Benchmark Details of Synthetic Database Benchmark/Workload for Grid Resource Information

Beth Plale, Craig Jacobs, Ying Liu, Charlie Moad, Rupali Parab and Prajakta Vaidya

Computer Science Department
Indiana University
Bloomington, Indiana

Technical Report TR583

Abstract

The Grid is a new paradigm for wide area distributed computing. Management of grid resource information on the grid is complicated by the fact that grid resources such as hosts, clusters, virtual organizations, people, mathematical libraries, software packages, and services have aspects of their description that change at millisecond rates. This defining characteristic makes traditional directory service solutions inappropriate. Our work contributes to the understanding of resource information representation and retrieval in grid computing through the development of a grid-specific synthetic database benchmark/workload for a grid resource information repository, and the application of the benchmark/workload to three platforms. The benchmark/workload is a set of queries and 'scenarios' developed from a platform-neutral data model of grid resources.

The contribution of this paper is threefold: the first is a synthetic database benchmark/workload. The queries are meaningful in a grid context, assume an underlying data model made up of realistic grid resources, and populated with realistic data. The second is the application of the database benchmark on three vastly heterogeneous "database" platforms: a relational database, mySQL 4.0, that uses the SQL query language; Xindice 1.1, a native XML database that uses XPath as its query language; and MDS2, an LDAP database that uses LDAP as its query language. The final contribution is a metric that captures both tangible and less tangible aspects of information retrieval. Some of the results we obtain are unexpected; others provide quantified results to substantiate suspicions that the grid community already holds.

The appendix gives details of the benchmark, including the representation of queries and scenarios in English, SQL, XPath, and LDAP.

1 Introduction

The Grid is a new paradigm for wide area distributed computing wherein resources are organized as web services that can be flexibly and dynamically allocated and accessed, often to solve problems requiring resources that span multiple administrative domains [12]. A grid resource information repository is any repository of information about grid resources such as hosts, clusters, virtual organizations, people, mathematical libraries, software packages, and services [8, 14]. This repository might be a database, a grid registry, or might be information exported by a grid service. The defining characteristic about grid resource information that makes traditional directory service solutions inappropriate is that some resources have aspects of their description that change at millisecond rates [10]. For instance, a host is often described by attributes such as

the operating system version, chipset, and amount of memory; attributes that change on the order of every few months or more. Additional attributes, such as current CPU load and available virtual memory change far more rapidly, on the order of seconds. Given the anticipated size of the grid and number of physical resources that are expected to participate, management and representation of grid resource information is an important problem.

Our work contributes to the understanding of resource information representation and retrieval in grid computing through the development of a grid-specific synthetic database benchmark/workload for a grid resource information repository, and the application of the benchmark/workload to three platforms. The benchmark/workload is a set of queries and 'scenarios' developed against a data model representing grid resources that extends the Global Grid Forum [1] (GGF) proposed GLUE schema [9]. The database schemas for the three platforms are derived from this single data model using well-defined derivation rules. The databases are populated with synthetic but realistic data about grid resources. The data are realistic in terms of the types of entities represented, the proportions of the entities to one another, and attribute values that describe the entities. For instance, the database holds information about 60 clusters, 388 subclusters, 20 users, and 33605 hosts. The *queries* test a broad range of database functionality while asking realistic questions. The *scenarios* are short synthetic workloads that test query response time of a repository under a workload of concurrent query requests and update requests.

The benchmark is applied to three platforms that could viably host a grid information server: a relational database, mySQL, that uses the SQL query language; Xindice [5], a native XML database that uses XPath as its query language; and MDS2, an LDAP database that uses LDAP as its query language. Results of the performance evaluation are assessed on the metric of query response time and the less tangible but equally important ease of use. *Query response time* is the elapsed time from when a client issues a query and a response is received. *Ease of use* attempts to capture the less tangible aspects of a service by quantifying the amount of work a client must undertake in order to obtain desired information. Ease of use is expressed in terms of total amount of data returned to a user, and number of queries that must be written to describe the needed data. For instance, a request for data that can be stated in one query and then returns exactly the needed 1KB of data has a higher ease of use than one that requires 6 queries to express and returns a total of 1MB of data that the user must then manipulate. The cost metric, query response time and ease of use, has a decidedly client focus. This reflects our belief that the final judge of a service is the level of satisfaction that it provides to a user.

The contribution of this paper is threefold: the first is a synthetic database benchmark/workload. The queries are meaningful in a grid context, assume an underlying data model made up of realistic grid resources, and populated with realistic data. The composition of the query set addresses numerous orthogonal interests: simple queries versus complex, queries that test specific features versus those that could be posed by an actual user, sequential access versus concurrent access, small return sets versus large. The second contribution is the application of the database benchmark on three vastly heterogeneous "database" platforms: relational, XML, and LDAP. The repository deployed on each is mid-sized, representing something that a single organization might deploy as a single server for all its resource information. The final contribution is the complex metric that quantifies the less tangible aspects of information retrieval. We argue that this metric based on query response time and ease of use is more meaningful in service-based environments such as the Grid than are more traditional single variable metrics such as throughput.

Some of the results we obtain are unexpected; others provide quantified results to substantiate suspicions that the grid community already holds.

The paper is organized as follows: following immediately is a discussion of related work. Section 3 introduces the data model underlying the synthetic database benchmark/workload. Section 4 introduces the benchmark/workload following an organization of grouping the queries into logical subsets for ease of understanding. In Section 5 we discuss the performance evaluation, consisting of performance evaluation,

including the execution of the benchmark/workload against the three database platforms, and observations that can be drawn from the experiment. Section 6 offers concluding remarks and a glimpse at future work.

2 Related Work

The University of Wisconsin benchmark [7] was developed to evaluate the performance of off the shelf relational database management systems. It was applied against a relatively small database of synthetic tables and data, and queries were designed solely to test features of the database. The Transaction Processing Council's decision support benchmark (TPC-H) [15] is an application specific benchmark for decision support. It is closest to ours in spirit, but does not capture the unique dynamic needs of a grid information service. The XMark project [16] provides a framework to assess the abilities of an XML database.

A number of people have examined aspects of performance of the Grid Information Server, an early server for resource information. The definitive reference implementation of a Grid Information Server for the Grid has been Globus MDS [8] and earlier versions. MDS is widely deployed on grids across the world. MDS2, the version used in this study, is organized as a hierarchical architecture of lower level Grid Resource Information Servers (GRIS), connected to one or more higher-level index servers (GIIS). Multi-layer hierarchies of 4-5 levels are not uncommon [2]. MDS2 and earlier versions employed the LDAP information model and protocol. The new release of the Globus Toolkit, which is a reference implementation for the Global Grid Forum OGSA [18] grid services architecture, shifts management of grid resource information onto the registries, which like UDDI [19], are directory services for discovery of web services. Information management is also shifted onto the resource itself, which is responsible for providing a web service that exports the information about the resource.

Smith [17] examined MDS performance using different versions of LDAP. The work predates MDS' hierarchical GRIS/GIIS architecture so the main result of the paper is exposing LDAP's poor performance under even minor update loads. Aloisio *et. al* [3] studied the MDS grid information server and conducted experiments that were focused on simple tests of the Grid Index Information Service (GIIS). Schopf [22] recently examined the scalability of three information servers, MDS2, RGMA [11], and Hawkeye of the Condor system [6], all of which are tightly coupled to monitoring systems from which they obtain input data. These systems were subject to scalability testing under increasing user loads and focused on the ability of the system to handle increasing connection load. A single, simple query used is throughout. Schopf's work complements ours; ours is focused on a broad set of queries and scenarios issued against a rich data set.

3 Resource Information Data Model

The resource information data model is a database platform-neutral representation of the entities in a system and the relationships that exist between them. The data model is neutral in that it allows relationships to be expressed between entities but does not mandate how these relationships should be implemented. A relationship between a user account and a subcluster might be represented as a separate table of rows showing subclusters to which a user has access, or might be implemented as a nested pointer in an object.

The data model used in our work is based on the GGF proposed GLUE schema v8 as of October 2002 [9]. The GLUE schema defines entities representing clusters, subclusters, hosts, processors, jobs, and computing Elements to name a few (see Figure 1.) We gathered use cases by talking to managers of production high performance computing systems for the purpose of having in the benchmark queries that are realistic. It is through this process that we decided to extend the GLUE schema with entities representing people, user accounts, and communication channels between machines. Since taking the snapshot of the GLUE schema,

some of our extensions have become part of the GLUE data model.

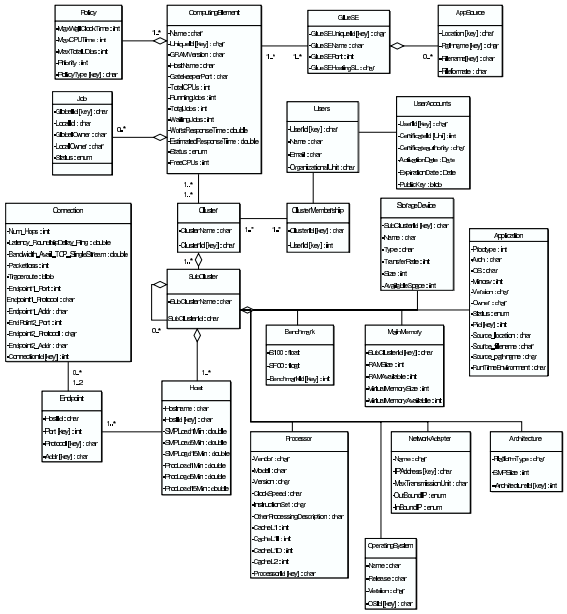

Figure 1: UML data model used by benchmark/workload.

The key to understanding the diagram in Figure 1 is to focus on the ComputingElement, Cluster, SubCluster, and Host hierarchy that runs vertically down the figure. A ComputingElement, which can be thought of intuitively as a queue in a batch scheduling system, serves one or more Clusters, while a Cluster has one or more ComputingElements. A SubCluster belongs to one or more Clusters, and to zero or more SubClusters. A Host belongs to one or more SubClusters. Entities such as MainMemory and Processor are attributes grouped functionally that describe a SubCluster. A Connection represents a communication channel such as a TCP socket that is currently active between two endpoints.

The data model is a necessary starting point from which database specific schemas are derived. The act of deriving a specific schema from a data model represented as a UML diagram is defined by well known rules [20], though the same experienced judgment required to develop good data models in the first place is needed to ensure a compliant representation of the derivation. We use the term schema to describe the description of all entities and relationships in a particular database. The term has another common meaning as a definition of syntax for an XML document type. The data model is also a necessary starting point for the benchmark itself since the representation of a request for data (a query) encoded in a language more precise than English requires knowledge of the underlying data representation. The benchmark is the topic of the next section.

Terminology used when referring to entities in a data model is very dependent on the implementation. As shown in Table 1, what is known as a 'table' in a relational database is referred to as an 'entry' in LDAP and a 'collection' in Xindice. Whereas an individual instance or member of a relational table is called a 'tuple', in LDAP it is called an 'object', and in Xindice it is called a 'document'. For purposes of this paper, we use the term 'collection' to refer to a collection of instances, and 'object' to refer to a data element. These are shown italicized in the table. We refer to the fields that describe a member as *attributes*.

The databases are populated with pseudo synthetic values informed from several large MDS dumps dated between November 2000 and January 2002 that we obtained. Each of the three database platforms hold the same entities and relationships. The number of instances of entities and relationships is also held constant

	Relational	*LDAP*	*Xindice*
collection level	table	entry	*collection*
member level	tuple	*object*	document

Table 1: Terminology for three data models.

across the three platforms. A database platform contains 34 entities/relationships and 81684 instances. In following with the standard adopted by the GLUE schema, a relation between two entities is represented as a separate collection. The distribution of instances among the entities shown in Figure 1 are given in Table 2 for a sampling of the entities.

Collection	*Number of Objects*
Cluster	20
UserAccounts	60
ComputingElement	106
Subcluster	345
Application	600
Connection	12200
Host	29743

Table 2: Object distribution for the major collections.

4 Grid Resource Benchmark/Workload

The kinds of queries and updates issued against a grid resource repository can vary widely, limited only by the user's knowledge of the query language, the expressiveness of the query language, and limitations of the underlying implementation. Our goal is a set of queries that when taken as a whole exercise several orthogonal axes: simple queries versus complex, queries that test specific search support versus those that could be posed by a user, sequential versus concurrent access; small return set versus large. The intent is to provide a synthetic database workload that is both broad and representative of actual workloads so as to accurately assess the strengths and weaknesses of different grid resource information repositories. This section introduces the benchmark/workload.

The synthetic database benchmark is built upon a data model of entities and relationships. This would be the case for any database benchmark. The key is that the benchmark is developed against a *data model-neutral* representation of entities and relationships so the benchmark has no bias in that regard. The synthetic database workload that we developed consists of 16 queries and updates and four scenarios. The queries/updates are grouped into five major categories: scoping, index, join, selectivity, and base operations; the grouping is for purposes of ease of understanding. Over half the queries are paired for purposes of testing the presence or absence of a feature (*e.g.*, an index). Features not undergoing testing are controlled across the pairs.

Scoping. A scope defines a starting point for a search. It is relevant in schemes having a nested or hierarchical organization. By stating the starting scope of a query, (*i.e.,* starting point in a tree), one can restrict query evaluation to a particular subtree. As this often results in increased efficiency in hierarchically organized data, scoped queries should perform well in hierarchal databases (*i.e.*, MDS2, Xindice.)

The 'scoping' queries consist of two pairs of queries (four queries in total.) The first pair tests over a smaller object collection and the scope is set at one level above the desired objects. Specifically, the scoped query asks for all subclusters for a given cluster whereas the non-scoped query asks for the subclusters directly. The subcluster table is of moderate size, that is, 345 objects.

The second scoping query spans three levels of the ComputingElement-Cluster-SubCluster-Host hierarchy to retrieve information from the much larger Host table (approximately 30000 objects). The Host table describes all hosts, or individual computers, served by the grid resource information repository.

Indexing. Query response times are often dramatically improved when indexes are used. Indexes provide fast access for queries that request indexed attributes. Our query set includes one index pair, that is a pair of queries wherein the independent variable is whether or not the requested value is indexed.

Selectivity. The selectivity of a query is the number of objects returned. According to DeWitt [7], coverage of the performance domain can be achieved with queries that return 1 tuple, 1% of tuples, and 10% of tuples. These queries execute over the Connection table which contains information about 12200 active network connections.

Joins Joins occur when a user requests information that resides in more than one table. Joins can occur either across tables, or as multiple joins over a single table. The latter is called a self-join and occurs when a user asks for more than one instance of a particular resource. For instance, a user could request four hosts that meet a specific criteria such as amount of memory. Our query benchmark includes two distinct (*i.e.* non-paired) join queries: the first queries over six collections; the second query is a realistic job submission query, specifically, a user is seeking a subcluster wherein the needed software environment exists, the job owner has an account, and the binary is resident. The purpose of this latter query is not to test a different aspect of a system, but to ask a realistic, grid related question; one that might be posed by a scientist desiring to find a specific set of nodes on which her binary can and is allowed to execute.

Other Operations. The two final queries in the benchmark are connection request and update request. Connection request is a request to the database that consists only of a connection request followed immediately by a disconnection request. Update updates a single attribute in a set of objects that match a particular condition.

The benchmark consists of a set of scripts for each database. The mySQL scripts are issue SQL queries and are written in Perl. They communicate with the database using the mySQL C API. The Xindice queries are written in XPath. The scripts are written in Java and interface with Xindice using XML:DB. The MDS queries are written in the LDAP query language. The scripts are written in C and interface to MDS2 using the LDAP protocol. All scripts iterate 1000 and 10000 times. The full number of iterations was not achieved in practice for all platforms. Even at 1000 iterations Xindice in particular had excessive execution times that prevented the completion of the script. The call to the database is blocking. There is no delay between calls. For queries that are implemented as multiple calls to the database, query timing terminates the moment the final result set is produced.

Scenarios. A scenario is a scripted synthetic workload issued over a controlled time duration consisting of concurrent query and update requests. Scenarios are a key part of the benchmark as they expose the sensitivity of query response time to increasing update rates. We noted earlier that dynamic data attributes that require millisecond updates to stay current is a unique feature of grid information repositories, hence the scenarios expose an important aspect of performance of a database platform. A scenario is scripted as

follows: in Phase I, a number of concurrent clients are started. A client repeatedly issues a blocking query request to the database. This phase is to determine response time under no update load. In Phase II, update clients are added and execute concurrently. The updates and queries execute for the duration of Phase II before the updates are terminated. The query clients then continue alone for Phase III. The final phase captures the lingering effects of the updates.

The total scenario execution time is on the order of 10's of minutes. Specifically, for the results presented here, a scenario runs for a duration of 10 minutes. Query response time is sampled every 20 seconds, at time t_i where $0 < i < 1200$ and $i \ mod \ 20 = 0$. Query response time is defined as:

t_0: query response time at t_0 is 0 ($t_0 = 0$)

$t_1 - t_n$: query response time for t_i, i ! = 0, is the average of query responses received in the interval $(t_{i-1}, t_i]$. If no query responses received in the interval, then

$$\mathsf{queryResponseTime}(\mathsf{t_i}) = \mathsf{queryResponseTime}(\mathsf{t_{i-1}})$$

The definition accommodates the case of no query responses received over an interval. This case is common, particularly for complex queries over Xindice and MDS where a 20-40 second interval may not be large enough to capture even a single query response.

5 Performance Evaluation

In the performance evaluation we undertook the application of our synthetic query benchmark/workload to three database platforms: mySQL 4.0, Xindice 1.1, and MDS 2 (GT 2.2.) MySQL is configured with the InnoDB back end. InnoDB tables support foreign keys, provide ACID properties, row-level locking, and non-locking read in SELECTS for increased concurrency and performance. Xindice 1.1 is an XML open source database. Unlike Xindice 1.0, which was a standalone server, 1.1 is bootstrapped from an Apache Tomcat server. Based on results not reported here, we saw large reductions in query response times from Xindice 1.0 to 1.1; much of which can be attributed to the significant performance improvements in 1.1. The performance overhead of passing through the web server, however, is roughly 10% as reported on the Xindice users mailing list on April 2003 by a frequent contributor. MDS2 is configured as a single GRIS talking to a single GIIS with GRIS and GIIS co-located on the same dual processor server. All queries are issued against the GIIS.

The underlying hardware for all three databases is a dual processor Dell Poweredge 6400 Xeon server, 2GB RAM, 100 GB Raid 5, RedHat 7.3. The client platform is a Sun Blade 1000, SunOS 5.8; access is through switched Fast Ethernet. Each database is implemented as a standalone server and the client scripts are standalone clients on a separate machine.

Consistency across the databases is ensured so that a query issued against a relational database that returns 1000 tuples will also return 1000 documents when issued against the Xindice database. To ensure this consistency, the databases are populated in a chained fashion starting from a single Perl script used to create tables and indices in mySQL then to populate the tables. Xindice is created and populated from a script that reads and processes a dump of the mySQL database. MDS2 is created from a PHP script closely modeled on the mySQL Perl script; the PHP script creates an LDIF file, the format recognized by LDAP, that is then read by a simple provider external to MDS that reads the LDIF file and pipes the file to the *stdout* pipe on which the GRIS is waiting. Standalone XML schema compliance checking is performed using Xerces [4] during population of the Xindice database but no checking is performed during performance evaluation. This chaining has proved to be extremely useful when the databases needed upgrading or when a consistent state needs to be restored.

It should be noted that MDS2 query response times are approximated. We are resolving an issue in getting MDS2 to scale to the size of database that we use in this experiment. To obtain the results shown here, we interpolated from measurements done on database sizes of 5%, 10%, and 25% of the full size. We will refine these numbers as progress is achieved.

5.1 Query Response Time

Query response time is a measure of the amount of time it takes for a server to complete a query request and return the result set. For the queries in the benchmark (not scenarios), since query scripts are nonthreaded and blocking, a script executes one query at at time. The query response times, captured in Figures 2 through 6, show results organized by the query groups described in Section 4. Listed across the X-axis of each are the individual queries and their results for the three different platforms. If a query is part of a binary pair, its pair resides to its right and is prefaced with 'non'. The Y-axis plots query response time in milliseconds. It is important to note that the Y-axis scale is logarithmic.

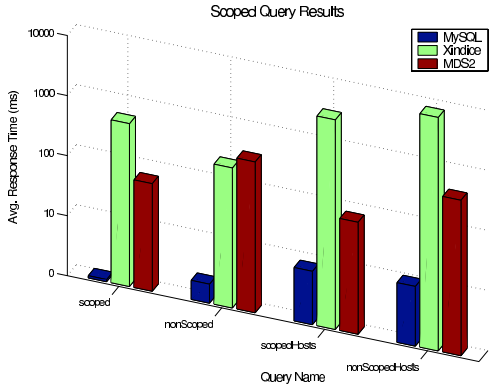

Figure 2: Scoped: scoping limits a search to a particular subtree. Hierarchical databases should show good results for *scoped* queries (but not *non-scoped*.)

We are surprised by the scoping results shown in Figure 2. Scoping should favor hierarchical databases because scope limits the search space. This is not the case with Xindice for *nonScopedHosts*. What appears to be overshadowing the benefits of scope is the number of XPath queries that must be issued to the database in order to implement one higher level query. From results reported in [21], we observed a linear correlation between the number of XPath queries that must be issued per query and the query response time.

The indexed results of Figure 3 show a clear benefit of using index support in mySQL and Xindice. MDS2 does not use the native index support of openLDAP [13] but instead employs a cache in the GIIS and serves requests from cache. Hence there is no difference between the indexed and non-indexed queries for MDS. Due to difficulties in getting data to stay in the GIIS caches, the MDS numbers shown are all satisfied out of cache. Our solution is to give the cached data objects extended times to live; this was required in order to get the queries to not fail as objects disappeared.

Selectivity is defined as the number of objects that satisfy a query. From the measurements shown in Figure 4, we can conclude that query response time is not sensitive to number of objects returned for both mySQL and Xindice.

The join queries shown in Figure 5 measure a repository's ability to assemble a result from numerous collections. The results of the two queries are roughly the same, which is reasonable since both queries touch the same number of collections.

An interesting observation can be made by comparing the join results in Figure 5 with the selectivity results of Figure 4. The join queries take on the order of 2.5 ms for mySQL whereas *selectivity1*, on the other hand, completes in 53.24 ms. Similarly for Xindice, joins finish in approximately 1000 ms whereas *selectivity1* completes in 38,572.33 ms. The reason is that though the joins are over six collections, the collections are small, on the order of hundreds of objects. The collection over which the selectivity queries are executed contains 12,200 objects. The results indicate that collection size impacts performance more than does number of joins.

The *update* times shown in Figure 6 are for simple, one-attribute updates. Thus they show an upper bound on the rate at which a database can accept updates. For mySQL this rate is 41 updates per second whereas for Xindice the rate is 0.2 updates per second. The low update rate for Xindice is an overriding factor in the conclusion we draw that Xindice is generally inappropriate as a platform for a grid resource repository. It also explains the odd behavior shown in the scenarios discussed next.

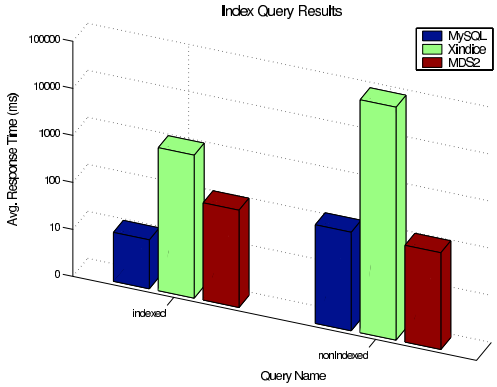

Figure 3: Indexed: *indexed* is a query over an indexed attribute; *nonIndexed* queries over a non-indexed attributes. Data repositories that employ indexes, mySQL and Xindice in this case, should perform significantly better for an indexed query than a nonindexed one. MDS uses a cache instead of indexes.

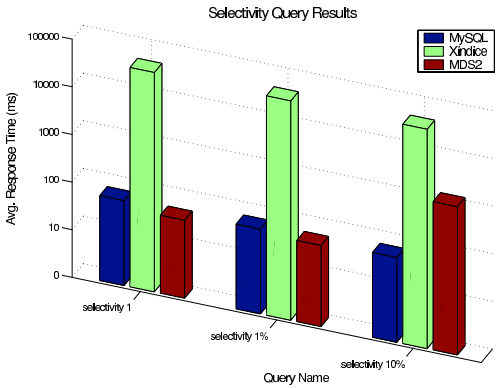

Figure 4: Selectivity: selectivity is number of objects returned from a query. *Selectivity 1* returns one query from a collection of 12,200 objects; *selectivity 1%* returns 1%; *selectivity 10%* returns 10%. The three queries together give an indication of the sensitivity of query response time to number of objects returned.

Scenarios. The *scenarios* are scripted synthetic workloads designed to capture the sensitivity of query response time to update load. Each scenario, as depicted in Figures 7 and 8, begins with the execution of three concurrent query streams. This is Phase I. Phase II starts three minutes into the run when update threads start up. In the case of mySQL, ten to fifty update threads are started. In the case of Xindice, given its exhibited poor update rate, only three update threads are run. In Phase III, the update threads are terminated and the query threads allowed to run through until the end of the run. As mentioned in the caption of Figure 6, MDS2 does not support updates through the traditional client query interface. As such, we forced updates for purposes of the scenarios by issuing a client query that queried a single attribute that has a short time to live. In that way, every time the query is executed, it forces the value to be updated through the provider mechanism (*i.e.*, LDIF file piped to stdout) described in paragraph three of Section 5.

Two scenarios are tested. In Scenario1, shown in Figure 7, the queries are simple in that they do not access the same collection and they touch only small collections. The impact of update streams is observable in all three cases. MySQL average query response times are 2-10 millisecond range when no update threads are running, and increase to 10-100 milliseconds under the update load. Xindice varies between 1000-2000 milliseconds when no update threads are running and 2000-4000 milliseconds when they are. MDS2 results fall between that of mySQL and Xindice, but the reader is asked to note that the MDS scenario is run on a database that is significantly smaller (95% smaller) than the database size used for the other platforms. We expect response times to be slower by an order of magnitude or

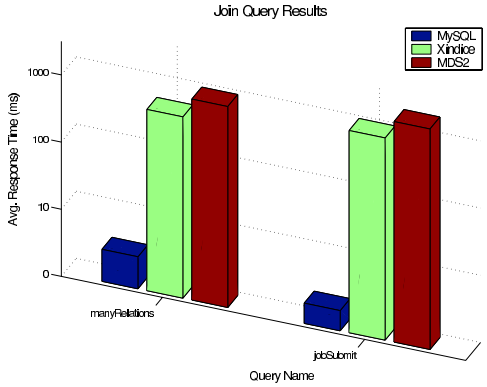

Figure 5: Joins: *manyRelations* touches six different collections in evaluating the query. *JobSubmit* also touches six collections in asking a realistic job-related question.

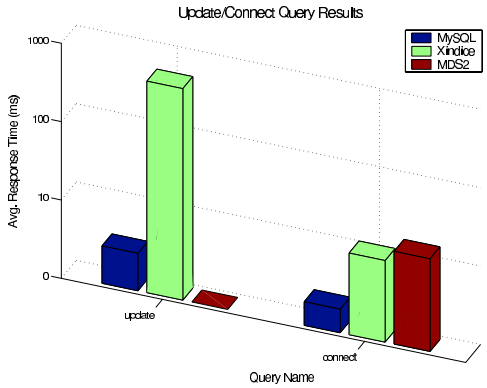

Figure 6: Other: *Update* modifies a single attribute in a subset of a collection. In MDS2 updates through the query interface are disabled so this query is not able to capture update time. *connect* connects to the database, disconnects and returns.

more when applied to a full sized database. The failure of the MDS2 curve to taper down at the end of the scenario could be due to a lingering effect of rapid cache refresh being triggered for the particular attribute after the query request has terminated.

In Scenario 2, the queries are more complex in both dimensions of collection overlap and collection size. As can be seen, Xindice behavior becomes irregular. Shown across the top of the graph are response times for two of the three Xindice queries. These are generally regular through the duration of the run, spiking noticeably during update execution. The large vertical spike at the end of the graph is the delayed processing of the third Xindice query. It begins a full two minutes after other activity has concluded and completes a single execution in an inordinately long time. The update threads, not shown, behave in similarly non-uniform ways.

5.2 Ease of Use

The *ease of use* metric attempts to capture the less tangible aspects of performance of a grid information service, in particular, the amount of work a client must undertake in order to obtain desired information. That is, the number of queries that a user must issue, and the amount of processing required on the returned data that falls upon the user. Though more difficult to quantify, ease of use is an important metric not only for assessing the friendliness of a grid resource information repository, but for obtaining early understanding of the workload a service might encounter when

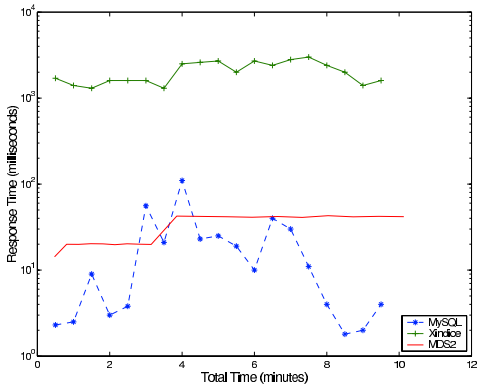

Figure 7: Scenario1: Average query response time of three simple queries on small collections, with no overlap in access. MDS2 response times are obtained on a database that is significantly smaller (5%) than the others. Scenarios show sensitivity of query response time to rapid updates.

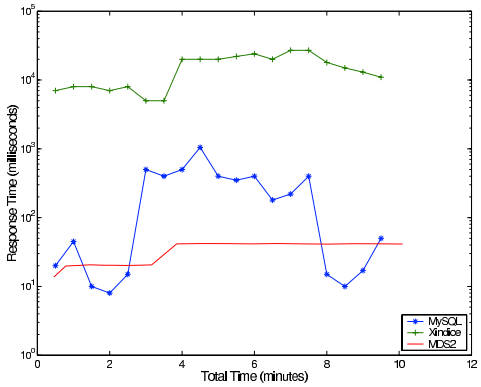

Figure 8: Scenario2: similar to Scenario1 but for queries on larger collections and that overlap. MDS2 database is significantly smaller (5%) of the others.

under intense load in production use. For instance, if one service can respond with the exact data set in one request, and a second server requires an exchange of six request/response sequences before the exact data set is received at the client, the former will scale better under increased workloads.

The metrics used to quantify ease of use are number of bytes returned and number of queries needed in order to retrieve the requested data. These numbers, shown in Tables 3 and 4, are not independent. Database platforms requiring more queries to obtain data correspondingly return a larger number of bytes. The problem could lie in limitations in the database platform or in the query language that the platform supports. For instance, a hierarchy of collections in Xindice must be searched one collection at a time; that is, by issuing one XPath query per level. If the client were interested in all subclusters belonging to the cluster "titan", it would have to issue a query to search the ClusterSubcluster relationship to retrieve all documents for which "titan" is a parent cluster. The client would then for each document process it to extract the subcluster ID and issue a second query to retrieve the processor ID from the Processor table.

The problem is exaggerated somewhat by the finer granularity decomposition of attributes in the GLUE schema. For instance, Processor, Operating System, and MainMemory are separate objects. This finer granularity has the advantage of reducing the size of a database by eliminating redundancies, but for hierarchical languages like XPath and LDAP, it comes at a cost of additional queries.

Description	*mySQL 4.0 (KB)*	*Xindice 1.1 (KB)*	*MDS2 (KB)*
scoping	0.4 - 46.0	7.5 - 549.5	*5.6 - 47.3*
indexing	9.5 - 11.0	139.8 - 140.9	*9.6 - 24.0*
selectivity	0.04 - 52.9	0.48 - 691.0	*0.03 - 267.0*
joins	0.03 - 0.03	40.1 - 131.8	*0.98 - 1.9*

Table 3: Minimum and maximum number of bytes returned per query group. The MDS numbers are estimated from smaller database sizes so are shown in italics.

Description	*mySQL 4.0*	*Xindice 1.1*	*MDS2*
scoping	1	3	3
indexing	1	2	1
selectivity	1	1	1
joins	1	6	5

Table 4: Maximum number of queries issued to database per higher-level query.

6 Conclusions and Future Work

There is much more to the assessment of a grid resource information repository than what is conveyed by our metric of query response time and ease of use. Authentication and access control, distribution, and replication are a all important issues. Yet this study contributes a deeper understanding of the impact of queries, query languages, and workloads on grid resource information repositories through the development of a synthetic database benchmark/workload that is tailored to the types of objects and collections that exist in a grid resource information repository, and in particular addresses the unique aspect of a grid information server, namely the rapid update rates that its dynamic objects must undergo. It is through the inclusion of *scenarios*, or scripted synthetic workloads, that we address this unique aspect. Issues of distribution are ongoing work.

The foremost work on our agenda is to ascertain the limits to scalability we are witnessing with MDS2 and include our final assessment in the paper. A nice extension of the work would be to a XML/relational database. The XML font end would preserve the benefits of an XML solution, important in a grid services/OGSA [18] architecture, while addressing the performance problems highlighted here of a native XML solution.

A highly distributed partitioning of grid resource information, such as is used in Globus Toolkit 3.0, addresses the update rate problem by limiting the number of concurrent queries and updates against any one source, but it incurs a cost in query response time where queries must be partitioned and results assembled. We are examining a distributed version of our benchmark such that these costs can be quantified.

We are in the process of making the benchmark accessible via a portal so that others can utilize the results. This future work can be of broader practical benefit to the Grid community by tailoring the benchmark to a common subset of the schema that is supported by multiple sites. Then it would be possible to use the benchmark for exploratory queries to a grid information server to assess its current load.

7 Acknowledgments

We are deeply appreciative to the following people who have contributed to the success of this study: Peter Dinda, Northwestern, for tracking down and sharing the MDS dumps; Akshay Sharma for injesting the MDS dumps to write the primary data population script; Dennis Groth, Indiana University, for assistance in specifying the mySQL queries; and Ben Clifford, Globus team, for help in get MDS populated.

References

[1] Global grid forum. http://www.gridforum.org, 2003.

[2] Rob Allen. CLRC HPCGrid services portal. http://esc.dl.ac.uk/GridWG, 2003.

[3] G. Aloisio, M. Cafaro, I. Epicoco, and S. Fiore. Analysis of the globus toolkit grid information service. Technical Report Technical Report GridLab-10-D.1-001-GIS_Analysis, GridLab Project. http://www.gridlab.org/Resources/Deliverables/D10.1.pdf.

[4] Apache. Xerces java parser. http://www.apache.org/xerces-j, 2003.

[5] Apache. Xindice. http://xml.apache.org/xindice, 2003.

[6] Jim Basney and Miron Livny. *High Performance Cluster Computing*. Prentice Hall PTR, 1999.

[7] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking database systems: A systematic approach. Technical report, University of Wisconsin, Computer Sciences Dept., Madison, Wisconsin, 1983.

[8] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

[9] DataTAG. Glue schema: common conceptual data model for grid resources monitoring and discovery. http://www.cnaf.infn.it/ sergio/datatag/glue, 2003.

[10] Peter Dinda and Beth Plale. GWD-GIS-012-1 Unified relational approach to grid information services. http://www.gridforum.org/1_GIS/gis.htm, 2001.

[11] Steve Fisher. Relational model for information and monitoring. Technical Report GWD-Perf-7-1, Global Grid Forum, 2001.

[12] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.

[13] OpenLDAP Organization. OpenLDAP. http://www.openldap.org, 2003.

[14] Beth Plale, Peter Dinda, and Gregor von Laszewski. Key concepts and services of a grid information service. In *ICSA 15th International Parallel and Distributed Computing Systems*, September 2002.

[15] M. Poess and C. Floyd. New TPC benchmarks for decision support and web commerce. *ACM SIGMOD Record*, 29, December 2000.

[16] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, China, August 2002.

[17] Warren Smith, Abdul Waheel, David Meyers, and Jerry Yan. An evaluation of alternative designs for a grid information service. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.

[18] Steve Tuecke and Nirmal Mukhi. Grid services and web services tutorial. http://www.globusworld.org, January 2003.

[19] UDDI.org. Universal description, discovery, and integration (UDDI). http://www.uddi.org, 2003.

[20] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Prentice Hall, 1997.

[21] Prajakta Vaidya and Beth Plale. Benchmark of xindice as a grid information server. Technical Report Technical Report TR-585, Computer Science Dept., Indiana University, 2003.

[22] Xuehai Zhang, Jeffrey L. Freschl, and Jennifer M. Schopf. A peformance study of monitoring and information services for distributed systems. In *IEEE International High Performance Distributed Computing (HPDC)*, 2003.

A Details of Benchmark/Workload

The following provides the details of the benchmark/workload. Each 'query' is introduced with an English language statement of the intent of the query. Following this are the SQL, XPath, and LDAP representations of the query. As can be seen in both the XPath and LDAP cases, some queries require multiple communications with the server and additional user level processing.

B Queries

Q1: Scoped "Give me the list of subclusters and their configuration information (ID, Operating System type, and processor type) for cluster "mds.sdsc.edu"". The result set is 10 objects.
SQL:

```
SELECT CSC.ClusterId, CSC.SubClusterId, SCO.OSId, SCP.ProcessorID
FROM Cluster_SubCluster as CSC, SubCluster_Processor as SCP,
  SubCluster_OperatingSystem as SCO
WHERE
    CSC.ClusterId = "mds.sdsc.edu" and
    CSC.SubClusterId = SCO.SubClusterId and
    CSC.SubClusterId = SCP.SubClusterId;
```

XPath: The queryXindice() function performs a select operation on an entire tuple and the EvaluateQuery() function performs a query and extracts a particular attribute.

```
queryXindice("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/Cluster_SubCluster",
  "//Cluster_SubCluster[@ClusterID='mds.sdsc.edu']");

EvaluateQuery("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_Processor",
  "//SubCluster_Processor[@SubClusterId='"+subClusterId+"']","ProcessorID");

EvaluateQuery("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_OperatingSystem",
  "//SubCluster_OperatingSystem[@SubClusterId='"+subClusterId+"']","OSId");
```

LDAP: The first query searches the objectclass RGRClusterSubCluster for subclusters belonging to cluster "mds.sdsc.edu". The query returns a list of subclusters matching the criteria. The second and third queries are each executed once per subcluster in the list.

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob,o=Grid", LDAP_SCOPE_SUBTREE,
  "(&(objectclass=RGRClusterSubCluster)(GlueClusterUniqueID=mds.sdsc.edu))",
  "GlueSubClusterUniqueID", 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob,o=Grid", LDAP_SCOPE_SUBTREE,
``(&(objectclass=RGRSubClusterOperatingSystem)(GlueSubClusterUniqueID="scId[j]''),
"RGROperatingSystemID", 0 , &result)

ldap_search_s(ld, "Mds-Vo-name=GIISdquob,o=Grid", LDAP_SCOPE_SUBTREE,
``(&(objectclass=RGRSubClusterProcessor)(GlueSubClusterUniqueID="scId[j]''),
"RGRProcessorID", 0, &result)
```

Q2: NonScoped "Give me a list of subclusters that are running the Mac OS." Search is on indexed field (that is, OSId). Result set is 60 objects.
SQL:

```
SELECT SCO.OSId, SCO.SubClusterId
FROM SubCluster_OperatingSystem as SCO
WHERE
 SCO.OSId = "Mac OS";
```

XPath:

```
queryXindice("xmldb:
 xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_OperatingSystem",
"//SubCluster_OperatingSystem[@OSId= 'Mac OS']");
```

LDAP:

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob,o=Grid", LDAP_SCOPE_SUBTREE,
 "(&(objectclass=RGRSubClusterOperatingSystem)(RGROperatingSystemID=Mac*))",
"GlueSubClusterUniqueID",
0, &result)
```

Q3: ScopedHost "Give me list of hosts belonging to cluster mds.sdsc.edu." Result set is 914 objects.
SQL:

```
SELECT SCH.HostId, CSC.SubClusterId, CSC.ClusterId
FROM SubCluster_Host as SCH, Cluster_SubCluster as CSC
WHERE
  CSC.ClusterId = "mds.sdsc.edu" and CSC.SubClusterId = SCH.SubClusterId;
```

XPath:

```
queryXindice("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/Cluster_SubCluster",
  "//Cluster_SubCluster[@ClusterID='mds.sdsc.edu']");

queryXindice("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_Host",
  "//SubCluster_Host[@SubClusterId='"+subClusterId+"']");
```

LDAP: Retrieve set of SubCluster IDs for give ClusterID (first query). For each subcluster, issue second query to obtain HostID.

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob,o=Grid", LDAP_SCOPE_SUBTREE,
  "(&(objectclass=RGRClusterSubCluster)(GlueClusterUniqueID=mds.sdsc.edu))",
  "GlueSubClusterUniqueID";
  0, &result)

ldap_search_s(ld, "Mds-Vo-name=GIISdquob,o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRSubClusterHost)(GlueSubClusterUniqueID= **subclusterID**'',
``GlueHostUniqueID'',
0, &result)
```

Q4: NonScopedHost "Give me list of hosts that have 'sharkestra' in hostname." Search is on indexed field. Result set is 1554 objects.
SQL:

```
SELECT H.HostId
FROM Host as H
WHERE
 H.HostId > "sharkestra50";
```

XPath:

```
queryXindice("xmldb:xindice:
 //bitternut.cs.indiana.edu:8080
 /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/Host/",
 "//Host[starts-with(@HostId,'tamarack')]");
```

LDAP:

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob,o=Grid", LDAP_SCOPE_SUBTREE,
  "(&(objectclass=GlueHost)(GlueHostUniqueID=tamarack*))",
  "GlueHostUniqueID", 0, &result);
```

Q5: Indexed "Give me list of hosts that have single processor load for 1 minute equal to some value. Search is on indexed field. Result set is 405 objects.
SQL:

```
SELECT H.HostId
FROM Host as H
WHERE
 H.ProcLoad1Min = .40;
```

XPath:

```
queryXindice("xmldb:xindice:
 //bitternut.cs.indiana.edu:8080
 /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/Host",
 "//Host[@ProcLoad1Min = 0.4]");
```

LDAP:

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob,o=Grid", LDAP_SCOPE_SUBTREE,
  "(&(objectclass=GlueHost)(GlueHostProcessorLoadLoad1Min=40))",
  '\0', 0, &results);
```

Q6: NonIndexed "Give me a list of hosts that have an SMP load equal to one value (.50) or another (.40)." Search is on non-indexed field. Result set is 356 objects.
SQL:

```
SELECT H.HostId
FROM Host as H
WHERE
  H.SMPLoad15Min = .50 or H.SMPLoad15Min = .40;
```

XPATH:

```
queryXindice("xmldb:xindice:
 //bitternut.cs.indiana.edu:8080
 /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/Host",
 "//Host[@SMPLoad1Min=0.5 or @SMPLoad1Min = 0.4]");
```

LDAP:

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob,o=Grid", LDAP_SCOPE_SUBTREE,
  "(&(objectclass=GlueHost)(|(GlueHostSMPLoadLoad15Min=50)(GlueHostSMPLoadLoad15Min=40)))",
  '\0', 0, &results);
```

Q7: Many Relations "Give me list of subclusters belonging to cluster 'mds.sdsc.edu' that are Linux-based, have Pentium processor, have copy of Globus 2.2 in current status of 'Not Running'." Result set is 1 object.
SQL:

```
SELECT SCO.OSId, SCP.ProcessorID, CSC.SubClusterId, AP.Pid
FROM SubCluster_Processor as SCP, SubCluster_OperatingSystem as SCO,
  Cluster_SubCluster as CSC,  SubCluster_Application as SCA, Application as AP
WHERE
  CSC.ClusterId = "mds.sdsc.edu"  and  CSC.SubClusterId = SCO.SubClusterId and
  CSC.SubClusterId = SCP.SubClusterId  and CSC.SubClusterId = SCA.SubClusterID and
  SCA.Pid = AP.Pid  and SCO.OSId = "Linux" and SCP.ProcessorId  = "PENTIUM" and
  AP.RunTimeEnvironment = "Globus 2.2" and AP.Status = "NOT RUNNING";
```

XPath:

```
queryXindice("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/Cluster_SubCluster",
  "//Cluster_SubCluster[@ClusterID='mds.sdsc.edu']");

queryXindice("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_Processor",
  "//SubCluster_Processor[@ProcessorId = 'PENTIUM']");

queryXindice("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_OperatingSystem",
  "//SubCluster_OperatingSystem[@OSId = 'Linux']");

queryXindice("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_Application",
  "//SubCluster_Application[@SubClusterId = '"+SC.get(j)+"']");

queryXindice("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080/db/GlueGeneralTop/Application",
  "//Application[@Pid = '"+ pid + "' and @Status = 'NOT RUNNING' and
  @RunTimeEnvironment='Globus 2.2']");
```

LDAP: The first query searches the objectclass RGRClusterSubCluster for subclusters belonging to cluster "mds.sdsc.edu". The second query selects subclusters whose Operting System is "Linux" based from subCluster list from the 1st query. With the results from second query, third query does further selection for processor="Pentium". The fourth query does final selection based on list from the third query.

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRClusterSubCluster)(GlueClusterUniqueID=mds.sdsc.edu))",
"GlueSubClusterUniqueID*", 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRSubClusterOperatingSystem)(&(GlueSubClusterUniqueID=GlueSubClusterUniqueID*)(RGRO
"GlueSubClusterUniqueID**", 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRSubClusterProcessor)(&(GlueSubClusterUniqueID=**)(RGRProcessorID=PENTIUM*)))",
"GlueSubClusterUniqueID***", 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRApplication)(&(GlueSubClusterUniqueID=***)(&(RGRAppStatus=NOT RUNNING)(RGRAppRunT
"GlueSubClusterUniqueID****", 0, &result);
```

Q8: Job Submit "Of machines in cluster XX, give me a list of subclusters and their total RAM that are running Linux, on which I have the binary YY resident, and I have an active account." Result set is 1 object.
SQL:

```
SELECT CSC.SubClusterId, M.RAMSize, SCA.Pid, App.Owner, App.OS
FROM Cluster_SubCluster as CSC, SubCluster_Application as SCA,
  Application as App, UserAccounts as UA, ClusterMembership as CM,
  MainMemory as M
WHERE
  CSC.ClusterId = "mds.sdsc.edu"  and CSC.SubClusterId = SCA.SubClusterId and
  SCA.Pid = App.Pid and App.Owner = "aksharma" and App.OS = "Linux" and
  App.Source_filename = "/u" and CM.UserId = "aksharma" and  UA.ActivationDate <
  now() and UA.ExpirationDate >= now() and  UA.UserId = CM.UserId and
  CSC.ClusterId = CM.ClusterId and M.SubClusterId = CSC.SubClusterId;
```

XPath: queryXindice() is shorthand for an XML:DB call to XPathQueryService.query(). Levels in parentheses indicate the loop nesting at which the query occurs. A level 1 query is not nested within a loop, a level 2 query is nested within one loop, etc. The first query retrieves the users record. From it, the client extracts activation date and expiration date and computes to determine if account is active. The second query (also at level 1) find clusters to which the user has access. The first level 2 query extracts a lisdt of subclusters for each cluster to which the user has access. The level 3 queries are performed on each subcluster in the list - the first query obtains RAM size. The second finds those that have applications. The third finds the applications with the right characteristics.

```
queryXindice("xmldb:xindice:   (level 1)
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/Users/UserAccounts",
  "//UserAccounts[@UserId = 'aksharma']");

queryXindice("xmldb:xindice:    (level 1)
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/ClusterMembership",
  "//ClusterMembership[@ClusterId='mds.sdsc.edu' and @UserId = 'aksharma']");

queryXindice("xmldb:xindice:   (level 2)
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/Cluster_SubCluster",
  "//Cluster_SubCluster[@ClusterID='mds.sdsc.edu']");

queryXindice("xmldb:xindice:   (level 3)
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/MainMemory",
  "//MainMemory[@SubClusterId = '" + subClusterId + "']","RAMSize'');

queryXindice("xmldb:xindice:   (level 3)
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_Application",
  "//SubCluster_Application[@SubClusterId = '" + subClusterId + "']");

queryXindice("xmldb:xindice:    (level 3)
  //bitternut.cs.indiana.edu:8080/db/GlueGeneralTop/Application",
  "//Application[@Pid = '" + pid + "' and @Owner = 'aksharma' and
  @OS = 'Linux' and @Source_filename = '/u']");
```

LDAP: The first query tests whether user account on cluster mds.sdsc.edu is active. If so, second query tests whether user has access to cluster. If so, the third query retrieves list of subClusters of cluster "mds.sdsc.edu". Based on list, fourth query selects subClusters with right application properties and returns list of subClusterID. With those IDs, retrieve RAM size for each subCluster.

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(&(&(objectclass=RGRUserAccounts)(RGRUserAccountID=plale))(RGRUserAccountActivationDate<now()))(R
"RGRUserAccountID", 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(&(objectclass=RGRClusterMembership)(RGRUserID=bitternut))(GlueClusterUniqueID=bitternut.cs.indian
"GlueClusterUniqueID", 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRClusterSubCluster)(GlueClusterUniqueID=bitternut.cs.indiana.edu))",
"GlueSubClusterUniqueID*", 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(&(&(&(objectclass=RGRApplication)(GlueSubClusterUniqueID=*))(RGRAppOS=Linux))(RGRAppOwner=plale)
"GlueSubClusterUniqueID**", 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRSubClusterMainMemory)(GlueSubClusterUniqueID=subClusterID**))",
"GlueHostMainMemoryRAMSize GlueSubClusterUniqueID", 0, &result);
```

Q9: One Tuple Selectivity "Give me list of connections matching the number of hops and bandwith criteria." Neither attribute is indexed. Result set is 1 object.
SQL:

```
SELECT Endpoint1_Addr, Endpoint1_Port, Endpoint2_Addr, Endpoint2_Port
FROM Connection
WHERE
  Num_Hops = 20 and Bandwidth_Avail_TCP_SingleStream = 65.62;
```

XPath:

```
queryXindice("xmldb:xindice:
 //bitternut.cs.indiana.edu:8080
 /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/Connection",
 "//Connection[@Num_Hops = 20 and @Bandwidth_Avail_TCP_SingleStream = 65.62]");
```

LDAP:

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(&(objectclass=RGRConnection)(RGRConnectionNumHops=20))(RGRConnectionAvailBandWTCP=6529))",
"RGRConnectionEndPoint1Address RGRConnectionEndPoint1Port RGRConnectionEndPoint2Address RGRConnectio
```

Q10: One Percent Selectivity "Give me list of connections having Num_Hops of 33." Attribute is non-indexed. 1% of objects returned. Result set is 131 objects.
SQL:

```
SELECT Endpoint1_Addr, Endpoint1_Port, Endpoint2_Addr, Endpoint2_Port
FROM Connection
WHERE Num_Hops = 33;
```

XPath:

```
queryXindice("xmldb:xindice:
 //bitternut.cs.indiana.edu:8080
 /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/Connection",
 "//Connection[@Num_Hops = 33]")
```

LDAP:

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRConnection)(RGRConnectionNumHops=33))",
"RGRConnectionEndPoint1Address RGRConnectionEndPoint1Port RGRConnectionEndPoint2Address RGRConnectio
0, &result);
```

Q11: Ten Percent Selectivity "Return list of connections." 10% of Connection objects are returned. Result set is 1428 objects.
SQL:

```
SELECT Endpoint1_Addr,Endpoint1_Port,Endpoint2_Addr, Endpoint2_Port,Num_Hops
FROM Connection
WHERE
  Num_Hops > 89;
```

XPath:

```
queryXindice("xmldb:xindice:
 //bitternut.cs.indiana.edu:8080
 /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/Connection",
 "//Connection[@Num_Hops > 89]");
```

LDAP:

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRConnection)(RGRConnectionNumHops>=89))",
"RGRConnectionEndPoint1Address RGRConnectionEndPoint1Port RGRConnectionEndPoint2Address RGRConnectio
0, &result);
```

Q12: Attribute Update Update nonIndexed field for subset of subclusters Result set is 11 objects updated.
SQL:

```
UPDATE MainMemory
SET VirtualMemoryAvailable = (VirtualMemoryAvailable * 1.05)
WHERE
   MainMemory.SubClusterId > "sharkestra01" and
   MainMemory.SubClusterId < "sharkestra12";
```

XPath: First query returns list of subclusers matching the criteria. Second operation uses XUpdate to update the attribute.

```
queryXindice("xmldb:xindice:
 //bitternut.cs.indiana.edu:8080
 /db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/MainMemory",
 "//MainMemory[starts-with(@SubClusterId,'sharkestra')]");

"<xu:modifications version=\"1.0\"" +"xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
 "<xu:update select=\"/MainMemory[@SubClusterId = '"
+ key+"']/@VirtualMemoryAvailable\">" + Double.toString(MemAvail) + "</xu:update>"
+ "</xu:modifications>";
```

LDAP: MDS2 makes a distinction between users, who are only given query access rights to the grid information server, and providers who update values in the grid information service. The query below was run on an openLDAP server that was loaded with same schema as MDS2. First we need to get current value of VirtualMemoryAvailable against the condition about MainMemory.SubClusterId. Then we use ldap_modify_s() to update this value.

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(&(objectclass=RGRSubClusterMainMemory)(GlueSubClusterUniqueID>sharkestra01))(GlueSubClusterUnique
"GlueHostMainMemoryVirtualAvailable",
0, &result);

ldap_modify_s(ld, "RGRSubclusterMainMemory=**,o=subclustermainmemory,o=rgr,Mds-Vo-name=GIISdquob, o=G
LDAPmodification)
```

Q13: Connect SQL:

```
mysql_real_connect(&dbhandle,"bitternut.cs.indiana.edu",username,password,
       database_name,0,NULL,0)))
```

XPath:

```
queryXindice("xmldb:xindice:
  //bitternut.cs.indiana.edu:8080
  /db/GlueGeneralTop/ComputingElement/Cluster",
  "//Cluster[@ClusterId='mds.sdsc.edu1']");
```

LDAP:

```
ldap_open("bitternut.cs.indiana.edu", 2135)
ldap_simple_bind_s(ld, user, passwd)
```

C Details of Scenarios

Scenario 1: Easy Queries Scenario Queries and updates issued over collections with small cardinalities and queries access collections that are separate from those accessed by the updates (*i.e.*, no reader/writer dependencies.)
SQL:

```
//  returns 85 rows from table of 388.   Search is on indexed field.
SELECT SubClusterId
FROM SubCluster_Processor
WHERE ProcessorId = "Cyrix";

// returns 60 rows from table of 388.  Search is on indexed field.
SELECT SCO.OSId, SCO.SubClusterId
FROM SubCluster_OperatingSystem as SCO
WHERE
  SCO.OSId = \"Mac OS\";

// returns 81 rows from table of 388.
SELECT SCO.OSId, SCO.SubClusterId
FROM SubCluster_OperatingSystem as SCO
WHERE
  SCO.OSId = \"AIX\";

// updates 69 rows from table of 600.  Search (on Arch) is non-indexed
UPDATE Application
SET  Minosv = Minosv+1 WHERE Arch='sparc';

//  updates 69 rows from table of 600.
UPDATE Application
SET  Minosv = Minosv+1 WHERE Arch='sparc';";

//  updates 63 rows from table of 114.  Search (on GlueSEPort is
non-indexed.)
UPDATE GlueSE
SET GlueSEPort = GlueSEPort + 1 WHERE GlueSEPort > 10000;
```

XPath: Update process is as follows: first, select record from collection using queryXindice. Second, apply extract methods to extract specific attributes from selected records. Third, perform the updates ons on the fields. Fourth, post record back to database using xupdate command.

```
// Select 1
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_Processor",
"//SubCluster_Processor[@ProcessorId= 'Cyrix']");

TextExtractor(content,"SubClusterId");

// Select 2
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_OperatingSystem",
"//SubCluster_OperatingSystem[@OSId= 'Mac OS']");

TextExtractor(content,"OSId");
TextExtractor(content,"SubClusterId");

// Select 3
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/SubCluster_OperatingSystem",
"//SubCluster_OperatingSystem[@OSId= 'Mac OS']");

TextExtractor(content,"OSId");
TextExtractor(content,"SubClusterId");

// Update 1
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/GlueSE", "//GlueSE[@GlueSEPort > 10000]");

"<xu:modifications version=\"1.0\"" + "xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
 "<xu:update select=\"/GlueSE[@GlueSEUniqueId = '"+ key +"']/@GlueSEPort\">" +
    Double.toString(updatedglueseport) +"</xu:update>" +"</xu:modifications>";

// Update 2
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/Application","//Application[@Arch = 'sparc']");

"<xu:modifications version=\"1.0\"" +
"xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
"<xu:update select=\"/Application[@Pid = '"+ key +"']/@Minosv\">" +
  Double.toString(updateminosv) +
"   </xu:update>" + "</xu:modifications>";

//update3:
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/Application","//Application[@Arch = 'sparc']");

"<xu:modifications version=\"1.0\"" + "xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
"<xu:update select=\"/Application[@Pid = '"+ key +"']/@Minosv\">" +
 Double.toString(updateminosv) + "</xu:update>" +"</xu:modifications>";
```

LDAP: MDS2 does not support updates through the traditional client query interface. As such, we forced updates for purposes of the scenarios by issuing a client query that queried a single attribute that has a short time to live. In that way, every time the query is executed, it forces the value to be updated through the provider mechanism.

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRSubClusterProcessor)(RGRProcessorID=Cyrix*))",
"GlueSubClusterUniqueID", 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid",LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRSubClusterOperatingSystem)(RGROperatingSystemID=Mac*))",
"GlueSubClusterUniqueID", 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid",LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRSubClusterOperatingSystem)(RGROperatingSystemID=AIX*))",
"GlueSubClusterUniqueID", 0, &result);

// simulated update query for MDS2
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid",
LDAP_SCOPE_SUBTREE,"(objectclass=*)", "Mds-Fs-freeMB", 0, &result);
```

Scenario 2. Partial Overlap Scenario: Partial overlap in set of collections accessed by queries and updates. Table cardinalities are mix of large and small.

SQL:

```
SELECT * FROM Application WHERE arch='Intel Pentium IV';

SELECT * FROM Host WHERE SMPLoad1Min = 0.84;

SELECT * FROM GlueSE WHERE GlueSEPort > 35000;

UPDATE Application
SET  Minosv = Minosv + 1 WHERE Arch='sparc';

UPDATE Host
SET ProcLoad15Min = 1.05 * ProcLoad15Min WHERE SMPLoad1Min = 1.24;

UPDATE GlueSE
SET GlueSEPort = GlueSEPort + 1 WHERE GlueSEPort > 10000;
```

XPath:

```
// Query 1
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/Application","//Application[@Arch = 'Intel Pentium IV']");

// Query 2
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/Host",
"//Host[@SMPLoad1Min = 0.84]");

// Query 3
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/GlueSE","//GlueSE[@GlueSEPort < 10000]");

// Update 1
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/Application","//Application[@Arch = 'sparc']");

"<xu:modifications version=\"1.0\"" +
"xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
"<xu:update select=\"/Application[@Pid = '"+ key +"']/@Minosv\">" +
  Double.toString(updateminosv) +"</xu:update>" +"</xu:modifications>";

// Update 2
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/Host",
"//Host[@SMPLoad1Min = 1.24]");

"<xu:modifications version=\"1.0\"" +"xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
"<xu:update select=\"/Host[@HostId = '"+ key +"']/@ProcLoad15Min\">" +
  Double.toString(updatedprocloadm15min) +
"   </xu:update>" +"</xu:modifications>"

// Update 3
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/GlueSE","//GlueSE[@GlueSEPort > 10000]");

"<xu:modifications version=\"1.0\"" + "xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
"<xu:update select=\"/GlueSE[@GlueSEUniqueId = '"+ key +"']/@GlueSEPort\">" +
    Double.toString(updatedglueseport) + "</xu:update>" +"</xu:modifications>";
```

LDAP:

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRApplication)(RGRAppArchType=Intel Pentium IV))", NULL, 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=GlueHostSMPLoad)(GlueHostSMPLoadLoad1Min=84))", NULL, 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=GlueSE)(GlueSEPort<=10000))", NULL, 0, &result);

// simulated update query for MDS2
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid",
LDAP_SCOPE_SUBTREE,"(objectclass=*)", "Mds-Fs-freeMB", 0, &result);
```

Scenario 3: Multi-Attribute Update Scenario Object is held for longer period of time as multiple attributes within the object are updated.
SQL:

```
SELECT * FROM Application WHERE arch='Intel Pentium IV';

SELECT * FROM Host WHERE SMPLoad1Min = 0.84;

SELECT * FROM GlueSE WHERE GlueSEPort < 35000;

UPDATE Policy
SET MaxWallClockTime = MaxWallClockTime+1, MaxCPUTime = MaxCPUTime+1,
    MaxTotalJobs = MaxTotalJobs +1, MaxRunningJobs = MaxRunningJobs+1,
    Priority = Priority+1
WHERE PolicyType = 1000;
```

XPath:

```
// Query 1
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/Application","//Application[@Arch = 'Intel Pentium IV']");

// Query 2
queryXindice("xmldb:xindice:
//bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/Host",
"//Host[@SMPLoad1Min = 0.84]");

// Query 3
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/GlueSE","//GlueSE[@GlueSEPort < 35000]");

// Updates 1,2,3:
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/ComputingElement/Policy",
"//Policy[@PolicyType = 1000]");

"<xu:modifications version=\"1.0\"" +
"xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
"<xu:update select=\"/Policy[PolicyType = '" + key +"']/@MaxWallClockTime\">" +
Double.toString(updatedmaxwallclocktime) +
"</xu:update>" +  "</xu:modifications>";

"<xu:modifications version=\"1.0\"" +
"xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
"<xu:update select=\"/Policy[PolicyType = '" + key +"']/@MaxCPUTime\">" +
Double.toString(updatedmaxCPUtime) +
"</xu:update>" + "</xu:modifications>";

"<xu:modifications version=\"1.0\"" +
"xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
"<xu:update select=\"/Policy[PolicyType = '" + key +"']/@MaxTotalJobsTime\">" +
Double.toString(updatedmaxtotaljobstime) +
"</xu:update>" + "</xu:modifications>";


"<xu:modifications version=\"1.0\"" +
"xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
"<xu:update select=\"/Policy[PolicyType = '" + key +"']/@MaxRunningJobs\">" +
Double.toString(updatedmaxrunningjobs) +
"</xu:update>" + "</xu:modifications>";
```

LDAP:

```
ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=RGRApplication)(RGRAppArchType=Intel Pentium IV))", NULL, 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=GlueHostSMPLoad)(GlueHostSMPLoadLoad1Min=84))", NULL, 0, &result);

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid", LDAP_SCOPE_SUBTREE,
"(&(objectclass=GlueSE)(GlueSEPort<=10000))", NULL, 0, &result);

simulated multi-attribute update query for MDS2

ldap_search_s(ld, "Mds-Vo-name=GIISdquob, o=Grid",
LDAP_SCOPE_SUBTREE,"(objectclass=*)", "Mds-Fs-freeMB Mds-Fs-mount MdsFsTotal", 0,
&result);
```

Scenario 4. Record Add Update Scenario : new object added to collection.

SQL:

```
SELECT * FROM Application WHERE arch='Intel Pentium IV';

SELECT * FROM Host WHERE SMPLoad1Min = 0.84;

SELECT * FROM GlueSE WHERE GlueSEPort < 35000;

INSERT INTO Policy
SET  PolicyType = 1000*, MaxWallClockTime = 590806, MaxCPUTime=445584,
MaxTotalJobs=12000, MaxRunningJobs=550, Priority=0
```

XPath:

```
// Query 1
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/Application","//Application[@Arch = 'Intel Pentium IV']");

// Query 2
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/ComputingElement/Cluster/SubCluster/Host",
"//Host[@SMPLoad1Min = 0.84]");

// Query 3
queryXindice("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/GlueSE","//GlueSE[@GlueSEPort < 35000]");

// Update 1,2,3
// policy_xmldoc is an xml document for Policy collection to be inserted.
insertDocument("xmldb:xindice://bitternut.cs.indiana.edu:8080
/db/GlueGeneralTop/ComputingElement/Policy",policy_xmldoc);
```

LDAP:

We cannot insert any record into MDS2. So, we can just use simulated update query as before. Then, this is the same as scenario2 (Partial Overlap).