Using XML and XQuery for Data Management in HPSS

Michael Meseke (Author)
IBM GBS Federal
Houston, TX
Michael.Meseke@us.ibm.com

***Abstract*— The XML and XQuery language capabilities of modern databases can provide a powerful and flexible method of data management within a mass storage system. Within the High Performance Storage System (HPSS), the implementation of XML and XQuery capabilities for user metadata is called "User-defined Attributes" or UDA. The UDA feature provides a method for user applications to associate arbitrary metadata with HPSS namespace objects and store it in an organized, scalable, and searchable manner using XML. The implementation includes a simple key-value interface as well as exposure of the database's XQuery interface to allow for highly customized and atomic update, retrieval, and namespace-wide search requests. Using this architecture enables HPSS to provide client applications a high degree of flexibility in the storage, management, and access of user-defined metadata. This paper describes a brief history of data management within HPSS as well as the architectural decisions, implementation, and results of the UDA feature. Also discussed are considerations for planning and management of the UDA feature, current and in development UDA solutions created by developers and customers, and possible future data management work within HPSS.**

data management; xml; xquery; HPSS; user-defined attributes; extended attributes

I. INTRODUCTION

As data creation rates continue to soar and files are archived for decades, it becomes more difficult to understand the content and purpose of files in the file system. It is often challenging enough to manage and maintain knowledge of one's personal files, let alone an entire file system comprised of billions of files. This problem is greatly compounded in facilities which may have had thousands of users generating data over half a century or more. How are we to understand what those files contain, how we can use the data, and whether those files are even needed?

A hierarchical file system is excellent at separating data which may otherwise be grouped together. The location of a file in a hierarchical file system is within an ever-narrowing set of categories (e.g. My Documents / My Pictures / Birthdays / Tim's Birthdays / 2010). It is easy to look through Tim's birthday pictures in 2010. Assuming the file system is well organized and all birthday pictures appear in this sub-tree, it is possible, but not as easy, to look through all birthday pictures for all users for 2010. It is probably impossible to use information contained only within this directory structure to identify which birthday pictures have a chocolate cake; cake type is not a category within the existing directory structure.

Often when looking for data in a file system it is desirable to be able to slice the file system contents along different axes based upon specific file characteristics. In order to provide a better understanding of the content, meaning, provenance, and/or organization of the file at a later date, it is necessary to store this information as additional metadata associated with the file.

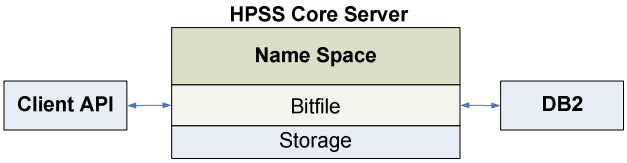

Figure 1. HPSS Namespace Metadata Flow

In order to begin describing our problems and solutions, it is important to have a high level understanding of the impacted areas. HPSS is a hierarchical storage management solution capable of managing petabytes of data across disks and robotic tape libraries. A central component of HPSS in its current version 7 architecture is the Core Server (Figure 1), which handles file namespace, bitfile, and storage operations. It is worth mentioning that in HPSS parlance a bitfile represents file data, which may be spread across multiple levels of the hierarchy. For the topic at hand we focus on namespace operations, especially user metadata. HPSS stores its namespace metadata within a database, and that metadata can be retrieved by external programs using the HPSS Client API library, HSI[1] (HPSS Shell Interface), FTP, or the HPSS Linux virtual file system. [4, 5, 6, 16]

The HPSS User-defined Attributes (UDA) feature was created in order to help users better understand the data they have access to, and more easily discover where data they are interested in may be found. It also provides HPSS tool developers with a standard space to store their application metadata which enables HPSS utilities and client applications to further extend the software.

978-1-4577-0428-4/11/$26.00 ©2011 IEEE

[1] HSI and HTAR are developed by Gleicher Enterprises, LLC.

We will begin by describing the state of HPSS data management prior to the addition of the UDA feature. From there we define the enhancement's goals and describe approaches that were discussed or prototyped by developers and users during the design process, and why each was ultimately rejected. Next, we describe the selected approach, its architecture, and results. Finally, we finish up by discussing possible enhancements to data management within HPSS.

II. A Brief History of User Metadata In File Systems and HPSS

A number of file systems implement space for users and applications to store metadata associated with file system objects, from OS/2 and DOS era file systems to current era file systems. Both extended attributes (JFS/Ext) and file forks (NTFS/HFS) were not architecturally suitable for implementation in the existing HPSS system or were unable to meet our data management requirements.

	Extended Attributes	**File Fork**
Storage	i-node / File Block	Resource File
Storage Limit	1KB to Unlimited	Unlimited
Specific Drawbacks	Performance hit to i-node operations	Object complexity, portability
Other Drawbacks	User metadata not stored separately or differently from data.	

Figure 2. Extended Attributes and File Fork Properties

In the implementation of UDAs described by this paper, the attribute name and value are stored within an external database. This implementation is more akin to the extended attribute concept of storing attributes with other file metadata than it is to the file fork concept of carrying user metadata along as a separate object.

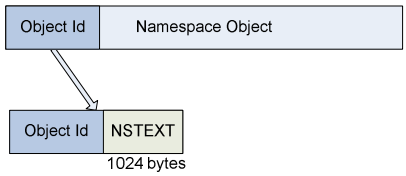

Figure 3. HPSS Comment Field Metadata

Prior to HPSS 7.3 and the UDA feature, HPSS only had a comment field for storing user metadata. It was a fixed width column in the database [5] capable of storing 1 kilobyte of data. The HPSS comment field (Figure 3) was originally created to allow users to add text annotations, but was also used by applications to store metadata. Some advantages of the comment field were that it did not consume space if no comment existed, and access rules were the same as file data access. The comment field's limitations were significant when compared with most modern user metadata implementations. It had limited length and had no inherent capacity to organize multiple sets of data.

Applications and users were not able to share the comment field well unless they had knowledge of which pieces of the comment field were in use by others and how the information was formatted. Sharing the comment field could easily result in its corruption if all parties were not careful and aware of the others' activities.

Many HPSS sites began storing user-defined metadata in a site database to meet their data management requirements. These external, unsupported databases tied metadata to the file using the file's object ID [15]. This was sufficient for many sites despite the cost of developing and maintaining their own custom solution; however, generic HPSS applications such as HSI, HTAR (HPSS TAR), or GHI (GPFS-HPSS Interface) could not take advantage of these site databases [11, 16, 17] because of their customized nature. These applications either used the comment field or developed their own version of forked files, unsupported by HPSS or other tools.

III. Initial Approaches

We had several requirements for building an effective data management system for HPSS. One of these was the ability to understand more about a file than can be contained in a filename or inferred from a file's position in the hierarchy. Another was the ability to find files with a set of desired characteristics. Other requirements were to minimize the work required to manage the new feature, minimize its storage requirements, and maximize flexibility and reuse of existing features and strengths of the software stack.

We considered several different approaches in the design and development of a data management feature in HPSS. Among these were creating each attribute as its own table within the database, creating all attributes as a single database table of key-value pairs, and using a single table with an XML column per file to store the attributes.

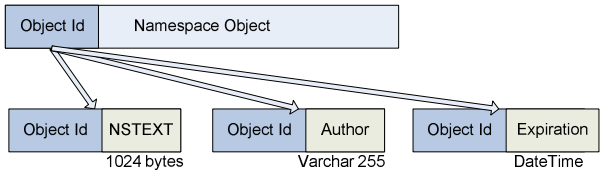

Figure 4. Multiple Attribute Tables with "Author" and "Expiration" tables

Using multiple attribute tables (Figure 4) had major downsides. The burden was placed on the administrator to create and register new attribute tables whenever they were needed. Doing searches or attribute listings would be difficult as it could require joining dozens or hundreds of tables together.

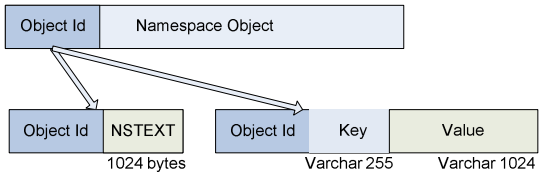

Figure 5. Single Attribute Table with Key Value Pairs

A single attribute table (Figure 5) would solve many of these issues, but was still limiting. In order to search, a query language would need to be devised, and a method of restricting the attributes implemented. Indexes would also be difficult to add to this table since even though all of the attributes were text, the types of values may not be.

IV. SELECTED APPROACH – XML TABLE

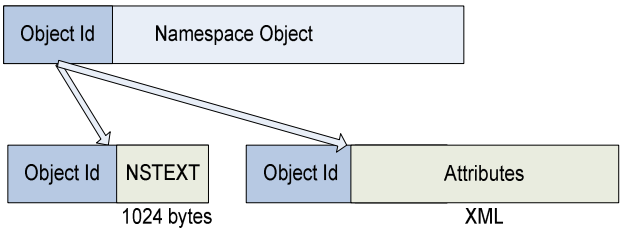

Figure 6. XML Table

The approach we chose was to use an XML table (Figure 6). This approach met all of the requirements we identified at inception as well as the requirements which were added while pursuing other designs. Using an XML table provided all the benefits of the single key-value table while also allowing us to resolve its most serious drawbacks. XML may seem like an odd choice given its verbosity; however, this has largely been addressed using compression (Figure 13).

The first benefit was database support for standard XML query languages, XQuery and XQuery Update [18], which provide the capability to query and modify XML data similar to how SQL does for relational data. We found that we could open the HPSS APIs up using XQuery instead of creating our own query language, thus minimizing development effort.

Indexes within the XML table could also be defined on a per-XPath basis, which would allow them to be leaner and more specialized than those of the single key-value table [3, 7]. Wildcards and multiple indexes per XPath could also be defined, which would allow for more flexibility in how indexes were used with data.

XML Schema Definitions could be used to control which attributes may exist and what values they may take on. Administrators could load and set a default schema in the database which would always be used for the UDA table, or multiple schemas could be loaded to allow programmers to specify a schema to use to validate attribute changes. This could be useful in cases where several classes of files exist which may contain different attributes.

There is no limit for the number of attributes per file. Each file can contain up to 2 gigabytes of XML. This is a DB2 limitation for the size of an XML row [1]. The scalability of the namespace as a whole and the total user metadata associated with it is limited by metadata space allocated to the database. Additional metadata disk can be added as necessary using standard DB2 procedures.

HPSS provides the flexibility to use a separate database instance for UDAs. This may provide better manageability if the requirements of a site warrant dedicated hardware.

A. Database

DB2's PureXML engine was heavily utilized in the chosen architecture by providing functionality such as XML indexing, XML schema validation, XQuery, compression and federation. By using a COTS product we were able to skip a lot of development and test effort which would have gone into reproducing these features from scratch.

B. Core Server

The Core Server, which manages the HPSS namespace, bitfile, and storage components of HPSS, formats requests to facilitate communication between the API and the database, and controls access to UDA like typical file access. The search APIs are currently only available to those with "control" permission in HPSS – usually administrators or authorized applications.

The Core Server also uses a new module, HPSS Cursor Manager, to enable client requests to maintain an open cursor across requests. This allows clients to retrieve large result sets by making multiple requests using the same cursor handle.

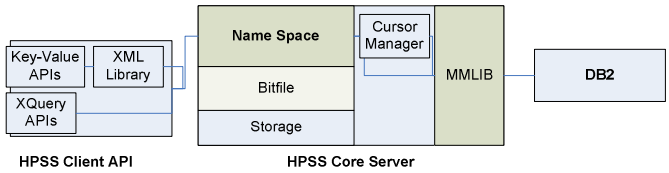

Figure 7. HPSS User-defined Attribute Architecture

C. Client API

The Client API provides two sets of APIs, one which is easy to use but relatively inflexible and another which is more complex but provides access to the flexibility and power of the XQuery interface. These two APIs sets share interfaces into HPSS by translating the key-value pairs of the simple APIs into XQuery retrieval and update statements. Several of the APIs appear in Figure 8 as an illustration. The first is an example of a key-value API. The hpss_userattr_list_t takes an array of key-value pairs. The hpss_UserAttrXQueryUpdate function does not take key-value pairs, but rather takes only an XQuery string.

```
int
hpss_UserAttrSetAttrs(
char   *Path,       /*IN-File Path*/
hpss_userattr_list_t    *Attr, /*IN-Attributes*/
char   *Schema); /*IN-Schema*/
int
hpss_UserAttrXQueryUpdate(
char  *Path,    /*IN-File Path*/
char  *XQuery,  /*IN-XQuery*/
char  *Schema); /*IN-Schema*/
```

Figure 8. A sample of key-value and XQuery APIs

V. RESULTS

Tests were executed to measure the scalability and throughput of the solution. The test system was running HPSS 7.4.0 under AIX 6.1, DB2 9.7, and a default table configuration. More detailed information may be found in Figure 14. Performance of the User-defined Attributes operations is heavily based upon the performance and configuration of the database. This configuration is not perfect, mostly due to the database being spindle poor; however, the results can still illustrate important characteristics of the various operations.

The XQuery APIs are capable of performing multiple operations at a much higher rate than the key-value APIs. The key-value APIs make each key-value request or update separately while all of the requests or updates of an XQuery processes all operations within a single request. The result is that as more key/value pairs are added, performance of those APIs drops proportionally. A properly defined XQuery does not exhibit this behavior. Results for the key-value APIs are not presented here.

A. Access, Update, Delete

The HPSS Client API provides interfaces for accessing and updating UDAs on a single file. XQuery statements were supplied directly by the user.

1) User-defined Attributes Update / Delete

Performance testing for update focused on the performance of a typical insert / update with a variable number of clients. The attributes inserted were of minimal size. The XQuery APIs were tested using a query similar to Figure 9:

```
copy $new := $DOC modify( do insert <a>p</a> into $new/hpss, do insert <b>p</b> into $new/hpss, do insert <c>p</c> into $new/hpss, do insert <d>p</d> into $new/hpss) return $new
```

Figure 9. An Example XQuery which inserts four attributes, /hpss/a, /hpss/b, /hpss/c, and /hpss/d.

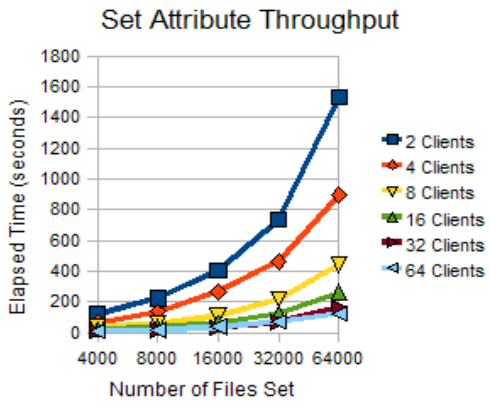

Figure 10. Update scalability using the XQuery API set with variation of clients and attributes.

Attribute updates scaled well up to 64 clients setting 64,000 files. Once the test reached 32 clients disk and CPU limitations began to be hit sporadically; once the client count was increased to 64 those limitations were hit more often. The majority of the CPU load was on the database (70-75%) with the HPSS Core Server using only 8-10%.

2) User-defined Attribute Access

Retrieval testing was done by issuing an XQuery request to retrieve the metadata which was created in the update test.

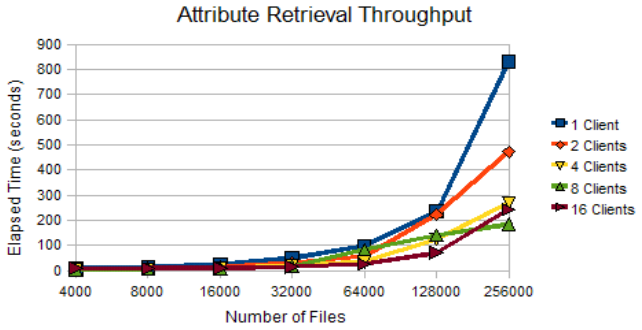

Figure 11. Retrieval scalability using the single XPath method with variation of clients and attributes.

Our eventual bottleneck during retrieval testing was CPU usage on the Core Server machine. Each of the 4 dual-core processors was pegged at 100% utilization when 16 concurrent processes were used to get attributes. The major users of CPU resources were the database (55-60%) followed by the HPSS Core Server (35-40%). Eight clients was the sweet spot on the machine we tested on – the CPU utilization was roughly 85% across all cores which allowed the configuration to scale up to 256,000 retrievals without encountering that bottleneck.

B. Search

The HPSS Client API provides interfaces for retrieving sets of HPSS object IDs which match a query. Searches were run with no matches, matching all files, matching a set of files equal to roughly 75% of the total number of UDA files (/hpss/a), and a very small result set (/hpss/color=green) (Figure 12). Each valid search term (/hpss/a and /hpss/color) was indexed in the database.

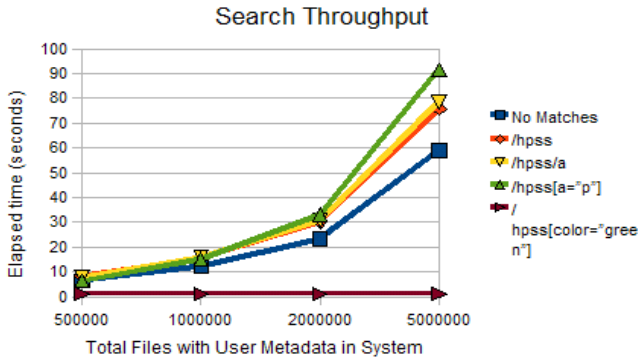

Figure 12. Object search files found per second scalability with variation of clients and number of files

The results here show that our results are very good as the number of files with user metadata grows so long as the

expected result set is fairly small. The result set which contained the majority of the files had mixed results when using the index. The time on the small result set (3 matches out of 5,000,000 files) was never more than 1.5 seconds. The total number of files in the system did not impact this test because only files which have user metadata have a row in the user metadata table.

This API only returns the object ID which matches the query. The object ID, when packaged together with Core Server information, is usable in Client API functions as a namespace object handle without a great deal of additional overhead.

C. Compression of UDAs

One of the chief arguments against XML is that it is verbose; however, this was minimized in our implementation by using database compression. An IBM DeveloperWorks article claims compression rates up to 80% [2]. To test the compression of UDAs in HPSS we used DB2 tools which estimate compression rates based upon statistical sampling of the table. The estimated compression rates were gathered from internal and external sites (Figure 13).

System	Files	Compression
GHI Development System	16 million	68%
PNNL Production System	22 million	70%
HPSS System Test	100,000	70%
Benchmark System	320,000	71%

Figure 13. XML compression estimates in test and production environments

VI. In Development and Future Usage of UDA

Since the initial phase of the UDA feature was completed there have been a number of applications created which rely on the functionality, with more being developed.

A. GHI Garbage Collection and Aggregate Statistics

GHI, the GPFS-HPSS Interface, has a garbage collection feature which allows GHI to reclaim storage space which is no longer referenced within their system. GHI uses UDAs to keep track of states of GPFS files stored in HPSS. [14, 16].

B. HPSS Checksum Tool

The *hpsssum* tool provides checksum support for HPSS files using a variety of checksum algorithms. A method for storing checksum digests is provided by UDAs. This makes automatic verification of HPSS files using the stored checksum possible.

UDAs also work with SELinux and HPSS VFS to provide mandatory access control functionality within HPSS. [6]

VII. Potential Future Work

The following thoughts and ideas resulted from usage of the current solution to provide additional function and value. Future work will be focused on improving the integration of the solution across HPSS to provide additional value and meet any new requirements.

A. Standard File Attribute / User-defined Attributes Integration

Being able to search and retrieve the UDAs is useful, but there are times when it would be valuable to retrieve or search standard file attributes as well as user metadata. To provide a unified perspective, system and user metadata could be merged together into a single cohesive view. HPSS has an policy that APIs may only be modified during a major release, which has kept us from integrating UDAs more deeply with standard file attributes. File attribute retrieval APIs will be looked at more deeply during the development of the next major release.

B. Alternative Navigation Methods

In a system where billions of files are stored and directories of millions of files is considered normal, human users may need alternative methods of finding and accessing their data. UDAs could be used to implement a database file system navigation layer to avoid a complex and overwhelmingly large hierarchical namespace. HPSS is required to support a POSIX namespace; however, it may be necessary or expedient to hide it in the future.

VIII. Decisions and Relation to Other Works

Speaking upon our approach and results in broader terms, utilizing XML, XQuery, and a database for data management provides a powerful, organized, and flexible method for managing data. There are a number of areas where this solution could be modified to meet a different set of requirements by tweaking the architecture.

Using an RDBMS (Relational Database Management System) provided this solution with better consistency, integrity, and atomicity than a noSQL solution at the cost of some lesser performance and scalability. For our purposes, using an RDBMS was extremely convenient and allowed the solution to integrate easily into a complex, established architecture. A noSQL solution could serve just as well in the role of our metadata backend in other solutions where performance requirements are higher and consistency and reliability requirements may be lower. [19]

Other naming conventions and authorities may be required if the number of potential applications and users is extremely high. For our purposes, the number of typical applications running against the solution is typically very small at any given installation, and so segregating metadata based upon application name has worked well for us. For solutions which expect more open development and usage in their deployments, a Java or Perl-like naming authority scheme could be used. A strict naming authority could also be used by requiring that new applications be registered with the metadata

engine prior to storing their metadata, and having the metadata engine assign them a unique metadata path.

While one of our goals was to maximize flexibility, allowing the feature too much power posed a risk to the security of the system. There were two capabilities which were removed because we perceived that they were more likely to be dangerous than to be useful. One was limiting an XQuery Update to acting on a single namespace object at a time. In XQuery it is perfectly valid to update multiple rows at a time; however, this feature was unnecessary for our implementation and we perceived it to more likely wind up being a major pitfall for users. Also, we do not allow the DB2-specific function *db2-fn:sqlquery* to appear in the user-supplied XQuery commands. This special function allows the user to execute arbitrary queries against the database, which we deemed was too risky for general purpose usage. In another solution these behaviors may have been desirable despite the high risk associated with them.

IX. CONCLUSION

User metadata storage has long been a desired feature for file systems in general and for HPSS users in particular. UDAs have been a topic of high interest within the HPSS community since their inception, and have already proven their worth in production environments as well as within the development community.

We have developed a replacement for the short, static-length, unorganized HPSS comment field. The replacement has significant capacity when compared to many solutions, and exists within a database to provide additional protection, validation, and searchability.

The UDA solution is unique in meeting all of our requirements. By taking advantage of capability within DB2, we provided a tool for data management which has been used to extend HPSS, and created a solution which can be tailored to the requirements of the customer, from providing a simple key-value storage and retrieval system to something more like a content management system with indexed lookups and schema validation. Performance is based upon a number of environmental factors as well as the types of operations used; however, on our benchmark system the feature delivered performance in line with other HPSS metadata operations.

In the future, HPSS developers and 3rd party programmers can take this work further by creating new tools and enhancements which utilize the flexibility and power of the XQuery interface beyond what can be done with the simple interfaces. For now, UDAs are firmly in place within the HPSS user community as a method to help ease the burden of managing multi-petabyte archives by allowing users to better understand what they have and where it may be found.

REFERENCES

[1] "SQL and XML Limits." IBM DB2 9.7 for Linux, UNIX, and Windows Information Center. IBM. May 2006. Web. 8 Dec. 2010.

[2] Ahuja (IBM), Rav. "Introducing DB2 9, Part 1: Data Compression in DB2 9." IBM - United States. 24 May 2006. Web. 08 Dec. 2010.

[3] "Compression Dictionary Creation." IBM DB2 9.7 for Linux, UNIX, and Windows Information Center. IBM. Web. 8 Dec. 2010.

[4] IBM. HPSS 7.3.2 Programmer's Reference. 1st ed. Houston: IBM, 2010. hpss-collaboration.com. Apr. 2010. Web. 15 Dec. 2010.

[5] IBM. HPSS 7.3.2 Installation Guide. 1st ed. Houston: IBM, 2010. hpss-collaboration.com. Aug. 2010. Web. 15 Dec. 2010. IBM. HPSS 7.3.2 Installation Guide. 2010. PDF.

[6] IBM. HPSS 7.3.2 Management Guide. 1st ed. Houston: IBM, 2010. hpss-collaboration.com. Aug. 2010. Web. 15 Dec. 2010.

[7] Nicola, Matthias. "15 Best Practices for PureXML Performance in DB2." IBM - United States. 26 May 2009. Web. 15 Dec. 2010.

[8] Teaff, D., R. W. Watson,, and R.A Coyne. "The Architecture of the **High** Performance Storage System (HPSS)," Proceedings of the Goddard Conference on Mass Storage & Technologies, College Park, March 1995

[9] Coyne, R. A. and Hulen, H. (1993). An Introduction to the Mass Storage System Reference Model, Version 5. In Proceedings of the Thirteenth Symposium on Mass Storage Systems, pages 47–53.

[10] T.W. Tyler, D.S. Fisher, "Using distributed OLTP technology in a high performance storage system," mss, pp.45, 14th IEEE Symposium on Mass Storage Systems, 1995

[11] HTAR. http://www.mgleicher.us/GEL/htar/

[12] Avantika, Mathur, Andreas Dilger, Alex Tomas, and Laurent Vivier. The New Ext4 Filesystem: Current Status and Future Plans. 2007. Web. 8 Dec. 2010.

[13] Rodrigues, Vitor. "Indexing XML Documents with DB2 9 PureXML." IBM, May 2006. Web. Jan. 2011.

[14] Frank B. Schmuck , Roger L. Haskin, GPFS: A Shared-Disk File System for Large Computing Clusters, Proceedings of the Conference on File and Storage Technologies, p.231-244, January 28-30, 2002

[15] Long, J. W., N. J. O'Neill, N. G. Smith, and R. R. Springmeyer. "Hopper File Management Tool." Proc. of Nuclear Explosives Code Development Conference, Livermore, CA. Llnl.gov, 17 Nov. 2004. Web. 11 Jan. 2011.

[16] IBM. *GHI Management Guide*. Houston: IBM, 2010. *Hpss-collaboration.com*. IBM, July 2010. Web. 15 Dec. 2010.

[17] HSI. http://www.mgleicher.us/GEL/hsi/

[18] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanascu. XQuery: A query language for XML. Technical report, World Wide Web Consortium, February 2001. Available from http://www.w3.org/TR/xquery/.

[19] Stonebraker, Michael. "SQL Databases v. NoSQL Databases." *Communications of the ACM* 53.4 (2010): 10-11. Print.

System Model	AIX 6.1 IBM 8203-E4A Power6 64-bit, non-partitioned
Processors	2 4.7 Ghz Quad Core CPU
Memory	32GB RAM
DB2 Tablespace Storage	IBM 3542-2RU, Dual Controller
DB2 Logging	RAID 10 (2+2)
UDA Tablespaces	Database Managed Tablespaces, Six raw logical volumes, each as RAID5 (4+1) and shared with other DB2 tablespaces
Table DDL	CREATE TABLE HPSS.USERATTRS (OBJECT_ID BIGINT NOT NULL , ATTRIBUTES XML) IN USERATTRS INDEX IN USERATTRSIDX ;

Figure 14. Test System Configuration