Performance Evaluation of MySQL 5.0 and Berkeley DB XML as a Grid Resource Information Manager (GRIM) with a Benchmark/Workload

Beth Plale and Xin Xiang
Indiana University
Bloomington, Indiana
{plale, xinxiang}@cs.indiana.edu

Abstract

A challenge in the distributed middleware that implements a grid envisioned to span the world, is the management of information about the resources available to the Grid. This paper describes an experimental study we undertook to better understanding the performance of the native XML database Berkeley DB XML, as a grid resource information manager system compare to MySQL 5.0. We run a benchmark set of queries along with several workload scenarios. These queries test a broad range of database functionalities through realistic questions asked on meaningful data. The scenarios are short synthetic workload that test query response time of a repository under a workload of concurrent query and update requests. Based on the results, we illuminate the strengths and weaknesses of a platform in its contribution to the requirements of a grid resource information manager.

Key Words: Grid Resource Information Management, MySQL, Berkeley DB XML, Workload, Query Response Time

1 Introduction

Grid computing is a new paradigm for wide area distributed computing. The Grid itself is a distributed middleware layer organized as web services. Critical functionality in a grid is management of information about resources existing on the grid. We define a *grid resource* as an entity that contributes value to the grid and must be managed. Grid resources include web services, hosts, file systems, databases, large scale instruments, libraries, and large-scale projects. The kinds of information describing a resource could include its owner, access policy, and current state. Resource information is used by schedulers, large data-set movers, web portals, other grid services, and users for purposes of discovery, description, and status updates. A *grid resource information manager (GRIM)* is a software entity that manages information about grid resources. [1]

This paper describes an experimental study we undertook to better understand how well the MySQL and BDB XML performs as a grid resource information manager in grid middleware. To address the question, we have developed a database of resource information that is realistic in size, and contains meaningful resource descriptions and relationships between resources. Resource descriptions are taken from the GLUE data model. This data model has wide acceptance in the Grid community. We developed a set of realistic queries and workload scenarios. The *queries* test a broad range of database functionality through realistic questions asked on meaningful data. The *scenarios* are short synthetic workloads that test query response time of a repository under a workload of concurrent query and update requests. The platforms under consideration are MySQL 5.0 and Berkeley DB XML.

2 Data Model

Our data model starting point is an October 2002 snapshot of the GLUE data model[2] defined by the Global Grid Forum. The goal of the GLUE effort is to bring together predominately US and European grid middleware researchers to reach agreement on a single data model that will ensure interoperability between efforts.

Terminology 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 a ’collection’ in Berkeley DB XML. Whereas an individual instance or member of a relational table is called a ‘tuple’, in Berkeley DB XML 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.*

	Relational	**DB XML**
Collection level	table	*collection*
Member level	tuple	document

Table 1 Terminology used in different data models

Each of the two database platforms holds the same kinds and numbers of entities and relationships. The number of objects is also held constant across the two 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 objects among collections for the major collections is shown in Table 2.

Collection	**Number of Objects**
Cluster	20
User accounts	60
Computing element	106
SubCluster	*345*
Application	*600*
Active network connection	*12200*
Host	*29743*

Table 2 Object counts for a sampling of collection types

3 Benchmark / Workload

The kinds of queries and updates that might be issued against a grid resource information manager can be broad, limited only by the user's knowledge of the query language and data, and the expressiveness of the query language. Our goal in constructing the benchmark is a set of queries that taken as a whole, exercise several orthogonal axes: simple queries versus complex, queries that test specific search support, realistic job submission requests, sequential versus concurrent access, small return set versus large. This section introduces the benchmark/workload.

The synthetic database workload consists of 11 queries and four scenarios. For descriptive purposes, we partition the benchmark into five major categories: scoping, index, join, and selectivity. Some queries are paired, that is, a single variable is tested, say the presence of an index. All other variables (*e.g.*, size of return set, collection size, string matching operators, collection size) are controlled. This pairing attempts to quantify the benefit of a variable.

Scoping. Queries that limit their search to a particular sub-tree by means of specifying a scope for the search. Intuitively scoped queries should yield better response times than their non-scoped counterparts in Berkeley DB XML.

Indexing. Our benchmark contains one index pair, which is a pair of queries wherein the independent variable is whether or not the requested value is indexed. Query response times are often dramatically improved when the attributes on which a search is done are indexed.

Selectivity. The selectivity of a query is the number of objects returned. 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 collection. Our query benchmark includes two 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. This latter query might be posed by a scientist desiring to find a specific set of nodes on which her binary can and is allowed to execute.

Scenarios. A scenario is a synthetic workload of queries and updates issued over controlled time duration. The purpose of the scenarios is to expose platform behavior under multi-user workloads. The synthetic workload consists of six streams, three of which perform repeated updates and three which perform queries. A stream contains the same request issued repeatedly. The workload was designed to quantify the effects of concurrency at a platform, and as such does capture realistic use scenarios.

Since continually changing resource descriptions are a key characteristic of resource information managers, the impact of update streams on user perceived performance is a key metric. A scenario executes as three phases:

Phase I -- a number of concurrent clients are started that repeatedly issue a blocking query request to the database.

Phase II -- update clients are added, these execute concurrently with the query clients for the duration of phase II.

Phase III -- The query clients then continue alone. This final phase captures any lingering effects the updates may have.

The total scenario execution time is configurable, but for our study a scenario runs for a duration of 20 minutes. Average query response time is calculated every 10 seconds as:

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

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 are received in an interval, i, then average QRT for the interval is taken as the QRT of the previous interval. [1]

4 Performance Evaluation

The model used in this study is a client-server model. That is, the database exists as an independent task. Client interaction is through the APIs exported by the platforms. The benchmark consists of a set of scripts for each database, one per query. The MySQL scripts issue SQL queries, are written in C, and communicate with the database using the MySQL C API. The Berkeley DB XML queries are written in XPath and XQuery, the scripts are written in C++, and the interface to Berkeley DB XML is through its own C++ library support [3]. The Benchmark cost metric is *Query Response Time* (QRT). QRT is the elapsed time as measured at the client from the issuance of the first request of a benchmark query to the receipt at the client of the last data from the last request of the benchmark query.

We also implemented different size of database for the queries and scenarios test. As we mentioned before, the data model we used in this project is an abstract representation of resource entities and their relationships. So the large size database is derived from increasing the Cluster

table size, which indicates the total number of clusters in our virtual grid environment. Because of the logic in our database deploy script, the whole database will increase as a result (almost 70% of all those tables will increase proportionally).

The architecture of the study consists of two database platforms: MySQL 5.0 and Oracle Berkeley DB XML on a dual AMD Processor 246 server, 16GB RAM. Each database is implemented as a stand-alone server. MySQL is configured with the InnoDB back end which provides row-level locking. Berkeley DB XML is a native XML database.

4.1 Basic Queries Performance

The results of applying the benchmark on the two platforms are shown in figures below. On the X-axis are the individual queries and their results for each platform. Query pairs appear next to each other. A pair is denoted by the presence of a query having the same name but prefaced with *non*. For instance, scoped and nonScoped are a pair. The Y-axis shows average query response time in milliseconds. Note that the Y-axis scale is logarithmic.

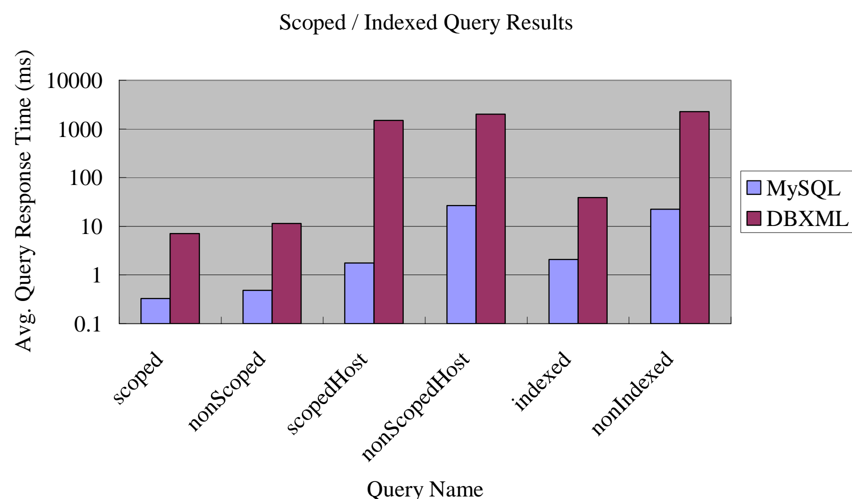


Figure 1 Query response times for scoped and indexed queries

Scoping Query Results

The leftmost four queries in Figure 1 test scoping, whereby a search is narrowed by the specification of a starting scope. While comparing the results on Berkeley DB XML for *scoped-nonScoped* pair and *scopedHosts-nonScopedHost* pair, one can see a benefit to scoping. In some XML database other than Berkeley DB XML, say Xindice, they can only accept query in XPath language, in which case we have to use multiple XPath command to implement one query. The result caused by using that kind of database, is that contrary result might be exhibited by “scoped” and “nonScoped” query. In that case the collection accessed by the query is small, and “scoped” query could run slower than “nonScoped” because scoped query is implemented as three XPath queries whereas “nonScoped” is implemented as two. The gain realized by scoping over a small collection in Xindice is overshadowed by the additional processing time required for the additional query.

Indexing Query Results

A comparison of the indexed and nonIndexed results in Figure 1 for relational and XML

platforms confirms the widely held belief that indexes greatly enhance performance.

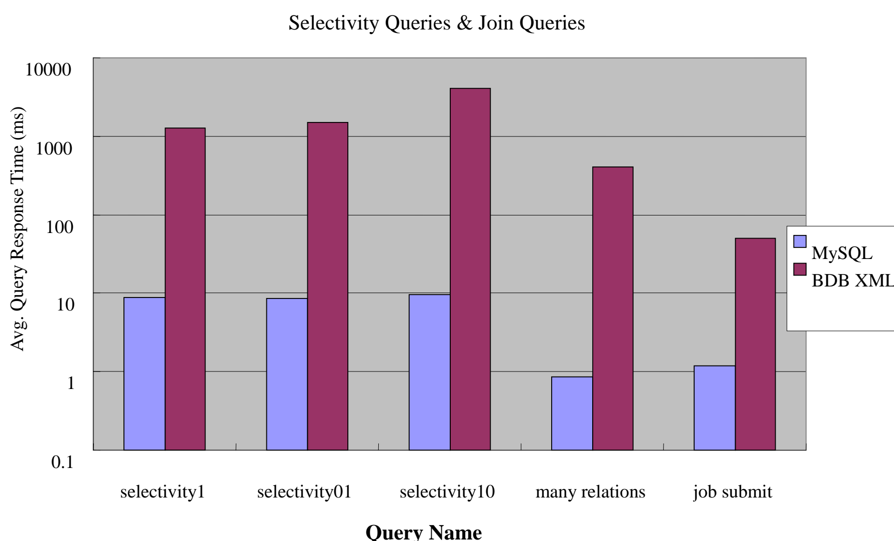


Figure 2 Query response times for selectivity and join queries

Selectivity Query Results

In Figure 2 we see that in MySQL, QRT shows no sensitivity to the number of objects returned. Berkeley DB XML, on the otherhand, shows considerable sensitivity to return set size. But actually we have two versions of selectivity queries for BDB XML – one is "long query" and another is "short query". The short query one does the same thing in Berkeley DB XML except returning the whole target nodes instead of specific information. For example, the selectivity01 query in SQL is:

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

In Berkeley DB XML, this query is implemented by using "return" clause in XQuery, along with "concat" function [4] to organize and filter the results. But in short version, we simply return the whole ConnectionTlb node. (Please refer to the query list in the appendix for details.) As a result of applying short selectivity queries, Berkeley DB XML also shows no sensitivity to the number of objects returned. Thus we can conclude that, while the number of returned objects increases, Berkeley DB XML spends more time to rebuild the result form.

The overall high response times for both two platforms are attribute to the fact that three queries access a large collection on a non-indexed attribute.

Join Query Results

The "manyRelations" and "jobSubmit" queries measure a repository's ability to assemble a result from numerous collections. The "manyRelation" and "jobSubmit" queries both touch six collections. "JobSubmit" asks the following realistic question that returns a single object as a result set: "Of the machines in cluster xx, give me a list of subclusters running Linux and their total RAM, but only where I have an active account and the binary yy is resident."

In comparing the Berkeley DB XML "manyRelation" join query with the selectivity results one can see that the join query is significantly faster. This belies the commonly held belief that

joins are too expensive and should be somehow disallowed in grid resource information management. In fact, collection size is the biggest determinant in predicting QRT than is number of joins.

4.2 Scenarios Performance

Scenarios are synthetic workloads that simulate multi-user access. They are an important part of the benchmark because they capture effects of concurrent query and update requests on user perceived performance. The goal of the scenarios is to expose the impact of various update workloads on user perceived response time. Our scenarios test the following: update to multiple attributes within a single record, insert a new record, and overlap in collections accessed. So every update thread accesses a table that is simultaneously being accessed by a query thread.

A scenario is a stand-alone client program consisting of six threads, three of which execute a query request and three an update request. When activated, each thread repeatedly issues a request to the database and blocks awaiting a response.

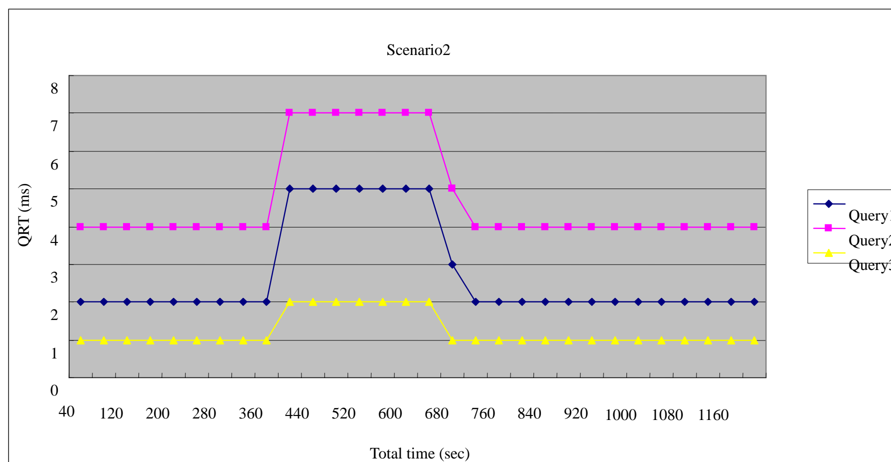


Figure 3 Scenario 2: SQL concurrent read/write overlap

The phases are easily discernable in the MySQL scenario results shown in Figure 3. Plotted are query response times for each of the three query threads that repeatedly over a 20 minute duration issue their individual request. A data point is the average QRT taken over all requests satisfied in the time interval since the last data point. During Phase I, which begins at time 0, only query threads are executing. After 360 seconds (6 minutes) into the execution, the update threads are launched and execute simultaneously with the query threads for 300 seconds (5 minutes). During Phase III only the query threads remain active.

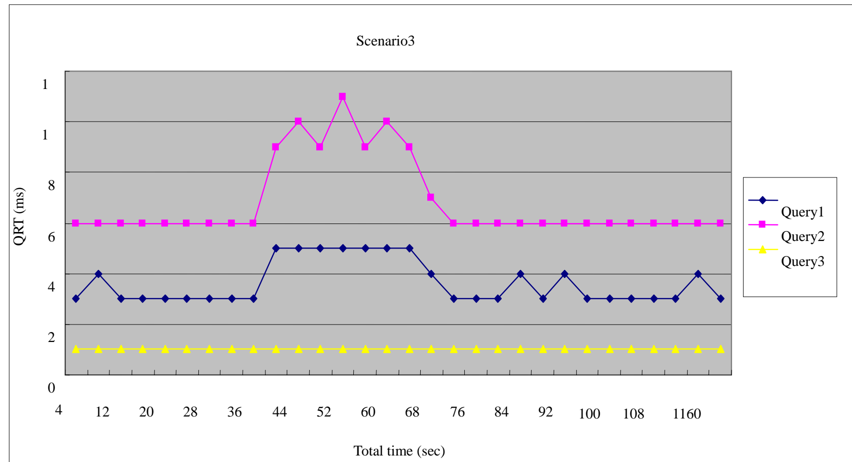


Figure 4 Scenario 3: SQL multiple attribute update

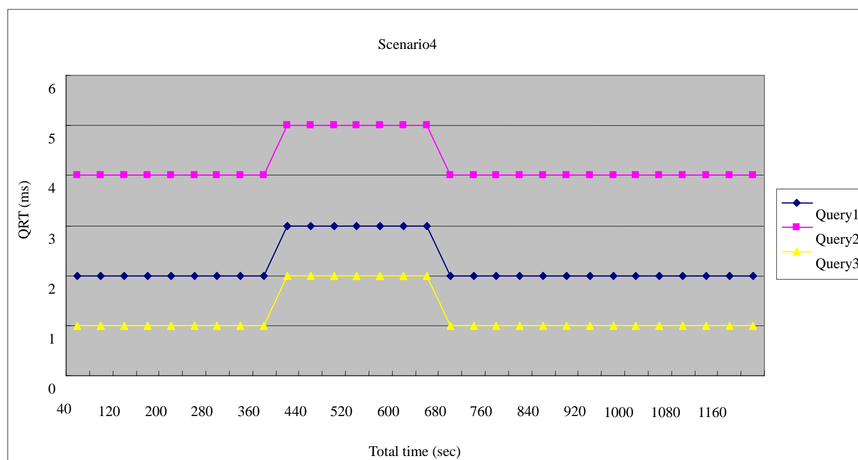


Figure 5 Scenario 4: SQL record insert

Within a scenario graph, the differences between the query plots are differences in the queries. For instance, in Figure 5 Query1 searches for a particular software service, such as a library, on a non-indexed field over a collection that is 640 objects in size. Query2 searches the large Host table on an indexed attribute. Query3 searches a tiny GlueSE collection, 60 objects, on a non-indexed attribute. The update threads, not shown, all add records to the Policy collection.

Across the scenario graphs shown in Figure 3, 4, and 5, the differences are in type of update being performed. In Figures 4 and 5, updates are to the Policy table. In Figure 4 multiple attributes in an object are changed and in Figure 5, a 340 byte object is added. In Figure 3, however, updates overlap with queries in collections accessed.

Comparing the three scenarios of Figures 3, 4, and 5, one can conclude that object insertion has a larger impact on QRT than does update to multiple attributes within an object, but updates that compete for collection access with queries have a much larger impact on QRT. These results were obtained with the InnoDB back end which provides row level locking.

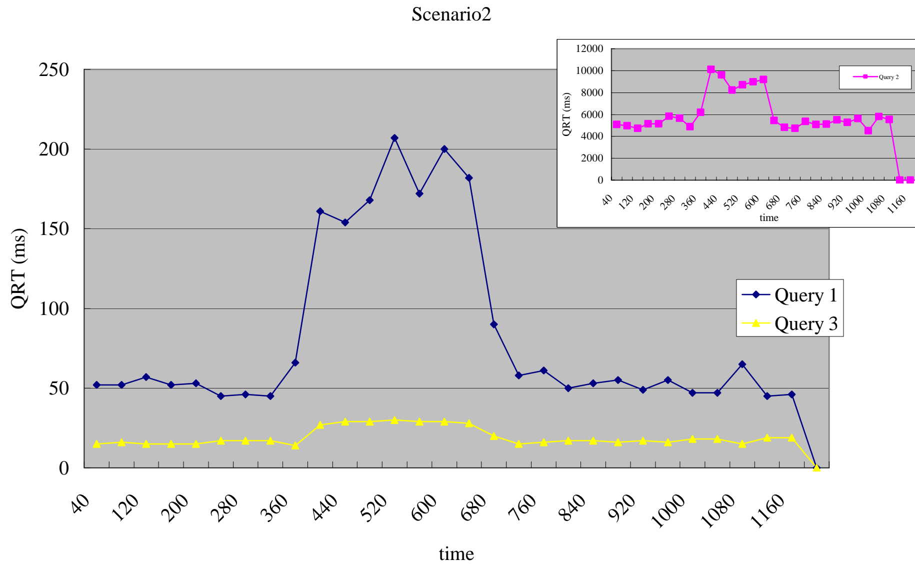


Figure 6 Scenario 2: XML concurrent read/write overlap

Berkeley DB XML performance for the concurrent read/write overlap scenario, shown in Figure 6, further substantiates our observation that the collection size a query accesses is the most cardinal factor relate to query response time. Suppose a query arrives at the database the same instant as an update request that updates a fast-changing variable. A query that takes 7 ms to process under conditions of concurrency on MySQL, will take about 10 seconds on Berkeley DB XML. This latency is clearly unacceptable for a GRIM server. But as comparing with other native XML DB platform, it exposes us great performance while processing these large size xml format files. The remaining scenarios are not shown for Berkeley DB XML because they resemble each other.

4.3 Impact of database size

We performed our test on four databases, the difference between them are the total size occupied. As a result, we find that all numbers increase proportionally to the db size, no matter in MySQL or Berkeley DB XML. Here is the QRT for indexed query on different size of database. We can see that then the database size reach 500 MB, performance from Berkeley DB XML decrease much more than MySQL, as shown from the great angle between blue line and red line. From this point, we can conclude that MySQL, as a mature database product in the market, can indeed handle larger database than Berkeley DB XML. But as we'll see in the next section, the situation changes when we introduce even lager database.

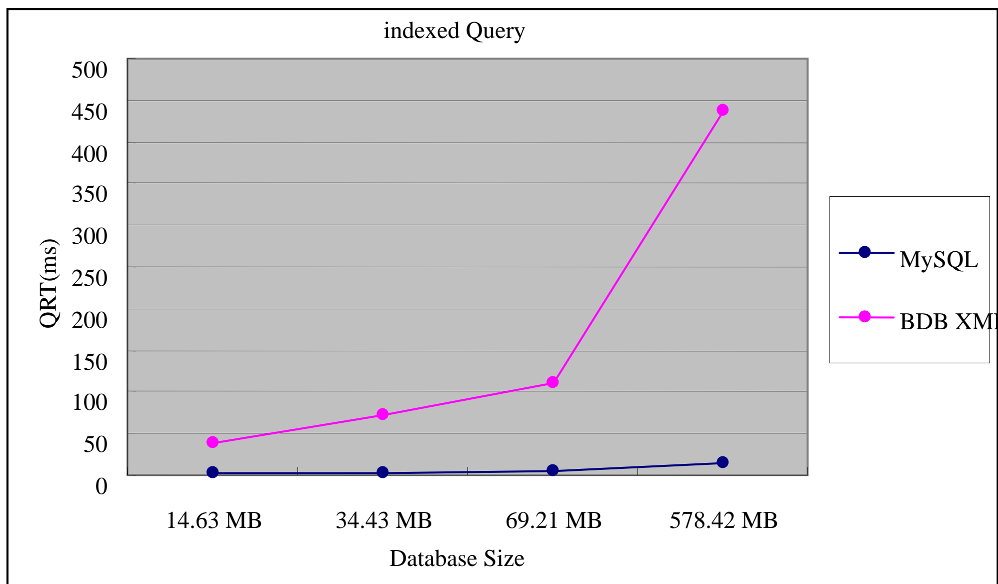


Figure 7 indexed query results from 4 size database

Figures from other queries are not shown here since they all represent same behavior.

Conclusion

A client is often located a long distance away from the grid resource information manager it needs to contact. Thus a platform which minimizes the query response time, as well as the number of requests and amount of data to be transferred, is preferred. Of the two platforms, MySQL overall has significant lower QRT fro all queries in the benchmark. Berkeley DB XML response times are 1-2 orders of magnitude greater than that of MySQL. Compare to other similar XML database (e.g. Xindice), Berkeley DB XML performs greatly since its XQuery based query mechanism. We have reason to believe that if more considerations are applied to the hierarchical structure in the XML database, a great extent of performance enhancement is absolutely possible. But Berkeley DB XML query response time degrades quickly as collection size increases. We noticed that much of the times are spent for extracting attributes from returned xml nodes and building results format. In the synthetic workload test, Berkeley DB XML QRT displays more sensitive to concurrent queries and updates. Specifically, QRT of a query doubles in the presence of simultaneous simple queries and quadruples under updates, while MySQL QRT times are minimally affected by the relatively light workloads.

Acknowledgments

We are grateful to the following people who have contributed ideas and probing insight to this work: Ying Liu, Andrew J Ragusa, Scott Jensen.

Reference

[1] Beth Plale. Resource Information Management in Grid Middleware: Evaluation of multiple Platforms with a Benchmark/Workload.
[2] DataTAG. Glue schema: common conceptual data model for grid resources monitoring and discovery. http:// www.cnaf.infn.it/~sergio/datatag/glue, 2003.
[3] Oracle Berkeley DB XML. http://www.oracle.com/technology/products/berkeley-db/xml/index.html
[4] W3C XPath and XQuery Functions and Operators. http://www.w3.org/TR/xquery-operators/

Appendix A

Query Set Details

- **Scoped Query**

SQL Query

```
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;
```

XQuery

```
query '
for $csc in collection("Cluster_SubCluster.dbxml")/Cluster_SubCluster[@ClusterID="mds.sdsc.edu"]
for $sco in collection("SubCluster_OperatingSystem.dbxml")/SubCluster_OperatingSystem[@SubClusterId=$csc/@SubClusterId]
for $scp in collection("SubCluster_Processor.dbxml")/SubCluster_Processor[@SubClusterId=$csc/@SubClusterId]
return concat("ClusterID: ", data($csc/@ClusterID), " SubClusterId: ", data($csc/@SubClusterId), " OSId: ", data($sco/@OSId), " ProcessorId: ",
data($scp/@ProcessorId))
'
```

- **NonScoped Query**

SQL Query

```
SELECT SCO.OSId, SCO.SubClusterId
FROM SubCluster_OperatingSystem as SCO
WHERE SCO.OSId = "Mac OS9.1";
```

XQuery

```
query '
for $sco in collection("SubCluster_OperatingSystem.dbxml")/SubCluster_OperatingSystem[@OSId="Mac OS9.1"]
return concat("OSId: ", data($sco/@OSId), " SubClusterId: ", data($sco/@SubClusterId))
'
```

- **ScopedHosts Query**

SQL Query

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

XQuery

```
query '
for $csc in collection("Cluster_SubCluster.dbxml")/Cluster_SubCluster[@ClusterID="perigee.sdsc.edu"]
for $sch in collection("SubCluster_Host.dbxml")/SubCluster_Host[@SubClusterId=$csc/@SubClusterId]
return concat("HostId: ", data($sch/@HostId), " SubClusterId: ", data($csc/@SubClusterId), " ClusterID: ", data($csc/@ClusterID))
'
```

- **NonScopedHosts Query**

SQL Query

```
SELECT H.HostId
FROM Host as H
WHERE H.HostId LIKE "%tamarack%";
```

XQuery

```
query '
for $h in collection("Host.dbxml")/Host[contains(@HostId,"tamarack")]
return    concat("HostId: ", data($h/@HostId))
'
```

- **Indexed Query**

SQL Query

```
SELECT H.HostId
FROM Host as H
WHERE H.SMPLoad1Min = .40 or H.SMPLoad1Min = .45;
```

XQuery

```
query '
collection("Host.dbxml")/Host[@SMPLoad1Min="0.4" or @SMPLoad1Min="0.45"]
'
```

- **NonIndexed Query**

SQL Query

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

XQuery

```
query '
collection("Host.dbxml")/Host[@SMPLoad15Min="0.5" or @SMPLoad15Min="0.4"]
'
```

- **Selectivity1 Query**

SQL Query

```
SELECT Endpoint1_Addr, Endpoint1_Port, Endpoint2_Addr, Endpoint2_Port
FROM ConnectionTlb
WHERE Num_Hops = 20 and Bandwidth_Avail_TCP_SingleStream = 65.10;
```

XQuery

```
query '
for $con in collection("ConnectionTlb.dbxml")/ConnectionTlb[@Num_Hops=20 and @Bandwidth_Avail_TCP_SingleStream="65.1"]
return    concat("Endpoint1_Addr: ", data($con/@Endpoint1_Addr), " Endpoint1_Port: ", data($con/@Endpoint1_Port), " Endpoint2_Addr: ",
data($con/@Endpoint2_Addr), " Endpoint2_Port: ", data($con/@Endpoint2_Port))
'
```

- **Selectivity01 Query**

SQL Query

```
Select    Endpoint1_Addr, Endpoint1_Port, Endpoint2_Addr, Endpoint2_Port
FROM ConnectionTlb
WHERE Num_Hops = 33;
```

XQuery

```
query '
for $con in collection("ConnectionTlb.dbxml")/ConnectionTlb[@Num_Hops=33]
return concat("Endpoint1_Addr: ", data($con/@Endpoint1_Addr), " Endpoint1_Port: ", data($con/@Endpoint1_Port), " Endpoint2_Addr: ", data($con/@Endpoint2_Addr), " Endpoint2_Port: ", data($con/@Endpoint2_Port))
'
```

- **Selectivity10 Query**

SQL Query

```
Select Endpoint1_Addr, Endpoint1_Port, Endpoint2_Addr, Endpoint2_Port, Num_Hops
FROM ConnectionTlb
where Num_Hops > 89;
```

XQuery

```
query '
for $con in collection("ConnectionTlb.dbxml")/ConnectionTlb[@Num_Hops>89]
return concat("Endpoint1_Addr: ", data($con/@Endpoint1_Addr), " Endpoint1_Port: ", data($con/@Endpoint1_Port), " Endpoint2_Addr: ", data($con/@Endpoint2_Addr), " Endpoint2_Port: ", data($con/@Endpoint2_Port))
'
```

- **ManyRelations Query**

SQL Query

```
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 = "sharkestra.uchicago.edu" and CSC.SubClusterId = SCO.SubClusterId and CSC.SubClusterId = SCP.SubClusterId and
    CSC.SubClusterId = SCA.SubClusterID and SCA.Pid = AP.Pid and SCO.OSId = "Linux7.1" and SCP.ProcessorId = "PENTIUMIV" and
    AP.RunTimeEnvironment = "Globus 2.2" and AP.Status = "NOT RUNNING";
```

XQuery

```
query '
for $csc in collection("Cluster_SubCluster.dbxml")/Cluster_SubCluster[@ClusterID="sharkestra.uchicago.edu"]
for $sco in collection("SubCluster_OperatingSystem.dbxml")/SubCluster_OperatingSystem[@SubClusterId=$csc/@SubClusterId and @OSId="Li
nux7.1"]
for $scp in collection("SubCluster_Processor.dbxml")/SubCluster_Processor[@SubClusterId=$csc/@SubClusterId and @ProcessorId="PENTIUM
IV"]
for $ap in collection("Application.dbxml")/Application[@RunTimeEnvironment="Globus 2.2" and @Status="NOT RUNNING"]
for $sca in collection("SubCluster_Application.dbxml")/SubCluster_Application[@SubClusterId=$csc/@SubClusterId and @Pid=$ap/@Pid]
return concat("OSId: ", data($sco/@OSId), " ProcessorId: ", data($scp/@ProcessorId), " SubClusterId: ", data($csc/@SubClusterId), " Pid: ",
 data($ap/@Pid))
'
```

- **JobSubmit Query**

SQL Query

```
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 = "golden.tacc.utexas.edu" and CSC.SubClusterId = SCA.SubClusterId and SCA.Pid = App.Pid and App.Owner =
```

```
"amahabal" and App.OS = "Linux" and App.Source_pathname = "/u" and CM.UserId = "amahabal" and UA.ActivationDate < now() and
UA.ExpirationDate >= now() and UA.UserId =    CM.UserId and CSC.ClusterId = CM.ClusterId and M.SubClusterId = CSC.SubClusterId;
```

XQuery

```
query '
for $csc in collection("Cluster_SubCluster.dbxml")/Cluster_SubCluster[@ClusterID="golden.tacc.utexas.edu"]
for $app in collection("Application.dbxml")/Application[@Owner="amahabal" and @OS="Linux" and @Source_pathname="/u"]
for $cm in collection("ClusterMembership.dbxml")/ClusterMembership[@UserId="amahabal" and @ClusterId=$csc/@ClusterID]
for $ua in collection("UserAccounts.dbxml")/UserAccounts[@UserId=$cm/@UserId and @ActivationDate<current-date() and @ExpirationDate>
=current-date()]
for $sca in collection("SubCluster_Application.dbxml")/SubCluster_Application[@SubClusterId=$csc/@SubClusterId and @Pid=$app/@Pid]
for $m in collection("MainMemory.dbxml")/MainMemory[@SubClusterId=$csc/@SubClusterId]
return concat("SubClusterId: ", data($csc/@SubClusterId), " RAMSize: ", data($m/@RAMSize), " Pid: ", data($sca/@Pid), " Owner: ", data
($app/@Owner), " OS: ", data($app/@OS))
'
```

Appendix B

Scenario Workloads

Scenario1: Easy Concurrency

Query1: SELECT SubClusterId FROM SubCluster_Processor WHERE ProcessorId = "Cyrix"

Query2: SELECT SCO.OSId, SCO.SubClusterId FROM SubCluster_OperatingSystem as SCO WHERE SCO.OSId = "Mac OS"

Query3: SELECT SCO.OSId, SCO.SubClusterId FROM SubCluster_OperatingSystem as SCO WHERE SCO.OSId = "AIX"

Update1: UPDATE Application SET Minosv = Minosv+1 WHERE Arch='sparc';

Update2: UPDATE Application SET Minosv = Minosv+1 WHERE Arch='sparc';

Update3: UPDATE GlueSE SET GlueSEPort = GlueSEPort + 1 WHERE GlueSEPort > 10000;

Scenario2: Partial Overlap

Query1: SELECT * FROM Application WHERE Arch = 'Intel Pentium IV';

Query2: SELECT * FROM Host WHERE SMPLoad1Min = 0.84;

Query3: SELECT * FROM GlueSE WHERE GlueSEPort < 35000;

Update1: UPDATE Application SET Minosv = Minosv + 1 WHERE Arch = 'sparc';

Update2: UPDATE Host SET ProcLoad15Min = 0.9999*ProcLoad15Min WHERE SMPLoad1Min = 1.24;

Update3: UPDATE GlueSE SET GlueSEPort = GlueSEPort + 1 WHERE GlueSEPort > 10000;

Scenario3: Muti-attributes Update

Query1: SELECT * FROM Application WHERE Arch = 'Intel Pentium IV';

Query2: SELECT * FROM Host WHERE SMPLoad1Min = 0.84;

Query3: SELECT * FROM GlueSE WHERE GlueSEPort < 35000;

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

Scenario4: Insert Record

Query1: SELECT * FROM Application WHERE Arch = 'Intel Pentium IV';

Query2: SELECT * FROM Host WHERE SMPLoad1Min = 0.84;

Query3: SELECT * FROM GlueSE WHERE GlueSEPort < 35000;

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

Appendix C

We also had a chance to visit a data warehouse product company Netezza (http://www.netezza.com) to run our benchmark queries and workload scenarios on their server, to give us a more intuitionistic comparison between NPS (Netezza Performance Server) and our MySQL 5.0 platform. Although the comparing is a little bit unfair, considering the high cost of a NPS machine and the free MySQL platform, we still can learn many about their strategy to manipulate data and get knowledge from the experiment.

Introduction to NPS (Netezza Performance Server)

The NPS system is a fresh design for tera-scale data warehousing and created an architecture that eliminates the barriers to performance of traditional systems (like inefficient use of administrators' time, inefficient installation, inefficient system management, inefficient data flow, etc). The NPS system combines server, storage and database in a single scalable platform to increase the integration ability of the whole appliance. Among all those novel features provided by Netezza, the most valuable ones are its Asymmetric Massively Parallel Processing architecture and pendent pending Intelligent Query Streaming Technology.

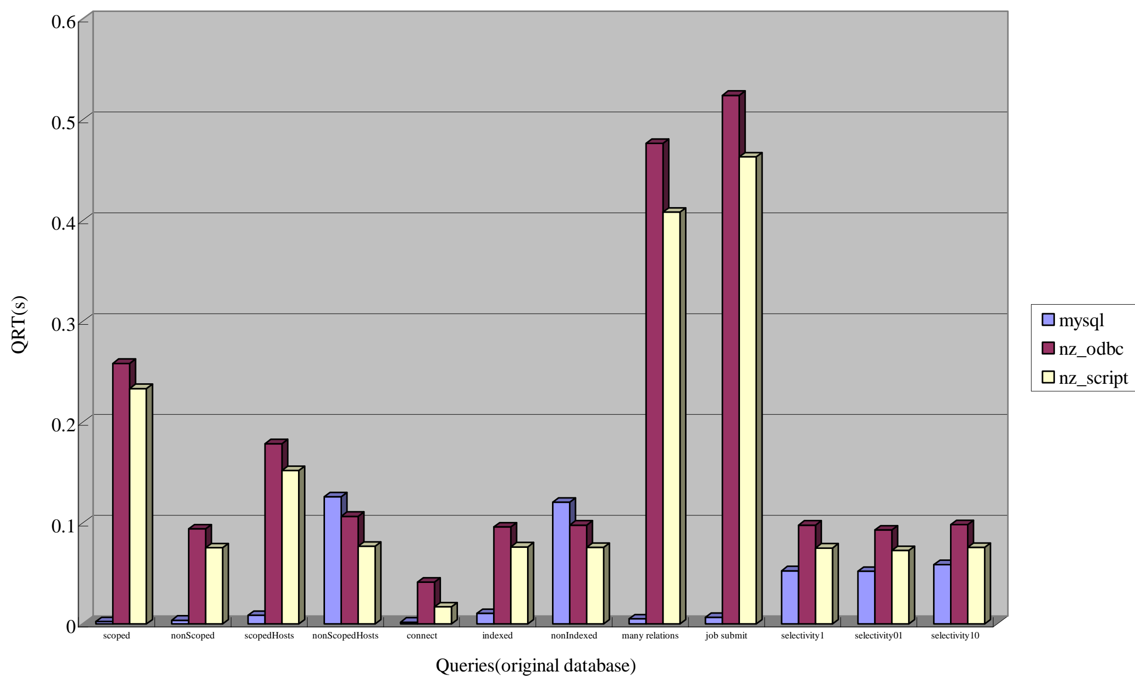
Netezza's AMPP architecture contains a Linux SMP host and dozens to hundreds or thousands of Snippet Processing Units operating in parallel. The SMP host compiles received queries and generates query execution plans, then divides a query into a sequence of sub-tasks and distributes the tasks to each SPU to do the real work. The SMP also takes the responsibility to distributed data to all those storage devices; each SPU in NPS is an intelligent query processing and storage node, which consists of a commodity processor (seems a P2.8G CPU), dedicated memory (1GB Ram), a disk drive (WestData 400GB) and a FPGA controller. The processor is responsible only to its own disk storage. The MPP architecture makes best use of the scalability and the Gigabytes Ethernet Wires used to connect them make the data transfer between each unit more quick and easy.

The Intelligent Query Streaming Technology is implemented in the FPGA chip with hard-wired logic to manage data flows at the disk level. The key point of the good performance of NPS is the fast data moving. The FPGA chip could query and analyze the database at the disk level, so that to alleviate the burden of moving large size data along I/O bus to memory which is a barrier in traditional solution. By comparing with other product, the NPS expose more efficient performance, which is mostly attribute to the functionality of FPGA. Also due to the SMP-SPU architecture, the scalability of NPS is also increased fast against the Moore's Law.

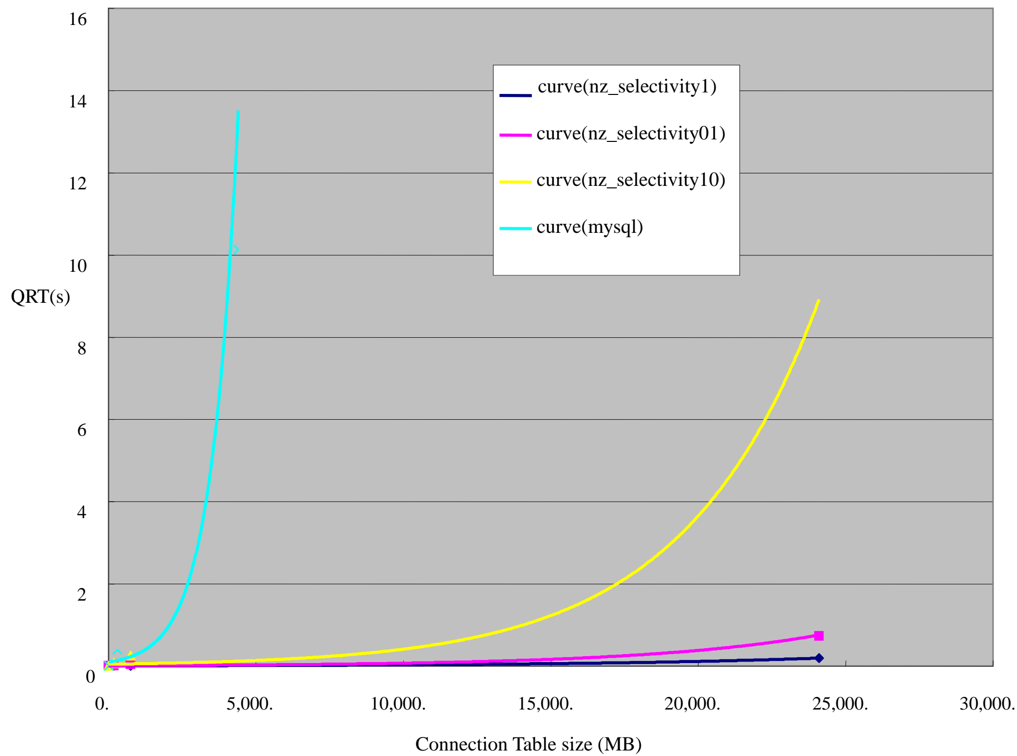
Our object at Netezza is to try to install and run our RGRBenchmark on NPS system, and speed it against MySQL 5.0, from which we expect to learn the whole process of configuring the system and to get an exciting performance result from our test programs.

Test and result

We spent sometime on configuring the environment. In order to build a comparable circumstance, we need the same data model, same database structure. Our implementation of workload benchmark contains scripts to create relation tables and populate data. But there's also many logic contains in the data, so the whole process is controlled by perl script. But NPS was not yet ready for Perl environment, and also their distributed database platform had not efficient interface for Perl's DBI connection and ODBC library, so we only did those benchmark tests which contain simple logic. Here is the figure for the performance comparison.



As we mentioned above, NPS is designed to handle large scale data storing and manipulation. This is achieved by distributed data into several server and run the query parallel. Since our test database is too small (only 50 Mb), the benefit from parallel querying is overshadowed by the data distributing and query parse. That's why we got the result that MySQL 5.0 perform better than a million dollars NPS.



But while we increase the database size on NPS, the great performance is exposed to us. Here is one example in which we run the selectivity queries on different size connection table. The connection table is the larges one in the whole relation database, and the selectivity queries depend only on this table. We can see that while the connection table's size increases, the query response time also increases exponentially, but much more slow in NPS's case. Actually, we can perform the query under a 25GB connection table in NPS, but the query can not even finish in a 5GB connection table in MySQL 5.0.