows more complex entities to be decomposed to a hierarchy of sub-components, achieving arbitrary level of granularity. For the purpose of extensibility it is responsibility to the sub Resource to attach itself to its parent.
- In datacenter-wide scope, we can say that Resources are physically or logically hosted on some autonomous datacenter entity that we can call a **Resource Host**. A Resource Host could be a datacenter node hosting different Resource entities – native processes, user applications, system log files, etc. Resource Hosts could also be completely logical units like user accounts or predefined maintenance procedures currently performed in the datacenter. Thus we distinguish two kinds of Resources – Host Resources and Component Resources. Every Component Resource "belongs" to certain Host Resource. Again for the purpose of extensibility it is responsibility to the Component Resource to attach itself to the respective Host.

In summary, we can say that Resources are represented as a set of State Variables, which is determined by the Resource type. There are defined two relations that may link Resources between different types – the **host-component** relation and the **parent-sub** relation - figure 1.

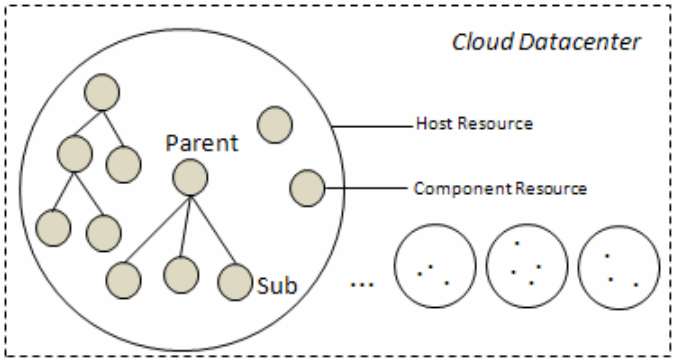

Figure 1. Resource relations

B. *Topology*

In our resource framework described in [15], management of datacenter resources is handled by a set of super-peers that we call Management Servers. The super-peers dynamically balance the load of resource management in a manner that each super-peer is responsible for a separate sub-set of the datacenter resources. The details of this mechanism [16] are not important to our exposition, but as we shall see in the Analysis section, the presence of such a distributed resource management is integrated very well with our data-store concept decisions, and influences the performance of query searches.

III. RESOURCE INFORMATION MANAGEMENT

On figure 2 is shown a simple draft of our three-layer system architecture. The Admin Applications are interested in management and monitoring of datacenter resources. They interact with the System layer that handles all generic activities like partial processings and load balancing.

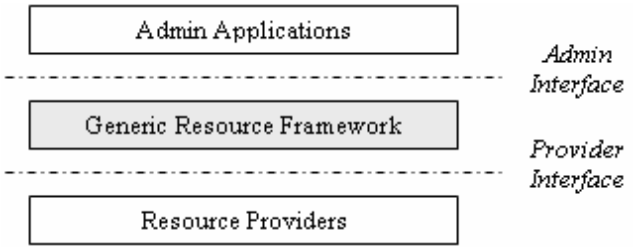

Figure 2. System architecture

The System retrieves resource information from the appropriate Resource Providers in the bottom layer. There is one Resource Provider for each resource type, which handles directly the communication with the underlying resource and maintains the persistency of resource information if needed. On behalf of the Providers, the System could be configured with ready-to-use access to different types of storages (RDBMS, LDAP, Round-robin, DFS), so that Providers to be able to use interface to a respective storage without caring about the configuration and availability of the database. For federation scaling purposes the System could be configured with arbitrary

number of storage instances of the same type. Providers are completely unconstrained about choosing the storage type/instance, and the format of their tables and data. On figure 3 is shown how a resource information request flows through the System, so that its data to be flexibly retrieved from arbitrary place and in arbitrary manner by the Provider.

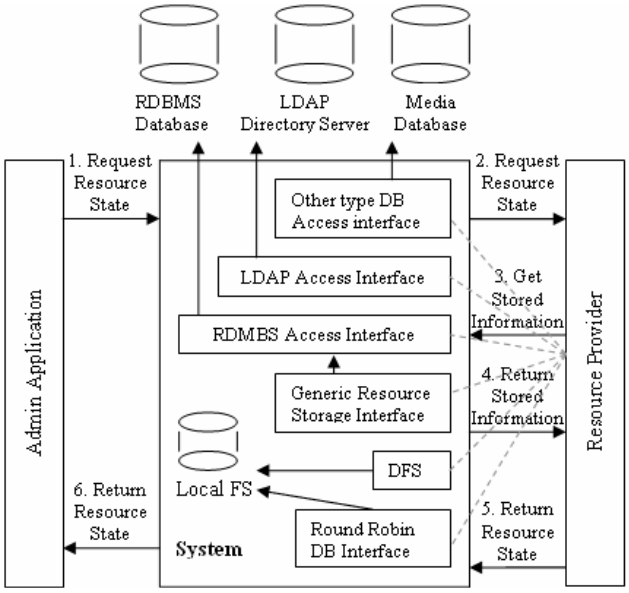

Figure 3. Request-response flow

The Admin Application requests a resource state from the System (1). Notice that Admin Applications view resources behind the generic resource abstraction and do not deal with direct database queries. The System redirects the call to the appropriate Provider for the respective resource type (2). The Provider gathers the resource information in his own way, potentially using some storage access interface provided by the System (4) and (5). Note that there is no constraint over the structure and format of data; the Provider would find his resource data in the same shape, in which it was populated by him via the same storage interface. The Provider represents the resource data into the generic abstraction and returns it to the System (5). The result is brought up to the Admin Application (6). Note that steps (3) and (4) are optional and up to the Provider to be performed. The Provider may also get the resource information via external mechanisms not dealing with System facilities, or can just return runtime information if storage is not needed for his type of resources.

IV. Search Scenarios

A main issue we introduce with our flexible data store is what kinds of searches we can perform if data is spread so unrestrictedly. In our framework, we can separate the search evaluations into three main sub-groups - presented respectively in sections A, B and C. The separation is done on the basis of participation of a complex cross-resource-type relation.

Note: *in this paper, we will provide filter texts as self-explained examples, since specifying complete filter grammar is not in focus of our exposition and current goals*.

A. No cross-type relation

This case encompasses filtering of resources on the basis of State Variable conditions in the scope of single Resource type. A meaningful search example would be to list all resources of type “application” with name “foo” and state “not responding”. A filter explaining this criterion could look as follows:

application: (name == foo & state == not_responding)

B. Host-component relations

This includes more complex queries that join information from different resource types on our host-component relation (defined in Chapter II). The meaning of this search is to enable administrators to find datacenter nodes (or other logical Host Resources) that “contain” or “not contain” other Component Resources satisfying criteria on their states; and the opposite – to find Component Resources that belong to Hosts satisfying criteria on the Host state. A meaningful example would be to find the datacenter nodes that contain certain application instance named “foo”, and the cpu usage for that node is underutilized (<20%). The filter could look as follows:

node: (cpu-usage < 20 & contains(app: (name == foo)))

In this example, we used an operator named “contains” to express relation with a Component Resource that is contained within the searched Host Resources. The reversed relation search would be to find the Component Resources of type “application” with name “foo”, that belong to Hosts with cpu usage under 20%. We will use “belongs” for this relation:

app: (name == foo & belongs(node: (cpu-usage < 20)))

C. Parent-sub relations

This case includes joining of information on our parent-sub relations (Chapter II). For instance, we if there are Component Resources of type *application* with sub-Component Resources of type *statistic* that provide some runtime performance metrics about the parent *application*, we may need to search for all application instances with certain performance statistics. A possible expressing of such filter would be:

app: (name == foo & sub(statistic: (cache-hit-rate < 50)))

Analogously to the host-component relation, we can use the parent-sub relation in both directions resource filters:

statistic: (cache-hit-rate < 50 & parent (app: (name == foo)))

V. The Resource Provider Filter Plugin

To enable searches based on resource states criteria, the System employs two basic approaches. One is to retrieve resource states from the Providers and then to perform state by state matching against the search filter. This approach seems far not efficient, since it discards any db level elimination of not needed data and does not prevent loading of potentially redundant information from db into the main memory. Even though, this approach could be good enough for certain types of resources considering their specific management use-cases. In order to enable utilization of the standard searches provided by databases, we define that a Resource Provider may optionally implement a so called - Filter Plugin interface. There

could be one Filter Plugin for a given resource type that should be able to interpret logical filters, whose base operands are State Variable conditions. The job of the Filter Plugin is to translate the filter into its storage-specific filtering abilities and retrieve a filtered set of requested resource entities. We define the interface of a Filter Plugin to be consisted of one method:

- *listResources(Filter)*

The result expected from the Plugins is a set tuples representing resource instance elements that satisfy the input filter. Each tuple should be consisted of the following three properties of a resource: {*resource-id*, *host-id*, *parent-id*}. This limited information is completely enough for the System to perform the 'joins' on the possible cross-resource-type relations as we shall see further.

In general, each Plugin is obliged to perform filtering only in the scope of its resource type. Since Plugins have the knowledge on the structure of data they employ, it would be easy for them to translate the generic filter into a storage and schema specific query. For instance, utilizers of RDBMS storage can transform the filters into SQL queries regarding the exact tables employed. If we consider example generic filter:

(name == foo & state == not_responding)

The example SQL transformation could be:

*select * from my_applications where app_name = 'foo' and app_state = 'not_responding'*

LDAP utilizers can translate this example into a suitable LDAP search with respect to the structure of their entries:

(&(name=foo)(version=xxx))

Utiliziers of the local file system can brute-forcibly read unordered entries and test whether the entry attributes satisfy the search criteria (no real benefit achieved). Or they can read only part of all entries if data is semi-structured at some degree into suitable directories or ordered within specific file names.

VI. Performing Searches

Our strategy will be to decompose the filters into parts suitable for passing them to the separate Filter Plugins. The partial results obtained from Plugins should then be consolidated so that correct filtered set to be produced in the end. To prepare the ground for such decomposition, we first classify the following distinguishable types of filter Operands:

- Simple Operands – name-value conditions targeted to the searched resource type. For instance: *name == foo*
- Join Operands – filters on cross-type resources that are in certain relation with the resources of the requested type. In Chapter IV, we used the so called Operators "*contains*", "*belongs*", "*parent*" and "*sub*" to express the concrete relations. All of them accept a nested filter for resources of a new type. Such a Join Operator together with its nested filter we distinguish as a Join Operand. The nested filters may in turn be composed of both - Simple and Join Operands.

Our processing algorithm is recursive, so that cross-type relations could be nested in arbitrary depth.

A. Processing filters composed only of Simple Operands

In this simplest case all filtering conditions are on the State Variables of the requested resource type. There are no cross-type relations so the filter could be fully forwarded to the *listResources(Filter)* method of the respective Resource Plugin. The result produced by the Plugin will be final result for the processed filter.

B. Processing filters with Join Operands

Distinguishing the Simple and Join Operands as described above, we transform the filter into Minimal Disjunctive Normal Form (MDNF) using the famous Quine-McClusky algorithm for minimizing Boolean expressions. The filter gets transformed into the following disjunction:

$$F = K_0 \mid K_1 \mid \dots K_n \quad (1)$$

where each K_i is a conjunction of Simple and Join Operands:

$$K_i = SP_{i0} \&... SP_{ir} \& JP_{i0} \& \dots JP_{is} \quad (2)$$

SP_{ij} and JP_{ij} stand respectively for Simple and Join Operands, each of which may potentially participate with negation.

The $(SP_{i0} \&... SP_{ir})$ part of the conjunctions is interpreted with the means described in Section A. The set produced by this partial filter is further intersected by each of the JP Operands remained in the conjunction. Table I shows the fields on which the sets are intersected for the different JP Operands:

TABLE I. JOIN FIELDS IINTERSECTION

Left-hand / Right-hand Operands		**Join Operator**
SP	***JP***	
[**res-id**, host-id, parent-id]	[res-id, **host-id**, parent-id]	*contains*
[res-id, **host-id**, parent-id]	[**res-id**, host-id, parent-id]	*belongs*
[**res-id**, host-id, parent-id]	[res-id, host-id, **parent-id]**	*sub*
[res-id, host-id, **parent-id**]	[**res-id**, host-id, parent-id]	*parent*

Intersection is done on the fields in bold

To obtain the set produced by a JP Operand (before using it in intersection) we evaluate its nested filter recursively as described in this Section B, i.e. starting again with MDNF decomposition. In case the nested filter includes only Simple Operands, it is evaluated as described in Section A.

If the JP is negated in the conjunction, the System calculates a set-difference of {all available Resources of the related type} and {the Resources produced by the Operand}.

When all conjunctions are evaluated, the System merges their results by performing a set-union.

To compute union, intersection and difference of two sets, there are two general approaches. If at least one of the sets is small enough to fit in a RAM buffer, the calculation is done using a one-pass algorithm, i.e. at once without using temporarily stored results. If data is too large, the System employs a two-pass algorithm. Two-pass algorithms are

characterized by processing parts of the data, storing temporal results to the disk, then reading them again for further processing. Reference [16] provides some nice descriptions of two-phase algorithms for set-operations based on sorting and based on hashing.

VII. Analysis

A. First Order Logic

Most existing database and storage tools provide filtering capabilities only in the limited scope of the Propositional Logic. The Propositional Logic is not Turing complete limiting the problems one can define, because it cannot express criteria for the composition of data [19]. First Order Logic extends the Propositional Logic with two new quantifier concepts - specifically universal and existential quantifiers. Universal quantifiers allow checking that something is true for everything – normally supported by the 'forall' conditional element. Existential quantifiers check for the existence of something - supported with 'not' and 'exists' conditional elements.

For our search capabilities, we inescapably employ the First Order Logic to express criteria on cross-type relations. For instance we express evaluation of criteria for finding Resource Hosts that “contain” or “not contain” a specific Component Resource (recall that we support negation upon the JP Operands). This is clearly a support of existential quantifier, to which we give specific names with relevance to the context of our resource compositions – *contains, belongs, parent* and *sub*.

Regarding the ‘forall’ universal quantifier, we can say it is expressible by ‘not exists’, so we may say we provide universal quantifier capability, too. For example listing of datacenter nodes where all application instances are in *(state == ‘not_responding’)*, can actually be listed as nodes that not contain an application in *(state != ‘not_responding’)*.

B. Supporting more cross-type relations if needed

To enable such functionally rich search capabilities (with subjective to our purposes), we generalized the relations between resources, bringing these relations up in the resource abstraction – *host-component* and *parent-sub*. In this way, we mandated Resource Providers to maintain a constrained format for the resource relations so that every Resource to bring its *resource-id, host-id* and *parent-id*. Except for this obligation, we are absolutely freeing Resource Providers to use unconstrained data structures for their Resource states, as well as to choose a physical storage of arbitrary type that best fits to the resource information demand. With this approach we achieve together - flexibility in data storing and well-defined joining of data. Although our joins are not fully functional and are preliminary limited to the *host-component* and *parent-sub* relations, the employed strategy works fine for our purposes simply because we get a rich-enough search capabilities and a lightweight evaluation in the same time.

If we need to create an extended System with additional join capability on a new relation, all we have to do is to define this new relation in the resource abstraction, and to extend the tuple retrieved by the Filter Plugin interface, so that respective intersection to be done on the new join-fields. As it may be seen, although we closely relate the proposed techniques to the specifics of our resource framework, the design could be easily adopted by other infrastructures.

C. Functional comparison with other Systems

To implement the First Order Logic needed in our search scenarios we used techniques employed in relational databases where data is structured into tables and various joins are possible over the data tuples. Actually, only the relational type of database offers support for First Order Logic queries. If we need to rely on LDAP type of storage, we could never perform the ‘not exists’ with an LDAP search request. Notice, that ‘exists’ is somewhat feasible, since we can find real LDAP objects having reference to the cross-type entity, although there will remain some duplication issues.

Although R-GMA is Grid Information System with relational storage that is expected to support complex relation searches upon resources, the reality is slightly different. R-GMA uses physically federated database servers, but R-GMA is not a distributed database management system. Instead, it relies on a much looser coupling of data providers across a Grid [6] and does not support implicit joining of data from different type Producers.

D. Performance Analysis

In this chapter, we shall try to analyze what performance impact our data store should imply in comparison with a single type data-store as it is with the existing grid solutions. The workloads relative to an Information Service are:

- Loading Resource states by Admin Applications
- Populating data by Resource Providers
- Search query evaluations

Regarding retrieval of Resource states (figure 3) and population of data, we can say that our System should be never outperformed, because every resource is fetched (or updated) against the optimum possible type of storage. Talking in examples, statistics data could be fetched from Round-robin db tool where data is preliminary summarized, log entries information is read from DFS instead of heavy-weight fully-functional database, users’ data comes via some popular highly optimized user management tool working normally with LDAP

Regarding the searching performance, we should distinguish two searching sub cases. For simple queries without cross-type relations, our System should not be again outperformed, since the whole filter is translated into storage-specific query and again processed against the optimal type of storage for the respective resource. The more complicated sub case is when searches relate cross-resource-type information. We shall compare our System to R-GMA configured with a centralized relational database, since currently this is the only famous grid configuration that can support similar functional cross-type capabilities. When database is centralized, table joins for different Producers become possible, although centralization brings more serious drawbacks like the scalability bottleneck and the single point of failure.

The main difference between our System and a centralized RDBMS regarding cross-type searches is within our System-handled processing of partially executed filters. A comparative disadvantage of our System is that we retrieve from databases some redundant information that otherwise could be eliminated at db level, i.e. if the set-operations were done on the database machine instead of transporting the partial sets to our processing node. Since we talk about transporting redundant information, we chose the network volume as a representative comparison metric.

TABLE II. NEWORK VOLUME COMPARISON

Set Operation	Size of Set Operands		Network Volume	
	Set 1	*Set 2*	*RDBMS Processing*	*Processing by our System*
Union	M	M	3/2 M	2M
Intersection	M	M	1/2 M	2M
Difference	2M	M	M	3 M

In table II, we show a relative difference of the data volume transmitted from db when the employed set-operations are performed by RDBMS or by our System. If we assume that each of the 3 operations has equal probability to be performed, then our drawback is 3M versus 7M, or we can conclude that we retrieve 7/3 (~2.3) times more data from database for each set-operation performed on our processing nodes.

As compensation to this drawback, our approach introduces a set of performance advantages and speed ups. In chapter II-B, we mentioned that our processing nodes are organized as cluster of super-peers. Since our cluster nodes work over independent not-intersected sets of Resources (exact mechanisms are explained in [16]), our set-operations become a distributed job with high parallelization efficiency. Additional unignorable speed up we can introduce with this parallelization is possible saving of the disk I/Os for the two-phase algorithms within set-operations. As stated in VI, two-phase algorithm would be needed if the smallest data-set is too large to fit in a RAM buffer. In our cluster of super-peers, the size of each retrieved set is reduced by a factor of N, where N is the number of super-peers. This provides an opportunity for Cloud System owners to configure their datacenters with an adequate number of super-peers with regards to the volume of the managed datacenter, so that two-phase operations to be eliminated due to the reduced amount of processed data by each super-peer. To make a distinction, the disk I/O savings would not be possible if set-operations are processed on a single database machine.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a flexible design of an Information Service that allows arbitrary storage type and unconstrained data structures to be used for various resources. We facilitate the administration of resources by decupling the Query Interface from the underlying data structures. We also provide a lightweight and still rich functionality regarding search capabilities. As for the performance, our System seems promising to be outperforming existing Grid Information Services, but this should be further proven. Our future work will be concentrated on creating a prototype and conducting comparative measurements that demonstrate the viability of the system, particularly in an actual deployment.

REFERENCES

[1] Plale, B., P. Dinda, and G. Laszewski, “Key Concepts and Services of a Grid Information Service”. ISCA 15th International Parallel and Distributed Computing Systems (PDCS), 2002.

[2] Arkills, B (2003). LDAP Directories Explained: An Introduction and Analysis. Addison-Wesley Professional. ISBN 020178792X.

[3] Beaulieu, Alan (April 2009). Mary E Treseler. ed. Learning SQL (2nd ed.). Sebastapol, CA, USA: O'Reilly. ISBN 978-0-596-52083-0.

[4] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. XML Path Language (XPath) 2.0. Technical Report W3C Working Draft, Version 2.0, World Wide Web Consortium, December 2001. http://www.w3.org/TR/xpath20/.

[5] Round Robin Database Tool (RRDtool). http://oss.oetiker.ch/rrdtool.

[6] R-GMA System Specification Version 6.2.0: http://www.r-gma.org/documentation/specification.pdf

[7] Codd, E.F. (1970). "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM 13 (6): 377–387. doi:10.1145/362384.362685.

[8] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor and R. Wolski, A Grid Monitoring Architecture, Global Grid Forum, Lemont, Illinois, U.S.A., GFD.I.7, January 2002

[9] Howes, T.: RFC 1960: A String Representation of LDAP Search Filters. IETF. (1996)

[10] A. Cooke, A.J.G. Gray, L. Ma, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Byrom, L. Field, S. Hicks, J. Leake, M. Soni, A.Wilson, R. Cordenonsi, L. Cornwall, A. Djaoui, S. Fisher,N. Podhorszki, B. Coghlan, S. Kenny, D. O’Callaghan, R-GMA: an information integration system for grid monitoring, in: Proceedings of the 10th International Conference on Cooperative Information Systems, 2003.

[11] A Globus Primer 4 available at http://www.globus.org/toolkit/docs/4.0/key/

[12] Douglas Thain, Todd Tannenbaum, and Miron Livny, "Condor and the Grid", in Fran Berman, Anthony J. G. Hey, Geoffrey Fox, in Grid Computing: Making the Global Infrastructure A Reality, John Wiley, 2003, ISBN:0-470-85319-0.

[13] Hawkeye Team. Hawkeye, Monitoring and Management Tool for Distributed Systems, 2004. http://www.cs.wisc.edu/condor/hawkeye

[14] Zhelev, R., V. Georgiev. 2010. Load Balanced Resource Management for Cloud Systems. 4th International Conference on Information Systems and Grid Technologies, ISGT'2010, 28-29. May 2009, Sofia, Bulgaria.

[15] Zhelev, R., V. Georgiev. 2010. Generic Resource Framework for Cloud Systems. 5th International Conference Distributed Computing and Grid Technologies in Science and Education, GRID'2010, 27. June - 04. July 2010, Dubna, Russia.

[16] H. Garcia-Molina, J. Ullman, J. Widom, Database Systems: The Complete Book, Prentice Hall, Upper Saddle River, New Jersey, 2002.

[17] Kobsa, A. and Fink, J.: An LDAP-Based User Modeling Server and its Evaluation. User Modeling and User-Adapted Interaction: The Journal of Personalization Research 16, (2006) 129-169, DOI 10.1007/s11257-006-9006-5.

[18] R. M. Smullyan. First-Order Logic. Springer-Verlag: Heidelberg, Germany, 1968.

[19] Duda, C., Kossmann, D., Zhou, C. – Predicate-based indexing for desktop search (Lang.: eng). – In: VLDB journal, 19(2010)5, pp. 735-758

[20] W. Xing, O. Corcho, C. Goble, and M. Dikaiakos, “Information Quality Evaluation for Grid Information Services,” submitted to CoreGrid Symposium in conjunction with Euro-Par 2007.